CLIMADA documentation

Release 3.1.2

CLIMADA contributors

Apr 21, 2022

CONTENTS

This is the documentation for version v3.1.2. In CLIMADA-project you will find CLIMADA's contributors, repository and scientific publications.

CHAPTER

ONE

GETTING STARTED WITH CLIMADA

This is a short summary of the guides to help you find the information that you need to get started. To learn more about CLIMADA, have a look at the introduction. You can also have a look at the paper repository to get an overview of research projects.

1.1 Installation

The first step to getting started is installing CLIMADA. To do so you will need: 1. To get the lastest release from the git repository CLIMADA releases or clone the project with git if you are interested in contributing to the development. 2. To build a conda environment with the dependencies needed by CLIMADA.

For details see the Installation Guide.

If you need to run a model on a computational cluster, have a look at this guide to install CLIMADA and run your jobs.

1.2 Programming in Python

It is best to have some basic knowledge of Python programming before starting with CLIMADA. But if you need a quick introduction or reminder, have a look at the short Python Tutorial. Also have a look at the python Python Dos and Don't guide and at the Python Performance Guide for best practice tips.

1.3 Tutorials

A good way to start using CLIMADA is to have a look at the Tutorials. The Main Tutorial will introduce you the structure of CLIMADA and how to calculate you first impacts, as well as your first appraisal of adaptation options. You can then look at the specific tutorials for each module (for example if you are interested in a specific hazard, like Tropical Cyclones, or in learning to estimate the value of asset exposure,...).

1.4 Contributing

If you would like to participate in the development of CLIMADA, carefully read the Git and Development Guide. Before making a new feature, discuss with one of the repository admins (Now Chahan, Emmanuel and David). Every new feature or enhancement should be done on a separate branch, which will be merged in the develop branch after being reviewed (see Checklist). Finally, the develop branch is merged in the main branch in each CLIMADA release. Each new feature should come with a tutorial and with Unit and Integration Tests.

1.5 Other Questions

If you have any other questions, you might find some information in the Miscellaneous guide. If you cannot find you answer in the guides, you can open an issue for somebody to help you.

CHAPTER

TWO

INTRODUCTION

CLIMADA implements a fully probabilistic risk assessment model. According to the IPCC *[1]*, natural risks emerge through the interplay of climate and weather-related hazards, the exposure of goods or people to this hazard, and the specific vulnerability of exposed people, infrastructure and environment. The unit chosen to measure risk has to be the most relevant one in a specific decision problem, not necessarily monetary units. Wildfire hazard might be measured by burned area, exposure by population or replacement value of homes and hence risk might be expressed as number of affected people in the context of evacuation, or repair cost of buildings in the context of property insurance.

Risk has been defined by the International Organization for Standardization as the "effect of uncertainty on objectives" as the potential for consequences when something of value is at stake and the outcome is uncertain, recognizing the diversity of values. Risk can then be quantified as the combination of the probability of a consequence and its magnitude:

$$risk = probability \times severity$$

In the simplest case, $\times$ stands for a multiplication, but more generally, it represents a convolution of the respective distributions of probability and severity. We approximate the $severity$ as follows:

$$severity = F(hazard\ intensity, exposure, vulnerability) = exposure * f_{imp}(hazard\ intensity)$$

where f_{imp} is the impact function which parametrizes to what extent an exposure will be affected by a specific hazard. While 'vulnerability function' is broadly used in the modelers community, we refer to it as 'impact function' to explicitly include the option of opportunities (i.e. negative damages). Using this approach, CLIMADA constitutes a platform to analyse risks of different hazard types in a globally consistent fashion at different resolution levels, at scales from multiple kilometres down to meters, depending on the purpose.

2.1 References

[1] IPCC: Climate Change 2014: Impacts, Adaptation and Vulnerability. Part A: Global and Sectoral Aspects. Contribution of Working Group II to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change, edited by C. B. Field, V. R. Barros, D. J. Dokken, K. J. Mach, M. D. Mastrandrea, T. E. Bilir, M. Chatterjee, K. L. Ebi, Y. O. Estrada, R. C. Genova, B. Girma, E. S. Kissel, A. N. Levy, S. MacCracken, P. R. Mastrandrea, and L. L. White, Cambridge University Press, United Kingdom and New York, NY, USA., 2014.

CHAPTER

THREE

INSTALLATION

Please execute the instructions of the following text boxes in a Terminal or Anaconda Prompt. Chose an installation directory, e.g., the user's home directory. In the following it will be referred to as `$[installation directory]`.

3.1 Install CLIMADA from sources

3.1.1 Download the latest version

```
cd $[installation directory]

git clone https://github.com/CLIMADA-project/climada_python.git
cd climada_python
git checkout develop
cd ..
```

3.1.2 Set up the environment

It's highly recommanded to install CLIMADA into a Conda environment. Without Conda, the installation of the dependencies can be cumbersome. If it is not already installed, download the latest version of Anaconda or Miniconda and execute it.

Open a command prompt (Windows) or shell (Mac, Linux) and run:

```
conda env create -n climada_env -f climada_python/requirements/env_climada.yml
conda activate climada_env
```

3.1.3 Install the climada package

Include the CLIMADA path into the Conda environment as follows:

```
pip install -e climada_python
```

With this setup, changes to CLIMADA's source code will immediately take effect. It removes the need to reinstall during development.

3.2 Install CLIMADA from PyPi (alternative)

Published releases of CLIMADA (not CLIMADA-petals) can also be installed from the PyPi repository https://pypi.org/project/climada/, since version 3.0.

3.2.1 Set up the environment

Download the environment specification file from the git repository and save it locally, then create the conda environment:

```
curl -o env_climada.yml https://raw.githubusercontent.com/CLIMADA-project/climada_python/
↪main/requirements/env_climada.yml
conda env create -n climada_env -f env_climada.yml
conda activate climada_env
```

If this was successful, `env_climada.yml` can be safely deleted.

3.2.2 Install the climada package

```
pip install climada
```

3.3 Install CLIMADA-petals (optional)

3.3.1 Download the latest version

```
cd $[installation directory]

git clone https://github.com/CLIMADA-project/climada_petals.git
cd climada_petals
git checkout develop
cd ..
```

3.3.2 Update the environment

It is possible that CLIMADA-petals has slightly different dependencies than CLIMADA-core, hence the conda environment should be updated.

```
conda env update -n climada_env -f climada_petals/requirements/env_climada.yml
conda activate climada_env
```

3.3.3 Install the climada_petals package

Include the CLIMADA-petals path into the Conda environment as follows:

```
pip install -e climada_petals
```

In case of a separate environment for CLIMADA-petals, one needs to install the core package as well, either with the package from PyPi (`pip install climada`) or from local sources (`pip install -e climada_python`)

3.4 Test the installation

Run the following command:

```
python -m unittest climada.engine.test.test_impact climada.engine.test.test_cost_benefit
```

If the installation has been successful, an OK will appear at the end (the execution shouldn't last more than 2 min).

3.4.1 Run tutorials

In the *Home* section of Anaconda, with `climada_env` selected, install and launch `jupyter notebook`. A browser window will show up. Navigate to your `climada_python` repository and open `doc/tutorial/1_main_CLIMADA.ipynb`. This is the tutorial which will guide you through all CLIMADA's functionalities. Execute each code cell to see the results, you might also edit the code cells before executing. See tutorials for more information.

3.5 Working with *Spyder*

It is possible to use climada in combination with the Spyder-IDE. However, depending on OS, CLIMADA version, Spyder version and installation tool, installing `spyder` into the `climada_env` environment may fail. In particular, MacOS and `conda install spyder` don't work well together. Although there are cases where `pip install spyder` bears satisfactory results, it is suggested to install `spyder` in a separate Conda environment and run it with the python interpreter from the `climada_env` environment instead. To do so, follow these instructions:

- Start Anaconda, create a new python_env environment (just click create and enter the name, then press Create) and install latest Spyder in there (currently 4.2.5). Start this version of Spyder. After Spyder has started, navigate to *Preferences* > *Python Interpreter* > *Use the following interpreter* and paste the output of the following terminal command into the text box:

```
conda activate climada_env
python -c "import sys; print(sys.executable)"
# this returns a path, like /Users/XXX/opt/anaconda3/envs/climada_env/bin/python
```

- Obtain the version of the package `spyder-kernels` (currently 1.10.2) that has automatically been installed along with Spyder in the Anaconda environment and install the same version of `spyder-kernels` in the `climada_env` environment:

```
conda activate climada_env
conda install spyder-kernels=1.10.2
```

- Start a new IPython console in Spyder and run `tests_install.py`.

3.6 FAQs

- **ModuleNotFoundError**
  - the CLIMADA packages are not found. Check whether the climada environment is activated. If not, activate it (`conda activate climada_env`). Otherwise, the installation has failed. Try and follow the installation instructions above.
  - an external python library is not found. It might happen that the pip dependencies of `env_climada.yml` (the ones specified after `pip:`) have not been installed in the environment `climada_env`. They can be installed manually one by one as follows:

```
conda activate climada_env
pip install library_name
```

  where `library_name` is the missing library.

  Another reason may be a recent update of the operating system (macOS). In this case removing and reinstalling Anaconda will be required.

- **Conda permission error**

  (`operation not permitted`) in macOS Mojave: try the solutions suggested here https://github.com/conda/conda/issues/8440

- **No 'impf_TC' column in GeoDataFrame**

  This may happen when a demo file from CLIMADA was not updated after the change in the impact function naming pattern from 'if_' to 'impf_' (climada v2.2.0).

  To solve it, run `python -c 'import climada; climada.setup_climada_data(reload=True)` in a terminal.

- **How to change the log level**

  By default the logging level is set to `'INFO'`, which is quite verbose. This can be changed
  - through configuration, by editing the config file `climada.conf` (see Guide_configuration.ipynb) and setting the value of the `global.log_level` property.
  - programmatically, globally, in a script or interactive python environment (Spyder, Jupyter, IPython) by executing e.g.,

```
from climada.util.config import LOGGER
from logging import WARNING
LOGGER.setLevel(WARNING)
```

  - programmatically, context based, using `climada.util.log_level`:

```
from climada.util import log_level
with log_level(level='WARNING'):
      quietly_do_something()
```

CHAPTER

FOUR

USING CLIMADA ON THE EULER CLUSTER (ETH INTERNAL)

4.1 Content

4.1.1 Access to Euler

See https://scicomp.ethz.ch/wiki/Getting_started_with_clusters for details on how to register at and get started with Euler.

For all steps below, first enter the Cluster via SSH.

Installation- and working directories

Please, get familiar with the various Euler storage options: https://scicomp.ethz.ch/wiki/Storage_systems. As a general rule: use `/cluster/project` for installation and `/cluster/work` for data processing.

For ETH WCR group members, the suggested installation and working directories are `/cluster/project/climate/$USER` and `/cluster/work/climate/$USER` respectively. You may have to create the insallation directory:

```
mkdir -p /cluster/project/climate/$USER \
         /cluster/work/climate/$USER
```

4.1.2 Pre-installed version of Climada

Climada is pre-installed and available in the default pip environment of Euler.

4.2 1. Load dependencies

```
env2lmod
module load gcc/6.3.0 python/3.8.5 gdal/3.1.2 geos/3.8.1 proj/7.2.1 libspatialindex/1.8.
↪5 hdf5/1.10.1 netcdf/4.4.1.1 eccodes/2.21.0 zlib/1.2.9
```

You need to execute these two lines every time you login to Euler before Climada can be used. To safe yourself from doing it manually, one can append these lines to the ~/.bashrc script, which is automatically executed upon logging in to Euler.

4.3 2. Check installation

```
python -c 'import climada; print(climada.__file__)'
```

should output something like this:

```
/cluster/apps/nss/gcc-6.3.0/python/3.8.5/x86_64/lib64/python3.8/site-packages/climada/__
↪init__.py
```

4.4 3. Adjust the Climada configuration

Edit a configuration file according to your needs (see Guide_Configuration). Create a climada.conf file e.g., in /cluster/home/$USER/.config with the following content:

```
{
    "local_data": {
        "system": "/cluster/work/climate/USERNAME/climada/data",
        "demo": "/cluster/project/climate/USERNAME/climada_python/data/demo",
        "save_dir": "/cluster/work/climate/USERNAME/climada/results"
    }
}
```

(Replace USERNAME with your nethz-id.)

4.5 4. Run a job

Please see the Wiki: https://scicomp.ethz.ch/wiki/Using_the_batch_system for an overview on how to use bsub.

```
cd /cluster/work/climate/$USER  # change to the working directory
bsub [bsub-options*] python climada_job_script.py  # submit the job
```

4.5.1 Working with Git branches

If the Climada version of the default installation is not according to your needs, you can install Climada from a local Git repository.

4.6 1. Load dependencies

See *Load dependencies* above.

4.7 2. Create installation environment

```
python -m venv --system-site-packages /cluster/project/climate/$USER/climada_venv
```

4.8 3. Checkout sources

```
cd /cluster/project/climate/$USER
git clone https://github.com/CLIMADA-project/climada_python.git
cd climada_python
git checkout develop  # i.e., your branch of interest
```

4.9 4. Pip install Climada

```
source /cluster/project/climate/$USER/climada_venv/bin/activate
pip install -e /cluster/project/climate/$USER/climada_python
```

4.10 5. Check installation

```
cd /cluster/work/climate/$USER
python -c 'import climada; print(climada.__file__)'
```

should output exactly this (with explicit $USER):

```
/cluster/project/climate/$USER/climada_python/climada/__init__.py
```

4.11 6. Adjust the Climada configuration

See *Adjust the Climada configuration* above.

4.12 7. Run a job

See *Run a job* above.

4.12.1 Fallback: Conda installation

If Climada cannot be installed through pip because of changed dependency requirements, there is still the possibility to install Climada through the Conda environment. > **WARNING:** This apporach is highly discouraged, as it imposes a heavy and mostly unnecessary burdon on the file system of the cluster.

4.13 1. Conda Installation

Download or update to the latest version of Miniconda. Installation is done by execution of the following steps:

```
cd /cluster/project/climate/USERNAME
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
miniconda3/bin/conda init
rm Miniconda3-latest-Linux-x86_64.sh
```

During the installation process of Miniconda, you are prompted to set the working directory according to your choice. Set it to `/cluster/project/climate/USERNAME/miniconda3`. Once the installation has finished, log out of Euler and in again. The command prompt should be preceded by `(base)`, indicating that the installation was a success and that you login in into conda's base environment by default.

4.14 2. Checkout sources

See *Checkout sources* above.

4.15 3. Climada Environment

Create the conda environment:

```
cd /cluster/project/climate/USERNAME/climada_python
conda env create -f requirements/env_climada.yml --name climada_env
conda env update -n climada_env -f requirements/env_developer.yml

conda activate climada_env
conda install conda-build
conda develop .
```

4. Adjust the Climada configuration

See *Adjust the Climada configuration* above.

4.16 5. Climada Scripts

Create a bash script for executing python scripts in the climada environment, `climadajob.sh`:

```
#!/bin/bash
PYTHON_SCRIPT=$1
shift
. ~/.bashrc
conda activate climada_env
python $PYTHON_SCRIPT $@
echo $PYTHON_SCRIPT completed
```

Make it executable:

```
chmod +x climadajob.sh
```

Create a python script that executes climada code, e.g., `climada_smoke_test.py`:

```
import sys
from climada import CONFIG, SYSTEM_DIR
from climada.util.test.test_finance import TestNetpresValue
TestNetpresValue().test_net_pres_val_pass()
print(SYSTEM_DIR)
print(CONFIG.local_data.save_dir.str())
print("the script ran with arguments", sys.argv)
```

4.17 6. Run a Job

Please see the Wiki: https://scicomp.ethz.ch/wiki/Using_the_batch_system.

With the scripts from above you can submit the python script as a job like this:

```
bsub [options] /path/to/climadajob.sh /path/to/climada_smoke_test.py arg1 arg2
```

After the job has finished the lsf output file should look something like this:

```
Sender: LSF System <lsfadmin@eu-ms-010-32>
Subject: Job 161617875: <./climada_job.sh climada_smoke_test.py arg1 arg2> in cluster
↪<euler> Done

Job <./climada_job.sh climada_smoke_test.py arg1 arg2> was submitted from host <eu-login-
↪41> by user <USERNAME> in cluster <euler> at Thu Jan 28 14:10:15 2021
Job was executed on host(s) <eu-ms-010-32>, in queue <normal.4h>, as user <USERNAME> in␣
↪cluster <euler> at Thu Jan 28 14:10:42 2021
</cluster/home/USERNAME> was used as the home directory.
</cluster/work/climate/USERNAME> was used as the working directory.
Started at Thu Jan 28 14:10:42 2021
Terminated at Thu Jan 28 14:10:53 2021
Results reported at Thu Jan 28 14:10:53 2021

Your job looked like:

------------------------------------------------------------
# LSBATCH: User input
./climada_job.sh climada_smoke_test.py arg1 arg2
------------------------------------------------------------

Successfully completed.

Resource usage summary:

    CPU time :                                   2.99 sec.
    Max Memory :                                 367 MB
    Average Memory :                             5.00 MB
    Total Requested Memory :                     1024.00 MB
    Delta Memory :                               657.00 MB
    Max Swap :                                   -
    Max Processes :                              5
    Max Threads :                                6
    Run time :                                   22 sec.
    Turnaround time :                            38 sec.

The output (if any) follows:

/cluster/project/climate/USERNAME/miniconda3/envs/climada/lib/python3.7/site-packages/
↪pandas_datareader/compat/__init__.py:7: FutureWarning: pandas.util.testing is␣
↪deprecated. Use the functions in the public API at pandas.testing instead.
  from pandas.util.testing import assert_frame_equal
/cluster/work/climate/USERNAME/climada/data
```

(continues on next page)

(continued from previous page)

```
/cluster/work/climate/USERNAME/climada/results
the script ran with arguments ['/path/to/climada_smoke_test.py', 'arg1' 'arg2']
python_script.sh completed
```

Conda Deinstallation

4.18 1. Conda

Remove the miniconda3 directory from the installation directory:

```
rm -rf /cluster/project/climate/USERNAME/miniconda3/
```

Delete the conda related parts from `/cluster/home/USERNAME/.bashrc`, i.e., everything between

`# >>> conda initialize >>>`
and
`# <<< conda initialize <<<`

4.19 2. Climada

Remove the climada sources and config file:

```
rm -rf /cluster/project/climate/USERNAME/climada_python
rm -f /cluster/home/USERNAME/climada.conf /cluster/home/USERNAME/*/climada.conf
```

CHAPTER

FIVE

TUTORIALS

5.1 CLIMADA overview

5.1.1 Contents

5.1.2 Introduction

What is CLIMADA?

CLIMADA is a fully probabilistic climate risk assessment tool. It provides a framework for users to combine exposure, hazard and vulnerability or impact data to calculate risk. Users can create probabilistic impact data from event sets, look at how climate change affects these impacts, and see how effectively adaptation measures can change them. CLIMADA also allows for studies of individual events, historical event sets and forecasts.

The model is a highly customisable, meaning that users can work with out-of-the-box data provided for different hazards, population and economic exposure, or can provide their own data for part or all of the analysis. The pre-packaged data make CLIMADA particularly useful for users who focus on just one element of risk, since CLIMADA can 'fill in the gaps' for hazard, exposure or vulnerability in the rest of the analysis.

The model core is designed to give as much flexibility as possible when describing the elements of risk, meaning that CLIMADA isn't limited to particular hazards, exposure types or impacts. We love to see the model applied to new problems and contexts.

CLIMADA provides classes, methods and data for exposure, hazard and impact functions (also called vulnerability functions), plus a financial model and a framework to analyse adaptation measures. Additional classes and data for common uses, such as economic exposures or tropical storms. Tutorials for every class are available: see the *CLIMADA features* section below.

This tutorial

This tutorial is for people new to CLIMADA who want to get a high level understanding of the model and work through an example risk analysis. It will list the current features of the model, and go through a complete CLIMADA analysis to give an idea of how the model works. Other tutorials go into more detail about different model components and individual hazards.

Resources beyond this tutorial

- Installation guide - go here if you've not installed the model yet
- CLIMADA Read the Docs home page - for all other documentation
- *List of CLIMADA's features and associated tutorials*
- CLIMADA GitHub develop branch documentation for the very latest versions of code and documentation
- CLIMADA paper GitHub repository - for publications using CLIMADA

5.1.3 CLIMADA features

A risk analysis with CLIMADA can include

1. the statistical risk to your exposure from a set of events,
2. how it changes under climate change, and
3. a cost-benefit analysis of adaptation measures.

CLIMADA is flexible: the "statistical risk" above could be describing the annual expected insured flood losses to a property portfolio, the number of people displaced by an ensemble of typhoon forecasts, the annual disruption to a railway network from landslides, or changes to crop yields.

Users from risk-analysis backgrounds will be familiar with describing the impact of events by combining exposure, hazard and an impact function (or vulnerability curve) that combines the two to describe a hazard's effects. A CLIMADA analysis uses the same approach but wraps the exposures and their impact functions into a single `Entity` class, along with discount rates and adaptation options (see the below tutorials for more on CLIMADA's financial model).

CLIMADA's `Impact` object is used to analyse events and event sets, whether this is the impact of a single wildfire, or the global economic risk from tropical cyclones in 2100.

CLIMADA classes

This is a full directory of tutorials for CLIMADA's classes to use as a reference. You don't need to read all this to do this tutorial, but it may be useful to refer back to.

- Hazard: a class that stores sets of geographic hazard footprints, (e.g. for wind speed, water depth and fraction, drought index), and metadata including event frequency. Several predefined extensions to create particular hazards from particular datasets and models are included with CLIMADA:
  - Tropical cyclone wind: global hazard sets for tropical cyclone events, constructing statistical wind fields from storm tracks. Subclasses include methods and data to calculate historical wind footprints, create forecast enembles from ECMWF tracks, and create climatological event sets for different climate scenarios.
  - Storm surge: (under development) global surge hazard for tropical storms. Runs the GeoClaw surge model to create and plot hazard from tropical storm tracks
  - European windstorms: includes methods to read and plot footprints from the Copernicus WISC dataset and for DWD and ICON forecasts.
  - River flooding: global water depth hazard for flood, including methods to work with ISIMIP simulations.
  - Crop modelling: combines ISIMIP crop simulations and UN Food and Agrigultre Organization data. The module uses crop production as exposure, with hydrometeorological 'hazard' increasing or decreasing production.
  - Wildfire (global): tutorial under development
  - Drought (global): tutorial under development
- *Entity*: this is a container that groups CLIMADA's socio-economic models. It's is where the Exposures and Impact Functions are stored, which can then be combined with a hazard for a risk analysis (using the Engine's Impact class). It is also where Discount Rates and Measure Sets are stored, which are used in adaptation cost-benefit analyses (using the Engine's CostBenefit class):
  - Exposures: geolocated exposures. Each exposure is associated with a value (which can be a dollar value, population, crop yield, etc), information to associate it with impact functions for the relevant hazard(s) (in the Entity's ImpactFuncSet), a geometry, and other optional properties such as deductables and cover. Exposures can be loaded from a file, specified by the user, or created from regional economic models accessible within CLIMADA, for example:
    - LitPop: regional economic model using nightlight and population maps together with several economic indicators
    - BlackMarble: regional economic model from nightlight intensities and economic indicators (GDP, income group). Largely succeeded by LitPop.
    - OpenStreetMap: methods to create exposures from data available through the OpenStreetMap API
  - ImpactFuncSet: functions to describe the impacts that hazards have on exposures, expressed in terms of e.g. the % dollar value of a building lost as a function of water depth, or the mortality rate for over-70s as a function of temperature. CLIMADA provides some common impact functions, or they can be user-specified. The following is an incomplete list:

        * ImpactFunc: a basic adjustable impact function, specified by the user
        * IFTropCyclone: impact functions for tropical cyclone winds
        * IFRiverFlood: impact functions for river floods
        * IFStormEurope: impact functions for European windstorms
    - DiscRates: discount rates per year
    - MeasureSet: a collection of Measure objects that together describe any adaptation measures being modelled. Adaptation measures are described by their cost, and how they modify exposure, hazard, and impact functions (and have have a method to do these things). Measures also include risk transfer options.
- Engine: the CLIMADA Engine contains the Impact and CostBenefit classes, which are where the main model calculations are done, combining Hazard and Entity objects.
    - Impact: a class that stores CLIMADA's modelled impacts and the methods to calculate them from Exposure, Impact Function and Hazard classes. The calculations include average annual impact, expected annual impact by exposure item, total impact by event, and (optionally) the impact of each event on each exposure point. Includes statistical and plotting routines for common analysis products.
    - CostBenefit: a class to appraise adaptation options. It uses an Entity's MeasureSet to calculate new Impacts based on their adjustments to hazard, exposure, and impact functions, and returns statistics and plotting routines to express cost-benefit comparisons.

This list will be updated periodically along with new CLIMADA releases. To see the latest, development version of all tutorials, see the tutorials page on the CLIMADA GitHub.

5.1.4 Tutorial: an example risk assessment

This example will work through a risk assessment for tropical storm wind in Puerto Rico, constructing hazard, exposure and vulnerability and combining them to create an Impact object. Everything you need for this is included in the main CLIMADA installation and additional data will be downloaded by the scripts as required.

5.1.5 Hazard

Hazards are characterized by their frequency of occurrence and the geographical distribution of their intensity. The `Hazard` class collects events of the same hazard type (e.g. tropical cyclone, flood, drought, ...) with intensity values over the same geographic centroids. They might be historical events or synthetic.

See the *Hazard tutorial* to learn about the Hazard class in more detail, and the *CLIMADA features* section of this document to explore tutorials for different hazards, including *tropical cyclones*, as used here.

Tropical cyclones in CLIMADA and the `TropCyclone` class work like any hazard, storing each event's wind speeds at the geographic centroids specified for the class. Pre-calculated hazards can be loaded from files (see the *full Hazard tutorial*, but they can also be modelled from a storm track using the `TCTracks` class, based on a storm's parameters at each time step. This is how we'll construct the hazards for our example.

So before we create the hazard, we will create our storm tracks and define the geographic centroids for the locations we want to calculate hazard at.

Storm tracks

Storm tracks are created and stored in a separate class, `TCTracks`. We use its method `from_ibtracs_netcdf` to create the tracks from the IBTRaCS storm tracks archive. The first time this runs on your machine it will need to download the full dataset which might take a little time. See the *full TropCyclone tutorial* for more detail and troubleshooting.

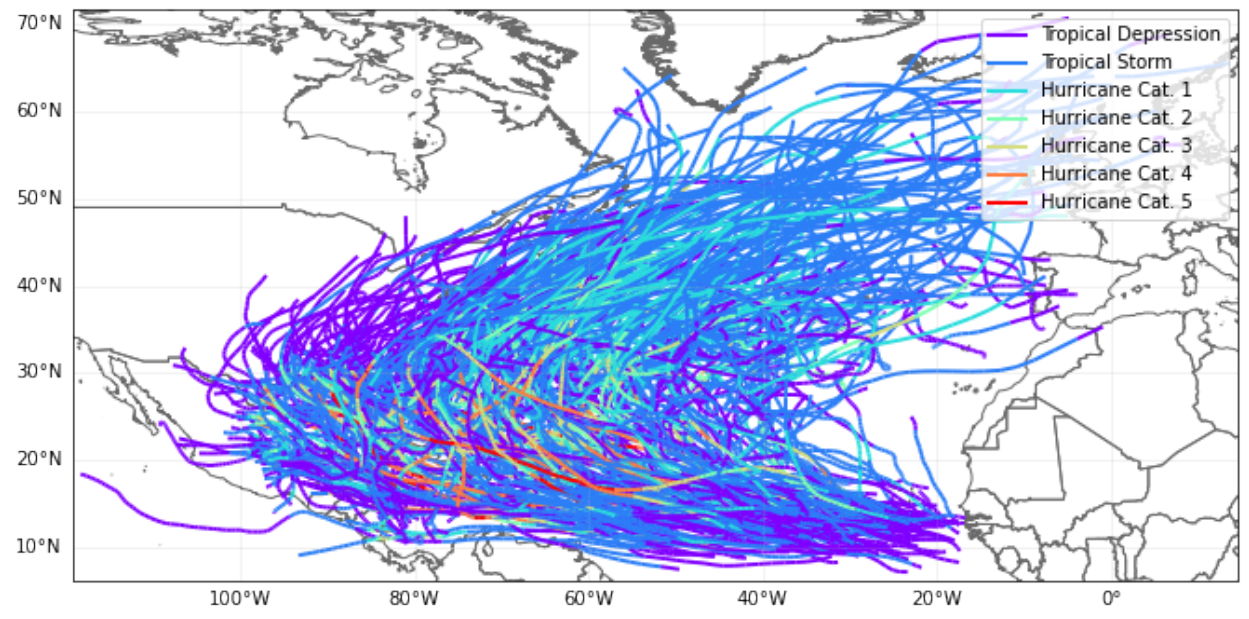
[1]:
```
import numpy as np
from climada.hazard import TCTracks

tracks = TCTracks.from_ibtracs_netcdf(provider='usa', basin='NA')
```

```
2021-10-19 16:41:37,771 - climada.hazard.tc_tracks - WARNING - 1125 storm events are␣
↪discarded because no valid wind/pressure values have been found: 1851175N26270,␣
↪1851181N19275, 1851187N22262, 1851192N12300, 1851214N14321, ...
2021-10-19 16:41:37,783 - climada.hazard.tc_tracks - WARNING - 127 storm events are␣
↪discarded because only one valid timestep has been found: 1852232N21293, 1853242N12336,
↪ 1855236N12304, 1856221N25277, 1856235N13302, ...
```

This will load all historical tracks in the North Atlantic into the `tracks` object (since we set `basin='NA'`). The `TCTracks.plot` method will plot the downloaded tracks, though there are too many for the plot to be very useful:

[2]:
```
tracks.plot()
```

[2]:
```
<GeoAxesSubplot:>
```



It's also worth adding additional time steps to the tracks (though this can be memory intensive!). Most tracks are reported at 3-hourly intervals (plus a frame at landfall). Event footprints are calculated as the maximum wind from any time step. For a fast-moving storm these combined three-hourly footprints give quite a rough event footprint, and it's worth adding extra frames to smooth the footprint artificially (try running this notebook with and without this interpolation to see the effect):

[3]:
```
tracks.equal_timestep(time_step_h=0.5)
```

Now, irresponsibly for a risk analysis, we're only going to use these historical events: they're enough to demonstrate CLIMADA in action. A proper risk analysis would expand it to include enough events for a statistically robust climatology. See the *full TropCyclone tutorial* for CLIMADA's stochastic event generation.

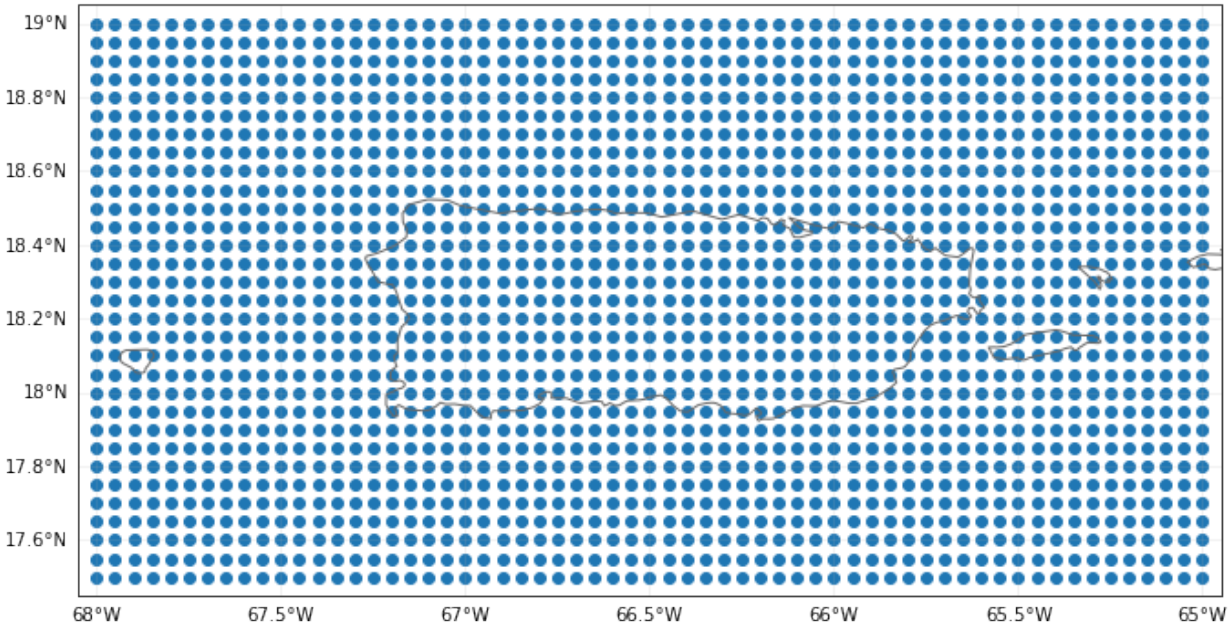
Centroids

A hazard's centroids can be any set of locations where we want the hazard to be evaluated. This could be the same as the locations of your exposure, though commonly it is on a regular lat-lon grid (with hazard being imputed to exposure between grid points).

Here we'll set the centroids as a 0.1 degree grid covering Puerto Rico. Centroids are defined by a `Centroids` class, which has the `from_pnt_bounds` method for generating regular grids and a `plot` method to inspect the centroids.

[4]:
```
from climada.hazard import Centroids

min_lat, max_lat, min_lon, max_lon = 17.5, 19.0, -68.0, -65.0
cent = Centroids.from_pnt_bounds((min_lon, min_lat, max_lon, max_lat), res=0.05)
cent.check()
cent.plot()
```

[4]:
```
<GeoAxesSubplot:>
```



Almost every class in CLIMADA has a `check()` method, as used above. This verifies that the necessary data for an objet is correctly provided and logs the optional variables that are not present. It is always worth running it after filling an instance of an object.

Hazard footprint

Now we're ready to create our hazard object. This will be a `TropCyclone` class, which inherits from the `Hazard` class, and has the `from_tracks` constructor method to create a hazard from a `TCTracks` object at given centroids.

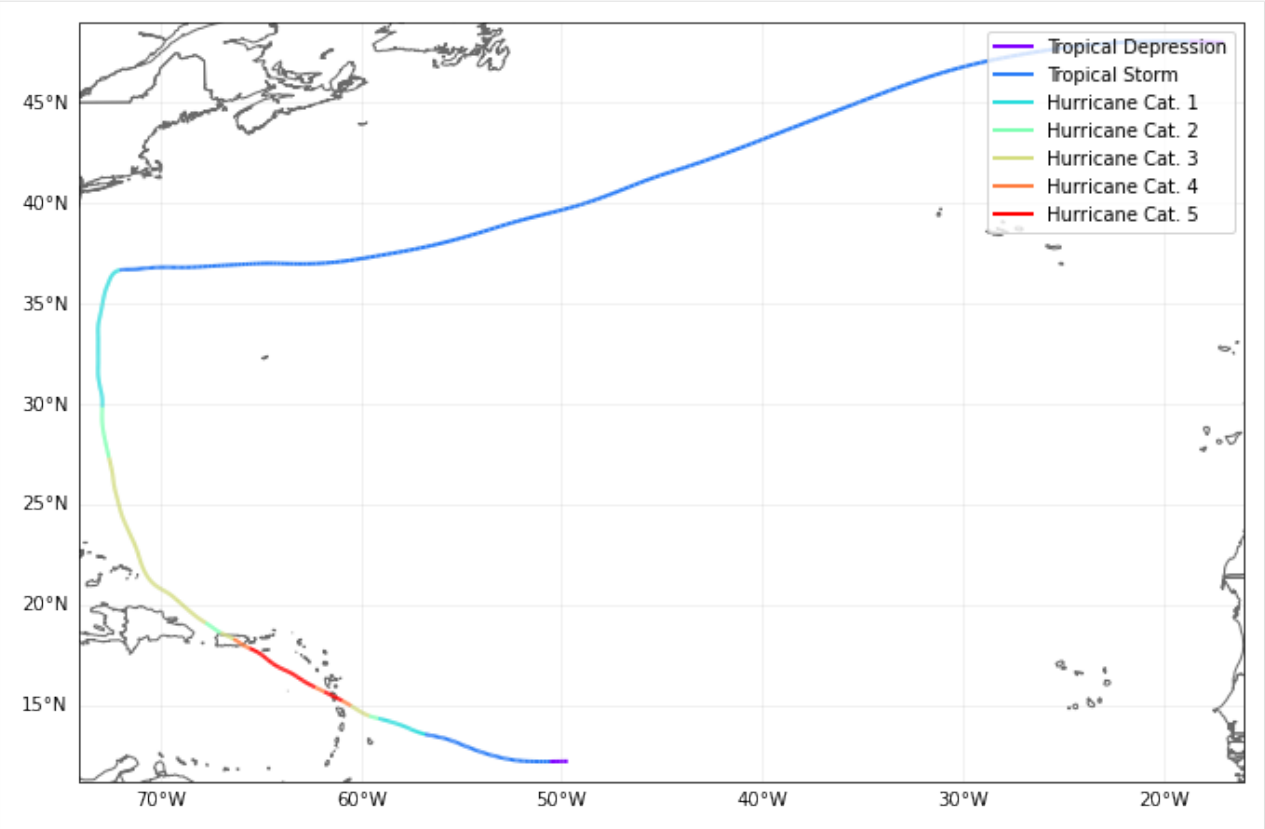
[5]:
```
from climada.hazard import TropCyclone

haz = TropCyclone.from_tracks(tracks, centroids=cent)
haz.check()
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

In 2017 Hurricane Maria devastated Puerto Rico. In the IBTRaCs event set, it has ID `2017260N12310` (we use this rather than the name, as IBTRaCS contains three North Atlantic storms called Maria). We can plot the track:

[6]:
```
tracks.subset({"sid": "2017260N12310"}).plot()  # This is how we subset a TCTracks object
```

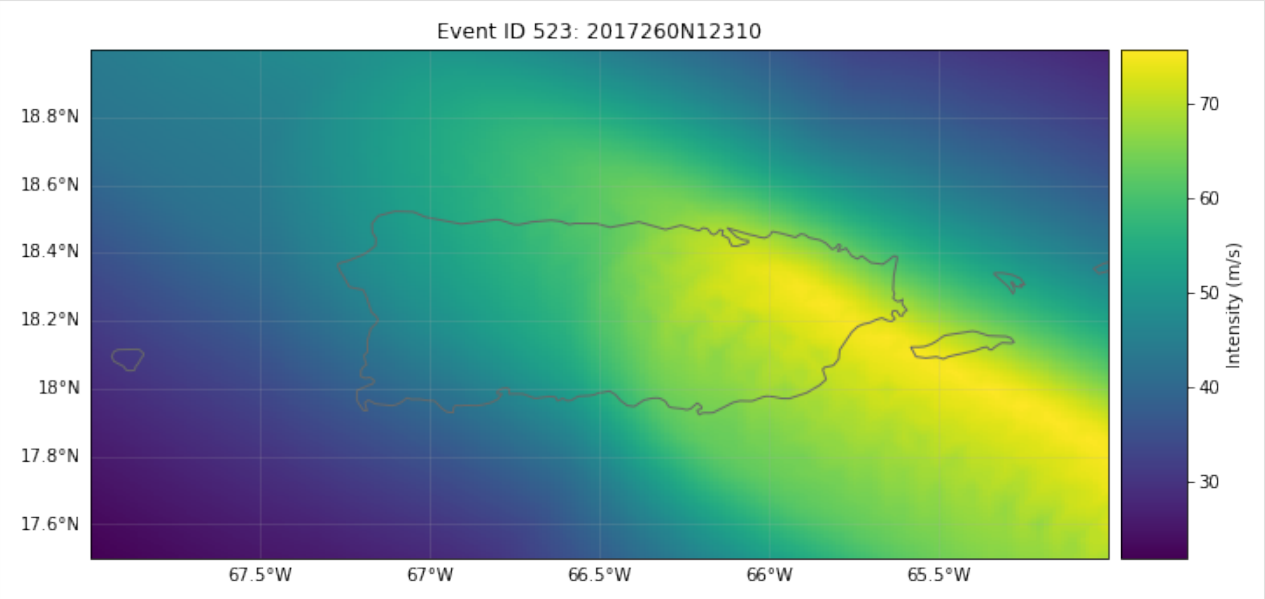
[6]:
```
<GeoAxesSubplot:>
```



And plot the hazard on our centroids for Puerto Rico:

[7]:
```
haz.plot_intensity(event='2017260N12310')
```

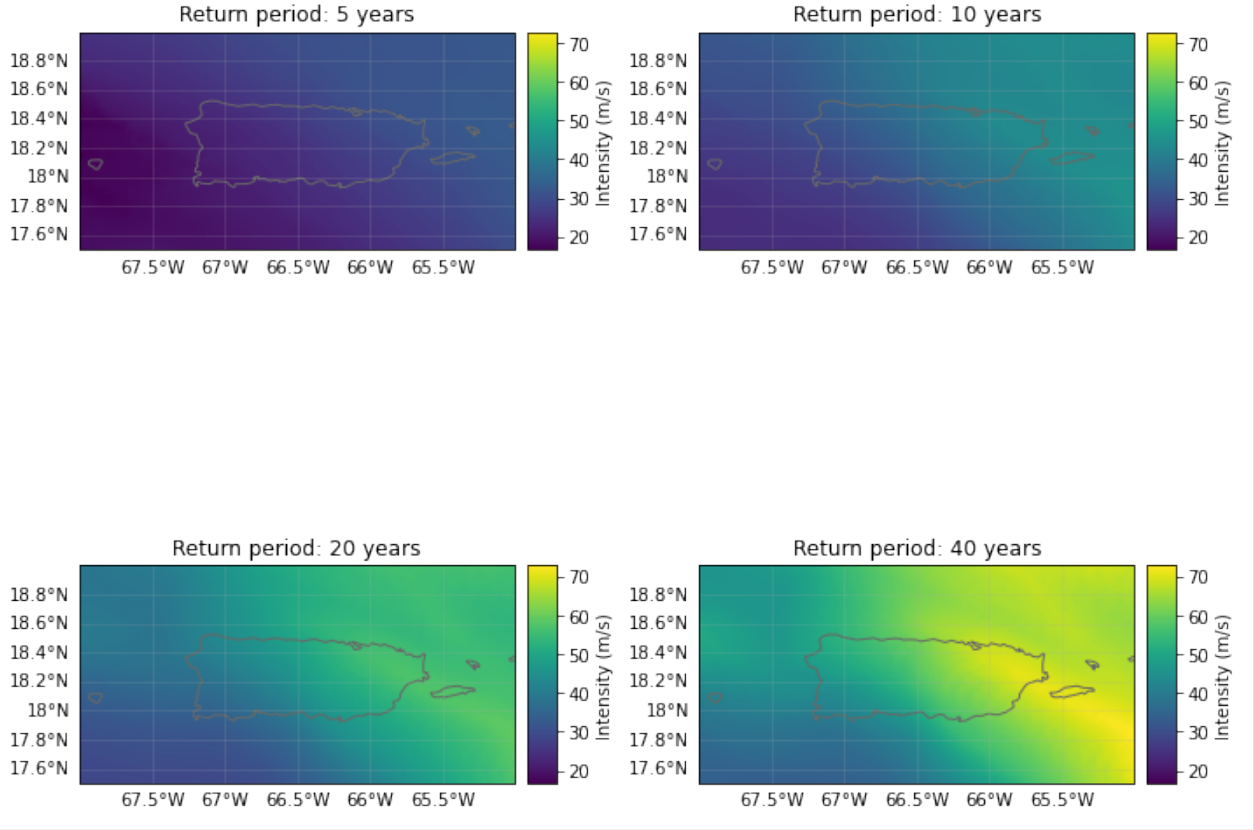
[7]:
```
<GeoAxesSubplot:title={'center':'Event ID 948: 2017260N12310'}>
```



A Hazard object also lets us plot the hazard at different return periods. The IBTRaCS archive produces footprints from 1980 onwards (CLIMADA discarded earlier events) and so the historical period is short. Therefore these plots don't make sense as 'real' return periods, but we're being irresponsible and demonstrating the functionality anyway.

[8]:
```
haz.plot_rp_intensity(return_periods=(5,10,20,40))
```

[8]:
```
(array([[<GeoAxesSubplot:title={'center':'Return period: 5 years'}>,
         <GeoAxesSubplot:title={'center':'Return period: 10 years'}>],
        [<GeoAxesSubplot:title={'center':'Return period: 20 years'}>,
         <GeoAxesSubplot:title={'center':'Return period: 40 years'}>]],
       dtype=object),
 array([[15.08108124, 15.11391293, 15.04906939, ..., 16.39247373,
         16.53654363, 16.73792023],
        [23.28332256, 23.25727297, 23.23369223, ..., 27.39433861,
         27.73305539, 27.91210025],
        [31.48556389, 31.40063301, 31.41831506, ..., 38.3962035 ,
         38.92956714, 39.08628027],
        [39.68780521, 39.54399305, 39.60293789, ..., 49.39806839,
         50.1260789 , 50.26046029]]))
```



See the *TropCyclone tutorial* for full details of the TropCyclone hazard class.

We can also recalculate event sets to reflect the effects of climate change. The `apply_climate_scenario_knu` method applies changes in intensity and frequency projected due to climate change, as described in 'Global projections of intense tropical cyclone activity for the late twenty-first century from dynamical downscaling of CMIP5/RCP4.5 scenarios' (Knutson *et al.* 2015). See the *tutorial* for details.

> **Exercise:** Extend this notebook's analysis to examine the effects of climate change in Puerto Rico. You'll need to extend the historical event set with stochastic tracks to create a robust statistical storm climatology - the `TCTracks` class has the functionality to do this. Then you can apply the `apply_climate_scenario_knu` method to the generated hazard object to create a second hazard climatology representing storm activity under climate change. See how the results change using the different

hazard sets.

Next we'll work on exposure and vulnerability, part of the Entity class.

5.1.6 Entity

The entity class is a container class that stores exposures and impact functions (vulnerability curves) needed for a risk calculation, and the discount rates and adaptation measures for an adaptation cost-benefit analysis.

As with Hazard objects, Entities can be read from files or created through code. The Excel template can be found in `climada_python/data/system/entity_template.xlsx`.

In this tutorial we will create an Exposure object using the LitPop economic exposure module, and load a pre-defined wind damage function.

First we create an empty Entity object:

[9]:
```
from climada.entity import Entity

ent = Entity()
```

Exposures

The `Entity`'s `exposures` attribute contains geolocalized values of anything exposed to the hazard, whether monetary values of assets or number of human lifes, for example. It is of type `Exposures`.

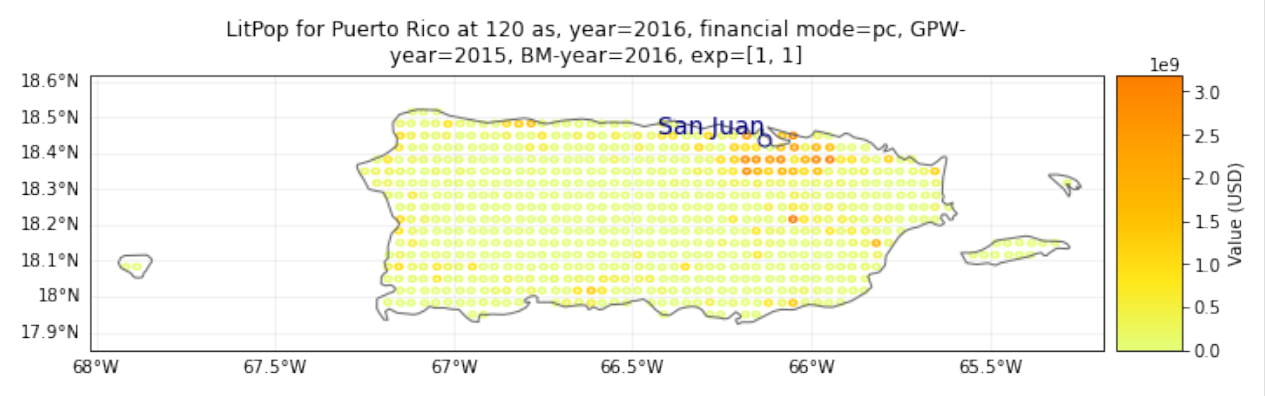
See the *Exposures tutorial* for more detail on the structure of the class, and how to create and import exposures. The *LitPop tutorial* explains how CLIMADA models economic exposures using night-time light and economic data, and is what we'll use here. To combine your exposure with OpenStreetMap's data see the OSM tutorial.

LitPop is a module that allows CLIMADA to estimate exposed populations and economic assets at any point on the planet without additional information, and in a globally consistent way. Here we can create an economic Exposure dataset for Puerto Rico, add it to our Entity, and plot it:

[10]:
```
from climada.entity.exposures import LitPop

exp_litpop = LitPop.from_countries('Puerto Rico', res_arcsec = 120)  # We'll go lower␣
↪resolution than default to keep it simple
exp_litpop.set_geometry_points() # Set geodataframe geometries from lat lon data

ent.exposures = exp_litpop

exp_litpop.plot_hexbin(pop_name=True, linewidth=4, buffer=0.1)
```

```
2021-10-19 16:43:39,830 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2021-10-19 16:43:40,203 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2021-10-19 16:43:40,850 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
```

[10]:
```
<GeoAxesSubplot:title={'center':"LitPop Exposure for ['Puerto Rico'] at 120 as, year:␣
↪2018, financial\nmode: pc, exp: (1, 1), admin1_calc: False"}>
```



LitPop's default exposure is measured in US Dollars, with a reference year depending on the most recent data available.

Once we've created our impact function we will come back to this Exposure and give it the parameters needed to connect exposure to impacts.

Impact functions

Impact functions describe a relationship between a hazard's intensity and your exposure in terms of a percentage loss. The impact is described through two terms. The Mean Degree of Damage (MDD) gives the percentage of an exposed asset's numerical value that's affected as a function of intensity, such as the damage to a building from wind in terms of its total worth. Then the Proportion of Assets Affected (PAA) gives the fraction of exposures that are affected, such as the mortality rate in a population from a heatwave. These multiply to give the Mean Damage Ratio (MDR), the average impact to an asset.

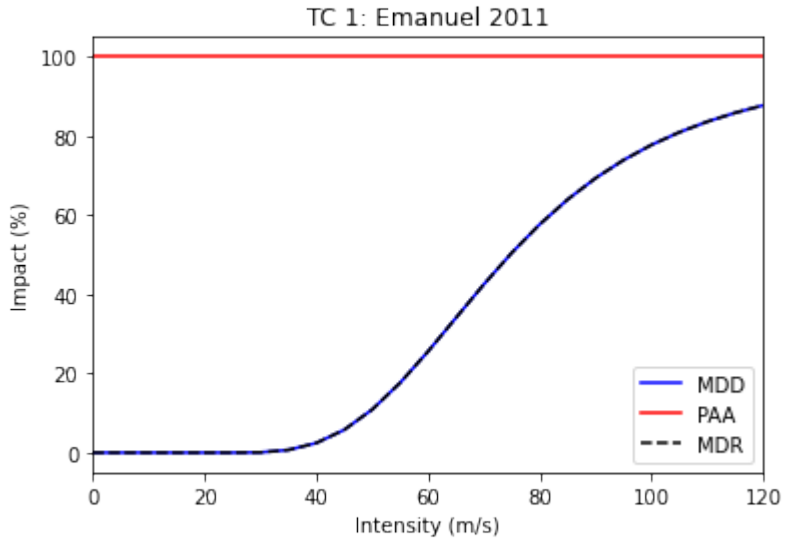
Impact functions are stored as the Entity's `impact_funcs` attribute, in an instance of the `ImpactFuncSet` class which groups one or more `ImpactFunc` objects. They can be specified manually, read from a file, or you can use CLIMADA's pre-defined impact functions. We'll use a pre-defined function for tropical storm wind damage stored in the `IFTropCyclone` class.

See the *Impact Functions tutorial* for a full guide to the class, including how data are stored and reading and writing to files.

We initialise an Impact Function with the `IFTropCyclone` class, and use its `from_emanuel_usa` method to load the Emanuel (2011) impact function. (The class also contains regional impact functions for the full globe, but we'll won't use these for now.) The class's `plot` method visualises the function, which we can see is expressed just through the Mean Degree of Damage, with all assets affected.

[11]:
```
from climada.entity.impact_funcs import ImpactFuncSet, ImpfTropCyclone

imp_fun = ImpfTropCyclone.from_emanuel_usa()
imp_fun.plot()
```

[11]:
```
<AxesSubplot:title={'center':'TC 1: Emanuel 2011'}, xlabel='Intensity (m/s)', ylabel=
↪'Impact (%)'>
```



The plot title also includes information about the function's ID, which were also set by the `from_emanuel_usa` class method. The hazard is "TC" and the function ID is 1. Since a study might use several impact functions - for different hazards, or for different types of exposure.

We then create an `ImpactFuncSet` object to store the impact function. This is a container class, and groups a study's impact functions together. Studies will often have several impact functions, due to multiple hazards, multiple types of exposure that are impacted differently, or different adaptation scenarios. We add it to our Entity object.

[14]:
```
imp_fun_set = ImpactFuncSet()
imp_fun_set.append(imp_fun)

ent.impact_funcs = imp_fun_set
```

Finally, we can update our LitPop exposure to point to the TC 1 impact function. This is done by adding a column to the exposure:

[13]:
```
ent.exposures.gdf['impf_TC'] = 1
ent.check()
```

```
2021-10-19 16:43:43,386 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
```

Here the `impf_TC` column tells the CLIMADA engine that for a tropical cyclone (TC) hazard, it should use the first impact function defined for TCs. We use the same impact function for all of our exposure.

This is now everything we need for a risk analysis, but while we're working on the Entity class, we can define the adaptation measures and discount rates needed for an adaptation analysis. If you're not interested in the cost-benefit analysis, you can skip ahead to the *Impact section*

Adaptation measures

CLIMADA's adaptation measures describe possible interventions that would change event hazards and impacts, and the cost of these interventions.

They are stored as `Measure` objects within a `MeasureSet` container class (similarly to `ImpactFuncSet` containing several `ImpactFuncs`), and are assigned to the `measures` attribute of the Entity.

See the *Adaptation Measures tutorial* on how to create, read and write measures. CLIMADA doesn't yet have pre-defined adaptation measures, mostly because they are hard to standardise.

The best way to understand an adaptation measure is by an example. Here's a possible measure for the creation of coastal mangroves (ignore the exact numbers, they are just for illustration):

[16]:
```
from climada.entity import Measure, MeasureSet

meas_mangrove = Measure()
meas_mangrove.name = 'Mangrove'
meas_mangrove.haz_type = 'TC'
meas_mangrove.color_rgb = np.array([0.2, 0.2, 0.7])
meas_mangrove.cost = 500000000
meas_mangrove.mdd_impact = (1, 0)
meas_mangrove.paa_impact = (1, -0.15)
meas_mangrove.hazard_inten_imp = (1, -10)

meas_set = MeasureSet()
meas_set.append(meas_mangrove)
meas_set.check()
```

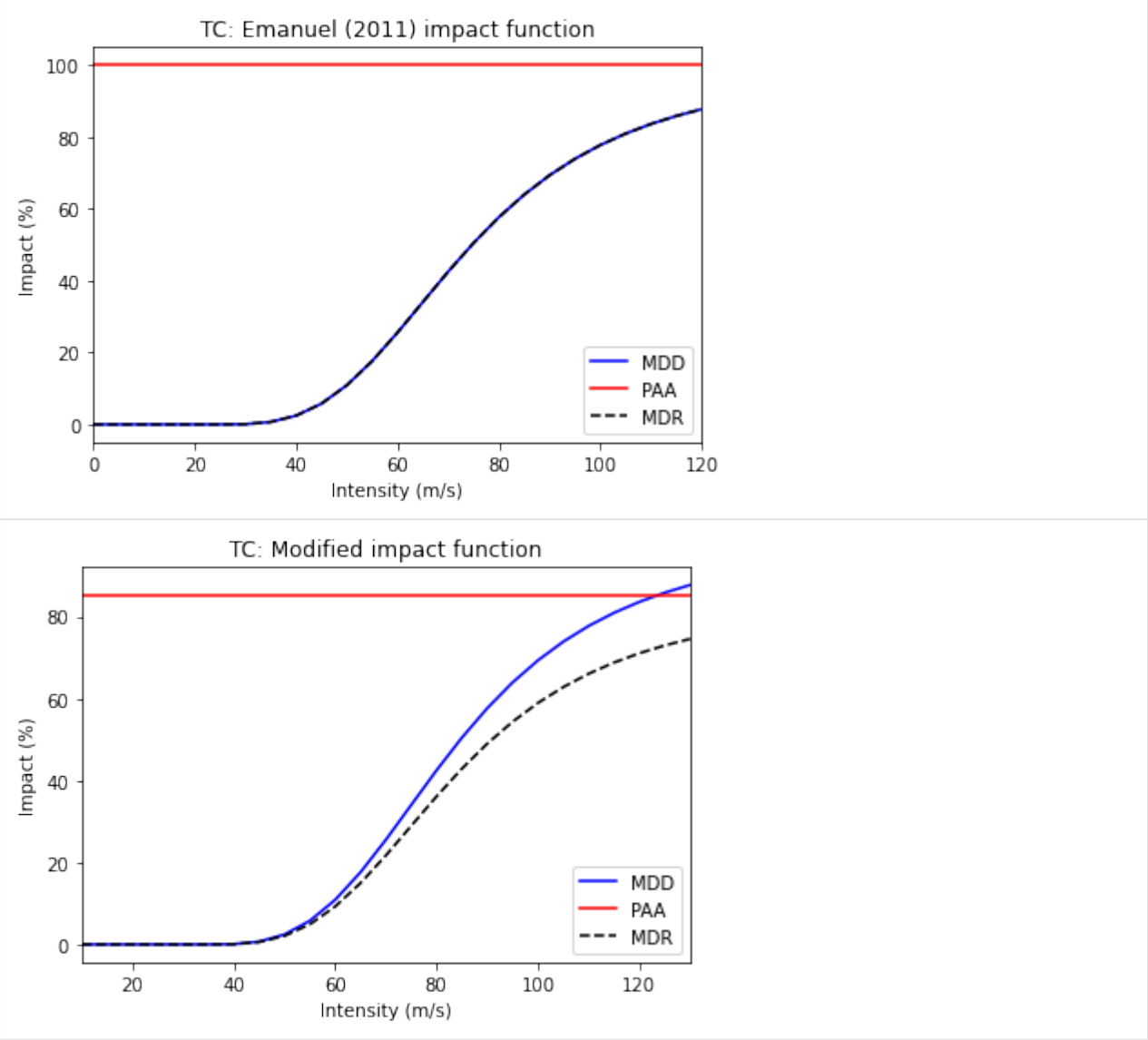
What values have we set here? - The `haz_type` gives the hazard that this measure affects. - The `cost` is a flat price that will be used in cost-benefit analyses. - The `mdd_impact`, `paa_impact`, and `hazard_inten_imp` attributes are all tuples that describes a linear transformation to event hazard, the impact function's mean damage degree and the impact function's proportion of assets affected. The tuple `(a, b)` describes a scalar multiplication of the function and a constant to add. So `(1, 0)` is unchanged, `(1.1, 0)` increases values by 10%, and `(1, -10)` decreases all values by 10.

So the Mangrove example above costs 50,000,000 USD, protects 15% of assets from any impact at all (`paa_impact = (1, -0.15)`) and decreases the (effective) hazard intensity by 10 m/s (`hazard_inten_imp = (1, -10)`).

We can apply these measures to our existing Exposure, Hazard and Impact functions, and plot the old and new impact functions:

[17]:
```
mangrove_exp, mangrove_imp_fun_set, mangrove_haz = meas_mangrove.apply(exp_litpop, imp_
↪fun_set, haz)
axes1 = imp_fun_set.plot()
axes1.set_title('TC: Emanuel (2011) impact function')
axes2 = mangrove_imp_fun_set.plot()
axes2.set_title('TC: Modified impact function')
```

[17]:
```
Text(0.5, 1.0, 'TC: Modified impact function')
```



Let's define a second measure. Again, the numbers here are made up, for illustration only.

[16]:
```
meas_buildings = Measure()
meas_buildings.name = 'Building code'
meas_buildings.haz_type = 'TC'
meas_buildings.color_rgb = np.array([0.2, 0.7, 0.5])
meas_buildings.cost = 100000000
meas_buildings.hazard_freq_cutoff = 0.1

meas_set.append(meas_buildings)
meas_set.check()

buildings_exp, buildings_imp_fun_set, buildings_haz = meas_buildings.apply(exp_litpop,␣
↪imp_fun_set, haz)
```

This measure describes an upgrade to building codes to withstand 10-year events. The measure costs 100,000,000 USD and, through `hazard_freq_cutoff = 0.1`, removes events with calculated impacts below the 10-year return period.

The *Adaptation Measures tutorial* descibes other parameters for describing adaptation measures, including risk transfer, assigning measures to subsets of exposure, and reassigning impact functions.

We can compare the 5- and 20-year return period hazard (remember: not a real return period due to the small event set!) compared to the adjusted hazard once low-impact events are removed.

[ ]:
```
haz.plot_rp_intensity(return_periods=(5, 20))
buildings_haz.plot_rp_intensity(return_periods=(5, 20))
```

It shows there are now very few events at the 5-year return period - the new building codes removed most of these from the event set. Finally we add the measure set to our Entity.

[18]:
```
ent.measures = meas_set
```

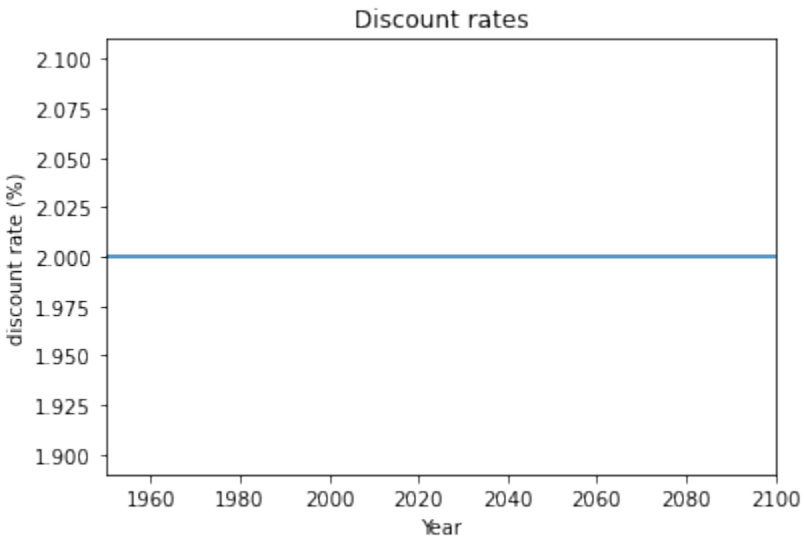
Discount rates

The `disc_rates` attribute is of type `DiscRates`. This class contains the discount rates for the following years and computes the net present value for given values.

See the *Discount Rates tutorial* for more details about creating, reading and writing the `DiscRates` class, and how it is used in calculations.

Here we will implement a simple, flat 2% discount rate.

[19]:
```
from climada.entity import DiscRates

disc = DiscRates()
disc.years = np.arange(1950, 2101)
disc.rates = np.ones(disc.years.size) * 0.02
disc.check()
disc.plot()

ent.disc_rates = disc
```



We are now ready to move to the last part of the CLIMADA model for Impact and Cost Benefit analyses.

5.1.7 Engine

The CLIMADA Engine is where the main risk calculations are done. It contains two classes, `Impact`, for risk assessments, and `CostBenefit`, to evaluate adaptation measures.

Impact

Let us compute the impact of historical tropical cyclones in Puerto Rico.

Our work above has given us everything we need for a risk analysis using the Impact class. By computing the impact for each historical event, the `Impact` class provides different risk measures, as the expected annual impact per exposure, the probable maximum impact for different return periods and the total average annual impact.

Note: the configurable parameter `MAX_SIZE` controls the maximum matrix size contained in a chunk. You can decrease its value if you are having memory issues when using the `Impact`'s `calc` method. A high value will make the computation fast, but increase the memory use. The configuration file is located at `climada_python/climada/conf/defaults.conf`.

CLIMADA calculates impacts by providing exposures, impact functions and hazard to an `Impact` object's `calc` method:

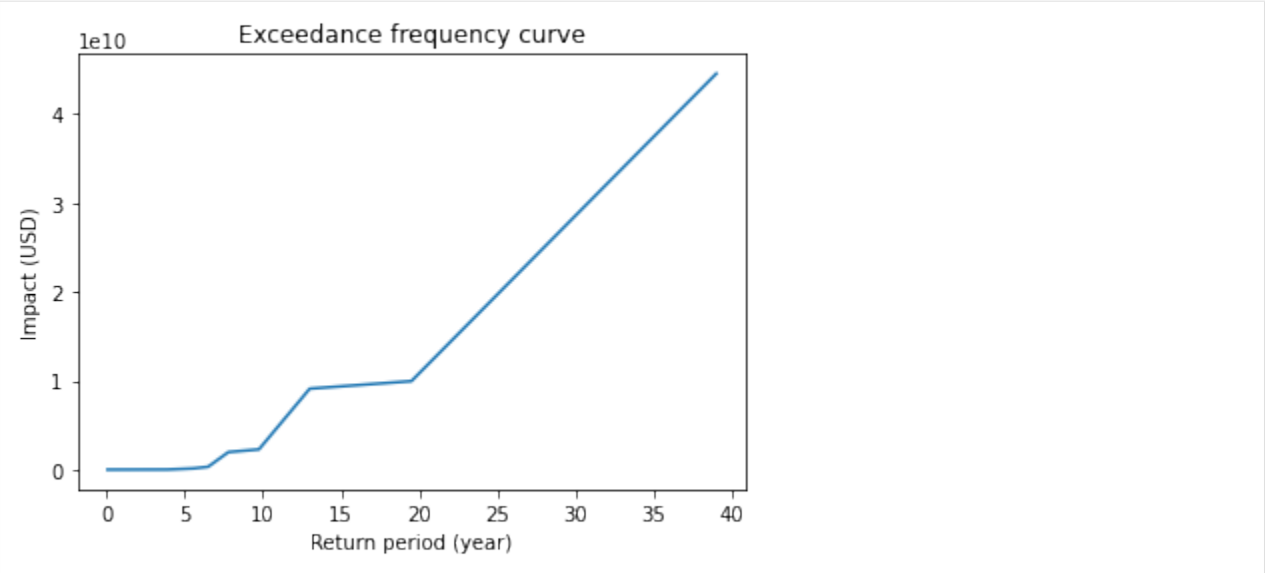
[20]:
```
from climada.engine import Impact

imp = Impact()
imp.calc(ent.exposures, ent.impact_funcs, haz)
```

A useful parameter for the `calc` method is `save_mat`. When set to `True` (default is `False`), the `Impact` object saves the calculated impact for each event at each point of exposure, stored as a (large) sparse matrix in the `imp_mat` attribute. This allows for more detailed analysis at the event level.

The `Impact` class includes a number of analysis tools. We can plot an exceedence frequency curve, showing us how often different damage thresholds are reached in our source data (remember this is only 40 years of storms, so not a full climatology!)

[21]:
```
freq_curve = imp.calc_freq_curve() # impact exceedence frequency curve
freq_curve.plot();

print('Expected average annual impact: {:.3e} USD'.format(imp.aai_agg))
```

```
Expected average annual impact: 1.754e+09 USD
```



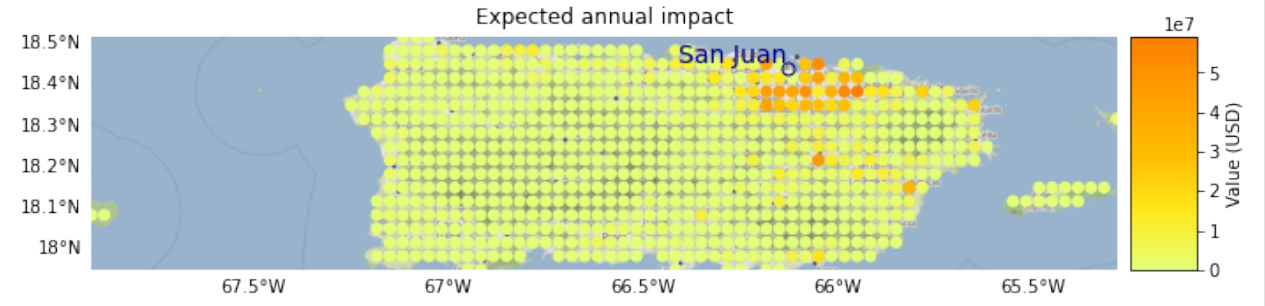
We can map the expected annual impact by exposure:

[22]:
```
imp.plot_basemap_eai_exposure(buffer=0.1); # average annual impact at each exposure
```

```
2021-04-23 11:52:47,128 - climada.util.coordinates - INFO - Setting geometry points.
2021-04-23 11:52:47,249 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
```

```
/Users/zeliestalhanske/python_projects/climada_python/climada/entity/exposures/base.py:
↪190: FutureWarning: Assigning CRS to a GeoDataFrame without a geometry column is now␣
↪deprecated and will not be supported in the future.
  self.gdf = GeoDataFrame(*args, **kwargs)
/Users/zeliestalhanske/miniconda3/envs/climada_env/lib/python3.8/site-packages/
↪contextily/tile.py:265: FutureWarning: The url format using 'tileX', 'tileY', 'tileZ'␣
↪as placeholders is deprecated. Please use '{x}', '{y}', '{z}' instead.
  warnings.warn(
```

```
2021-04-23 11:52:50,676 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
```



For additional functionality, including plotting the impacts of individual events, see the *Impact tutorial*.

> **Exercise:** Plot the impacts of Hurricane Maria. To do this you'll need to set `save_mat=True` in the earlier `Impact.calc()`.

We can save our variables in pickle format using the `save` function and load them with `load`. This will save your results in the folder specified in the configuration file. The default folder is a `results` folder which is created in the current

path (see default configuration file `climada/conf/defaults.conf`). However, we recommend to use CLIMADA's writers in `hdf5` or `csv` whenever possible.

[23]:
```
import os
from climada.util import save, load

### Uncomment this to save - saves by default to ./results/
# save('impact_puerto_rico_tc.p', imp)

### Uncomment this to read the saved data:
# abs_path = os.path.join(os.getcwd(), 'results/impact_puerto_rico_tc.p')
# data = load(abs_path)
```

`Impact` also has `write_csv()` and `write_excel()` methods to save the impact variables, and `write_sparse_csr()` to save the impact matrix (impact per event and exposure). Use the *Impact tutorial* to get more information about these functions and the class in general.

Adaptation options appraisal

Finally, let's look at a cost-benefit analysis. The adaptation measures defined with our `Entity` can be valued by estimating their cost-benefit ratio. This is done in the class `CostBenefit`.

Let us suppose that the socioeconomic and climatoligical conditions remain the same in 2040. We then compute the cost and benefit of every adaptation measure from our Hazard and Entity (and plot them) as follows:

[ ]:
```
from climada.engine import CostBenefit

cost_ben = CostBenefit()
cost_ben.calc(haz, ent, future_year=2040) # prints costs and benefits
cost_ben.plot_cost_benefit() # plot cost benefit ratio and averted damage of every␣
↪exposure
cost_ben.plot_event_view(return_per=(10, 20, 40)) # plot averted damage of each measure␣
↪for every return period
```

This is just the start. Analyses improve as we add more adaptation measures into the mix.

Cost-benefit calculations can also include - climate change, by specifying the `haz_future` parameter in `CostBenefit.calc()` - changes to economic exposure over time (or to whatever exposure you're modelling) by specifying the `ent_future` parameter in `CostBenefit.calc()` - different functions to calculate risk benefits. These are specified in `CostBenefit.calc()` and by default use changes to average annual impact - linear, sublinear and superlinear evolution of impacts between the present and future, specified in the `imp_time_depen` parameter in `CostBenefit.calc()`

And once future hazards and exposures are defined, we can express changes to impacts over time as waterfall diagrams. See the CostBenefit class for more details.

> **Exercise:** repeat the above analysis, creating future climate hazards (see the first exercise), and future exposures based on projected economic growth. Visualise it with the `CostBenefit.plot_waterfall()` method.

5.1.8 What next?

Thanks for following this tutorial! Take time to work on the exercises it suggested, or design your own risk analysis for your own topic. More detailed tutorials for individual classes were listed in the *Features* section.

Also, explore the full CLIMADA documentation and additional resources *described at the start of this document* to learn more about CLIMADA, its structure, its existing applications and how you can contribute.

5.2 Exposures class

5.2.1 What is an exposure?

Exposure describes the set of assets, people, livelihoods, infrastructures, etc. within an area of interest in terms of their geographic location, their value etc.; in brief - everything potentially exposed to hazards.

5.2.2 What options does CLIMADA offer for me to create an exposure?

CLIMADA has an `Exposures` class for this purpuse. An `Exposures` instance can be filled with your own data, or loaded from available default sources implemented through some Exposures-type classes from CLIMADA. If you have your own data, they can be provided in the formats of a `pandas.DataFrame`, a `geopandas.GeoDataFrame` or simply an `Excel` file. If you didn't collect your own data, exposures can be generated on the fly using CLIMADA's *LitPop*, BlackMarble or OpenStreetMap modules. See the respective tutorials to learn what exactly they contain and how to use them.

5.2.3 What does an exposure look like in CLIMADA?

An exposure is represented in the class `Exposures`, which contains a geopandas GeoDataFrame that is accessible through the `Exposures` `gdf` attribute. Certain columns of `gdf` *have to* be specified, while others are optional (this means that the package `climada.engine` also works without these variables set.) The full list looks like this:

Mandatory columns	Data Type	Description
latitude	float	latitude
longitude	float	longitude
value	float	a value for each exposure

Optional columns	Data Type	Description
impf_*	int	impact functions ids for hazard types.important attribute, since it relates the exposures to the hazard by specifying the impf_act functions.Ideally it should be set to the specific hazard (e.g. `impf_TC`) so that different hazards can be setin the same Exposures (e.g. `impf_TC` and `impf_FL`).If not provided, set to default `impf_` with ids 1 in check().
geometry	Point	geometry of type PointMain feature of geopandas DataFrame extensionComputed in method `set_geometry_points()`
deductible	float	deductible value for each exposure. Used for insurance
cover	float	cover value for each exposure. Used for insurance
category_id	int	category id (e.g. building code) for each exposure
region_id	int	region id (e.g. country ISO code) for each exposure
centr_*	int	centroids index for hazard type.There might be different hazards defined: centr_TC, centr_FL, ...Computed in method `assign_centroids()`

Metadata variables	Data Type	Description
crs	str or int	coordinate reference system, see GeoDataFrame.crs
tag	Tag	information about the source data
ref_year	int	reference year
value_unit	str	unit of the exposures' values
meta	dict	dictionary containing corresponding raster properties (if any):width, height, crs and transform must be present at least (transform needs to contain upper left corner!).Exposures might not contain all the points of the corresponding raster.

How is this tutorial structured?

Part 1: Defining exposures from your own data (DataFrame, GeoDataFrame, Excel)

Part 2: Loading exposures from CLIMADA-files or generating new ones (LitPop, BlackMarble, OSM)

Part 3: Visualizing exposures

Part 4: Writing (=saving) exposures

Part 5: What to do with large exposure data

Part 1: Defining exposures from your own data The essential structure of an exposure is similar, irrespective of the data type you choose to provide: As mentioned in the introduction, the key variables to be provided are `latitudes`, `longitudes` and `values` of your exposed assets. While not mandatory, but very useful to provide for the impact calculation at later stages: the impact function id (see `impf_*` in the table above). The following examples will walk you through how to specifiy those four variables, and demonstrate the use of a few more optional parameters on the go.

Exposures from a pandas DataFrame

In case you are unfamiliar with the data structure, check out the pandas DataFrame documentation.

[1]:
```
import numpy as np
from pandas import DataFrame
from climada.entity import Exposures

# Fill a pandas DataFrame with the 3 mandatory variables (latitude, longitude, value)␣
↪for a number of assets (10'000).
# We will do this with random dummy data for purely illustrative reasons:
exp_df = DataFrame()
n_exp = 100*100
# provide value
exp_df['value'] = np.arange(n_exp)
# provide latitude and longitude
lat, lon = np.mgrid[15 : 35 : complex(0, np.sqrt(n_exp)), 20 : 40 : complex(0, np.sqrt(n_
↪exp))]
exp_df['latitude'] = lat.flatten()
exp_df['longitude'] = lon.flatten()
```

[2]:
```
# For each exposure entry, specify which impact function should be taken for which␣
↪hazard type.
# In this case, we only specify the IDs for tropical cyclone (TC); here, each exposure␣
↪entry will be treated with
# the same impact function: the one that has ID '1':
# Of course, this will only be relevant at later steps during impact calculations.
exp_df['impf_TC'] = np.ones(n_exp, int)
```

[3]:
```
# Let's have a look at the pandas DataFrame
print('\x1b[1;03;30;30m' + 'exp_df is a DataFrame:', str(type(exp_df)) + '\x1b[0m')
print('\x1b[1;03;30;30m' + 'exp_df looks like:' + '\x1b[0m')
print(exp_df.head())
```

```
exp_df is a DataFrame: <class 'pandas.core.frame.DataFrame'>
exp_df looks like:
   value  latitude  longitude  impf_TC
0      0      15.0  20.000000        1
1      1      15.0  20.202020        1
2      2      15.0  20.404040        1
3      3      15.0  20.606061        1
4      4      15.0  20.808081        1
```

[4]:
```
# Generate Exposures from the pandas DataFrame. This step converts the DataFrame into
# a CLIMADA Exposures instance!
exp = Exposures(exp_df)
print('\n\x1b[1;03;30;30m' + 'exp has the type:', str(type(exp)))
print('and contains a GeoDataFrame exp.gdf:', str(type(exp.gdf)) + '\n\n\x1b[0m')

# set geometry attribute (shapely Points) from GeoDataFrame from latitude and longitude
exp.set_geometry_points()
print('\n' + '\x1b[1;03;30;30m' + 'check method logs:' + '\x1b[0m')
```

(continues on next page)

(continued from previous page)

```
# always apply the check() method in the end. It puts metadata that has not been␣
↪assigned,
# and points out missing mandatory data
exp.check()
```

```
exp has the type: <class 'climada.entity.exposures.base.Exposures'>
and contains a GeoDataFrame exp.gdf: <class 'geopandas.geodataframe.GeoDataFrame'>


check method logs:
```

[5]:
```
# let's have a look at the Exposures instance we created!
print('\n' + '\x1b[1;03;30;30m'  + 'exp looks like:' + '\x1b[0m')
print(exp)
```

```
exp looks like:
tag:  File:
 Description:
ref_year: 2018
value_unit: USD
meta: {'crs': 'EPSG:4326'}
crs: EPSG:4326
data:
      value  latitude  longitude  impf_TC                    geometry
0         0      15.0  20.000000        1  POINT (20.00000 15.00000)
1         1      15.0  20.202020        1  POINT (20.20202 15.00000)
2         2      15.0  20.404040        1  POINT (20.40404 15.00000)
3         3      15.0  20.606061        1  POINT (20.60606 15.00000)
4         4      15.0  20.808081        1  POINT (20.80808 15.00000)
...     ...       ...        ...      ...                        ...
9995   9995      35.0  39.191919        1  POINT (39.19192 35.00000)
9996   9996      35.0  39.393939        1  POINT (39.39394 35.00000)
9997   9997      35.0  39.595960        1  POINT (39.59596 35.00000)
9998   9998      35.0  39.797980        1  POINT (39.79798 35.00000)
9999   9999      35.0  40.000000        1  POINT (40.00000 35.00000)

[10000 rows x 5 columns]
```

Exposures from a geopandas GeoDataFrame In case you are unfamiliar with with data structure, check out the geopandas GeoDataFrame documentation. The main difference to the example above (pandas DataFrame) is that, while previously, we provided latitudes and longitudes which were then converted to a `geometry` GeoSeries using the `set_geometry_points` method, GeoDataFrames alread come with a defined `geometry` GeoSeries. In this case, we take the geometry info and use the `set_lat_lon` method to explicitly provide latitudes and longitudes. This example focuses on data with POINT geometry, but in principle, other geometry types (such as POLYGON and MULTIPOLYGON) would work as well.

[6]:
```
import numpy as np
import geopandas as gpd
from climada.entity import Exposures

# Read spatial info from an external file into GeoDataFrame
```

(continues on next page)

(continued from previous page)

```
world = gpd.read_file(gpd.datasets.get_path('naturalearth_cities'))
print('\x1b[1;03;30;30m' + 'World is a GeoDataFrame:', str(type(world)) + '\x1b[0m')
print('\x1b[1;03;30;30m' + 'World looks like:' + '\x1b[0m')
print(world.head())
```

```
World is a GeoDataFrame: <class 'geopandas.geodataframe.GeoDataFrame'>
World looks like:
           name                    geometry
0  Vatican City  POINT (12.45339 41.90328)
1    San Marino  POINT (12.44177 43.93610)
2         Vaduz   POINT (9.51667 47.13372)
3    Luxembourg   POINT (6.13000 49.61166)
4       Palikir  POINT (158.14997 6.91664)
```

[7]:
```
# Generate Exposures: value, latitude and longitude for each exposure entry.
# Convert GeoDataFrame into Exposure instance
exp_gpd = Exposures(world)
print('\n' + '\x1b[1;03;30;30m' + 'exp_gpd is an Exposures:', str(type(exp_gpd)) + '\
↪x1b[0m')
# add random values to entries
exp_gpd.gdf['value'] = np.arange(world.shape[0])
# set latitude and longitude attributes from geometry
exp_gpd.set_lat_lon()
```

```

exp_gpd is an Exposures: <class 'climada.entity.exposures.base.Exposures'>
```

[8]:
```
# For each exposure entry, specify which impact function should be taken for which␣
↪hazard type.
# In this case, we only specify the IDs for tropical cyclone (TC); here, each exposure␣
↪entry will be treated with
# the same impact function: the one that has ID '1':
# Of course, this will only be relevant at later steps during impact calculations.
exp_gpd.gdf['impf_TC'] = np.ones(world.shape[0], int)
print('\n' + '\x1b[1;03;30;30m' + 'check method logs:' + '\x1b[0m')

# as always, run check method to assign meta-data and check for missing mandatory␣
↪variables.
exp_gpd.check()
```

```

check method logs:
```

[9]:
```
# let's have a look at the Exposures instance we created!
print('\n' + '\x1b[1;03;30;30m'  + 'exp_gpd looks like:' + '\x1b[0m')
print(exp_gpd)
```

```

exp_gpd looks like:
tag:  File:
 Description:
ref_year: 2018
value_unit: USD
```

(continues on next page)

(continued from previous page)

```
meta: {'crs': <Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
}
crs: epsg:4326
data:
             name                     geometry  value   latitude   longitude  \
0    Vatican City    POINT (12.45339 41.90328)      0  41.903282   12.453387
1      San Marino    POINT (12.44177 43.93610)      1  43.936096   12.441770
2           Vaduz     POINT (9.51667 47.13372)      2  47.133724    9.516669
3      Luxembourg     POINT (6.13000 49.61166)      3  49.611660    6.130003
4         Palikir    POINT (158.14997 6.91664)      4   6.916644  158.149974
..            ...                          ...    ...        ...         ...
197         Cairo    POINT (31.24802 30.05191)    197  30.051906   31.248022
198         Tokyo   POINT (139.74946 35.68696)    198  35.686963  139.749462
199         Paris     POINT (2.33139 48.86864)    199  48.868639    2.331389
200      Santiago  POINT (-70.66899 -33.44807)    200 -33.448068  -70.668987
201     Singapore    POINT (103.85387 1.29498)    201   1.294979  103.853875

     impf_TC
0          1
1          1
2          1
3          1
4          1
..       ...
197        1
198        1
199        1
200        1
201        1

[202 rows x 6 columns]
```

The fact that `Exposures` is built around a `geopandas.GeoDataFrame` offers all the useful functionalities that come with the package. The following examples showcase only a few of those.

[10]:
```
# Example 1: extract data in a region: latitudes between -5 and 5
sel_exp = exp_gpd.copy()  # to keep the original exp_gpd Exposures data
sel_exp.gdf = sel_exp.gdf.cx[:, -5:5]

print('\n' + '\x1b[1;03;30;30m' + 'sel_exp contains a subset of the original data' + '\
↪x1b[0m')
sel_exp.gdf.head()
```

```
sel_exp contains a subset of the original data
```

[10]:

	name	geometry	value	latitude	longitude	impf_TC
9	Tarawa	POINT (173.01757 1.33819)	9	1.338188	173.017571	1
13	Kigali	POINT (30.05859 -1.95164)	13	-1.951644	30.058586	1
15	Juba	POINT (31.58003 4.82998)	15	4.829975	31.580026	1
27	Bujumbura	POINT (29.36001 -3.37609)	27	-3.376087	29.360006	1
48	Kampala	POINT (32.58138 0.31860)	48	0.318605	32.581378	1

[11]:
```
# Example 2: extract data in a polygon
from shapely.geometry import Polygon
sel_polygon = exp_gpd.copy()  # to keep the original exp_gpd Exposures data

poly = Polygon([(0, -10), (0, 10), (10, 5)])
sel_polygon.gdf = sel_polygon.gdf[sel_polygon.gdf.intersects(poly)]

# Let's have a look. Again, the sub-selection is a GeoDataFrame!
print('\n' + '\x1b[1;03;30;30m' + 'sel_exp contains a subset of the original data' + '\
↪x1b[0m')
sel_polygon.gdf
```

```
sel_exp contains a subset of the original data
```

[11]:

	name	geometry	value	latitude	longitude	impf_TC
36	Lome	POINT (1.22081 6.13388)	36	6.133883	1.220811	1
84	Malabo	POINT (8.78328 3.75002)	84	3.750015	8.783278	1
113	Cotonou	POINT (2.51804 6.40195)	113	6.401954	2.518045	1
125	Sao Tome	POINT (6.73333 0.33340)	125	0.333402	6.733325	1

[12]:
```
# Example 3: change coordinate reference system
# use help to see more options: help(sel_exp.to_crs)
sel_polygon.to_crs(epsg=3395, inplace=True)
print('\n' + '\x1b[1;03;30;30m' + 'the crs has changed to ' +str(sel_polygon.crs))
print('the values for latitude and longitude are now according to the new coordinate␣
↪system: \x1b[0m')
sel_polygon.gdf
```

```
the crs has changed to epsg:3395
the values for latitude and longitude are now according to the new coordinate system:
```

[12]:

	name	geometry	value	latitude	longitude	impf_TC
36	Lome	POINT (135900.088 679566.334)	36	679566.333952	135900.087901	1
84	Malabo	POINT (977749.984 414955.551)	84	414955.550857	977749.983897	1
113	Cotonou	POINT (280307.458 709388.810)	113	709388.810160	280307.458315	1
125	Sao Tome	POINT (749550.327 36865.909)	125	36865.908682	749550.327404	1

[13]:
```
# Example 4: concatenate exposures
exp_all = Exposures.concat([sel_polygon, sel_exp.to_crs(epsg=3395)])

# the output is of type Exposures
print('exp_all type and number of rows:', type(exp_all), exp_all.gdf.shape[0])
print('number of unique rows:', exp_all.gdf.drop_duplicates().shape[0])

# NaNs will appear in the missing values
exp_all.gdf.head()
```

```
exp_all type and number of rows: <class 'climada.entity.exposures.base.Exposures'> 25
number of unique rows: 23
```

[13]:

	name	geometry	value	latitude	longitude	impf_TC
0	Lome	POINT (135900.088 679566.334)	36	679566.333952	1.359001e+05	1
1	Malabo	POINT (977749.984 414955.551)	84	414955.550857	9.777500e+05	1
2	Cotonou	POINT (280307.458 709388.810)	113	709388.810160	2.803075e+05	1
3	Sao Tome	POINT (749550.327 36865.909)	125	36865.908682	7.495503e+05	1
4	Tarawa	POINT (19260227.883 147982.749)	9	147982.748978	1.926023e+07	1

Exposures of any file type supported by Geopandas and Pandas

Geopandas can read almost any vector-based spatial data format including ESRI shapefile, GeoJSON files and more, see readers geopandas. Pandas supports formats such as csv, html or sql; see readers pandas. Using the corresponding readers, `DataFrame` and `GeoDataFrame` can be filled and provided to `Exposures` following the previous examples.

Exposures from an excel file

If you manually collect exposure data, Excel may be your preferred option. In this case, it is easiest if you format your data according to the structure provided in the template `climada_python/data/system/entity_template.xlsx`, in the sheet `assets`.

[14]:
```
import pandas as pd
from climada.util.constants import ENT_TEMPLATE_XLS
from climada.entity import Exposures

# Read your Excel file into a pandas DataFrame (we will use the template example for␣
↪this demonstration):
file_name = ENT_TEMPLATE_XLS
exp_templ = pd.read_excel(file_name)

# Let's have a look at the data:
print('\x1b[1;03;30;30m' + 'exp_templ is a DataFrame:', str(type(exp_templ)) + '\x1b[0m')
print('\x1b[1;03;30;30m' + 'exp_templ looks like:' + '\x1b[0m')
exp_templ.head()
```

```
exp_templ is a DataFrame: <class 'pandas.core.frame.DataFrame'>
exp_templ looks like:
```

[14]:
```
    latitude  longitude         value  deductible         cover  region_id  \
0  26.933899 -80.128799  1.392750e+10           0  1.392750e+10          1
1  26.957203 -80.098284  1.259606e+10           0  1.259606e+10          1
2  26.783846 -80.748947  1.259606e+10           0  1.259606e+10          1
3  26.645524 -80.550704  1.259606e+10           0  1.259606e+10          1
4  26.897796 -80.596929  1.259606e+10           0  1.259606e+10          1

   category_id  impf_TC  centr_TC  impf_FL  centr_FL
0            1        1         1        1         1
1            1        1         2        1         2
2            1        1         3        1         3
3            1        1         4        1         4
4            1        1         5        1         5
```

As we can see, the general structure is the same as always: the exposure has `latitude`, `longitude` and `value` columns. Further, this example specified several impact function ids: some for Tropical Cyclones (`impf_TC`), and some for Floods (`impf_FL`). It also provides some meta-info (`region_id`, `category_id`) and insurance info relevant to the impact calculation in later steps (`cover`, `deductible`).

[15]:
```
# Generate an Exposures instance from the dataframe.
exp_templ = Exposures(exp_templ)
print('\n' + '\x1b[1;03;30;30m' + 'exp_templ is now an Exposures:', str(type(exp_templ))␣
↪+ '\x1b[0m')

# set geometry attribute (shapely Points) from GeoDataFrame from latitude and longitude
print('\n' + '\x1b[1;03;30;30m' + 'set_geometry logs:' + '\x1b[0m')
exp_templ.set_geometry_points()
# as always, run check method to include metadata and check for missing mandatory␣
↪parameters

print('\n' + '\x1b[1;03;30;30m' + 'check exp_templ:' + '\x1b[0m')
exp_templ.check()
```

```
exp_templ is now an Exposures: <class 'climada.entity.exposures.base.Exposures'>

set_geometry logs:

check exp_templ:
```

[16]:
```
# Let's have a look at our Exposures instance!
print('\n' + '\x1b[1;03;30;30m'  + 'exp_templ.gdf looks like:' + '\x1b[0m')
exp_templ.gdf.head()
```

```
exp_templ.gdf looks like:
```

[16]:
```
    latitude  longitude         value  deductible         cover  region_id  \
0  26.933899 -80.128799  1.392750e+10           0  1.392750e+10          1
1  26.957203 -80.098284  1.259606e+10           0  1.259606e+10          1
2  26.783846 -80.748947  1.259606e+10           0  1.259606e+10          1
3  26.645524 -80.550704  1.259606e+10           0  1.259606e+10          1
```

(continues on next page)

(continued from previous page)

```
4  26.897796 -80.596929  1.259606e+10           0  1.259606e+10          1

   category_id  impf_TC  centr_TC  impf_FL  centr_FL  \
0            1        1         1        1         1
1            1        1         2        1         2
2            1        1         3        1         3
3            1        1         4        1         4
4            1        1         5        1         5

                     geometry
0  POINT (-80.12880 26.93390)
1  POINT (-80.09828 26.95720)
2  POINT (-80.74895 26.78385)
3  POINT (-80.55070 26.64552)
4  POINT (-80.59693 26.89780)
```

Exposures from a raster file

Last but not least, you may have your exposure data stored in a raster file. Raster data may be read in from any file-type supported by rasterio.

[17]:
```
from rasterio.windows import Window
from climada.util.constants import HAZ_DEMO_FL
from climada.entity import Exposures

# We take an example with a dummy raster file (HAZ_DEMO_FL), running the method set_from_
↪raster directly loads the
# necessary info from the file into an Exposures instance.
exp_raster = Exposures.from_raster(HAZ_DEMO_FL, window= Window(10, 20, 50, 60))
# There are several keyword argument options that come with the set_from_raster method␣
↪(such as
# specifying a window, if not the entire file should be read, or a bounding box. Check␣
↪them out.
```

[18]:
```
# As always, run the check method, such that metadata can be assigned and checked for␣
↪missing mandatory parameters.
exp_raster.check()
print('Meta:', exp_raster.meta)
```

```
Meta: {'driver': 'GSBG', 'dtype': 'float32', 'nodata': 1.701410009187828e+38, 'width':␣
↪50, 'height': 60, 'count': 1, 'crs': CRS.from_epsg(4326), 'transform': Affine(0.
↪009000000000000341, 0.0, -69.2471495969998,
       0.0, -0.009000000000000341, 10.248220966978932)}
```

[19]:
```
# Let's have a look at the Exposures instance!
print('\n' + '\x1b[1;03;30;30m'  + 'exp_raster looks like:' + '\x1b[0m')
exp_raster.gdf.head()
```

exp_raster looks like:

[19]:

	longitude	latitude	value	impf_
0	-69.24265	10.243721	0.0	1
1	-69.23365	10.243721	0.0	1
2	-69.22465	10.243721	0.0	1
3	-69.21565	10.243721	0.0	1
4	-69.20665	10.243721	0.0	1

Part 2: Loading CLIMADA-generated exposure files or generating new ones In case you already have a CLIMADA-generated file containing `Exposures` info, you can of course load it back into memory. Most likely, the data format will either be of `.hdf5` or of `.mat`. In case you neither have your own data, nor a CLIMADA-generated file, you can also create an exposure on the fly using one of the three CLIMADA-internal exposure generators: *LitPop*, BlackMarble or OpenStreetMap modules. The latter three are extensively described in their own, linked, tutorials.

[20]:
```
# read  generated with the Python version with from_hdf5()
# note: for .mat data, use the method from_mat() analogously.
from climada.util.constants import EXP_DEMO_H5

exp_hdf5 = Exposures.from_hdf5(EXP_DEMO_H5)
exp_hdf5.check()
print(type(exp_hdf5))
```

```
<class 'climada.entity.exposures.base.Exposures'>
```

Before you leave ...

After defining an `Exposures` instance use always the `check()` method to see which attributes are missing. This method will raise an ERROR if `value`, `longitude` or `latitude` ar missing and an INFO messages for the the optional variables not set.

Part 3: Visualize Exposures The method `plot_hexbin()` uses cartopy and matplotlib's hexbin function to represent the exposures values as 2d bins over a map. Configure your plot by fixing the different inputs of the method or by modifying the returned `matplotlib` figure and axes.

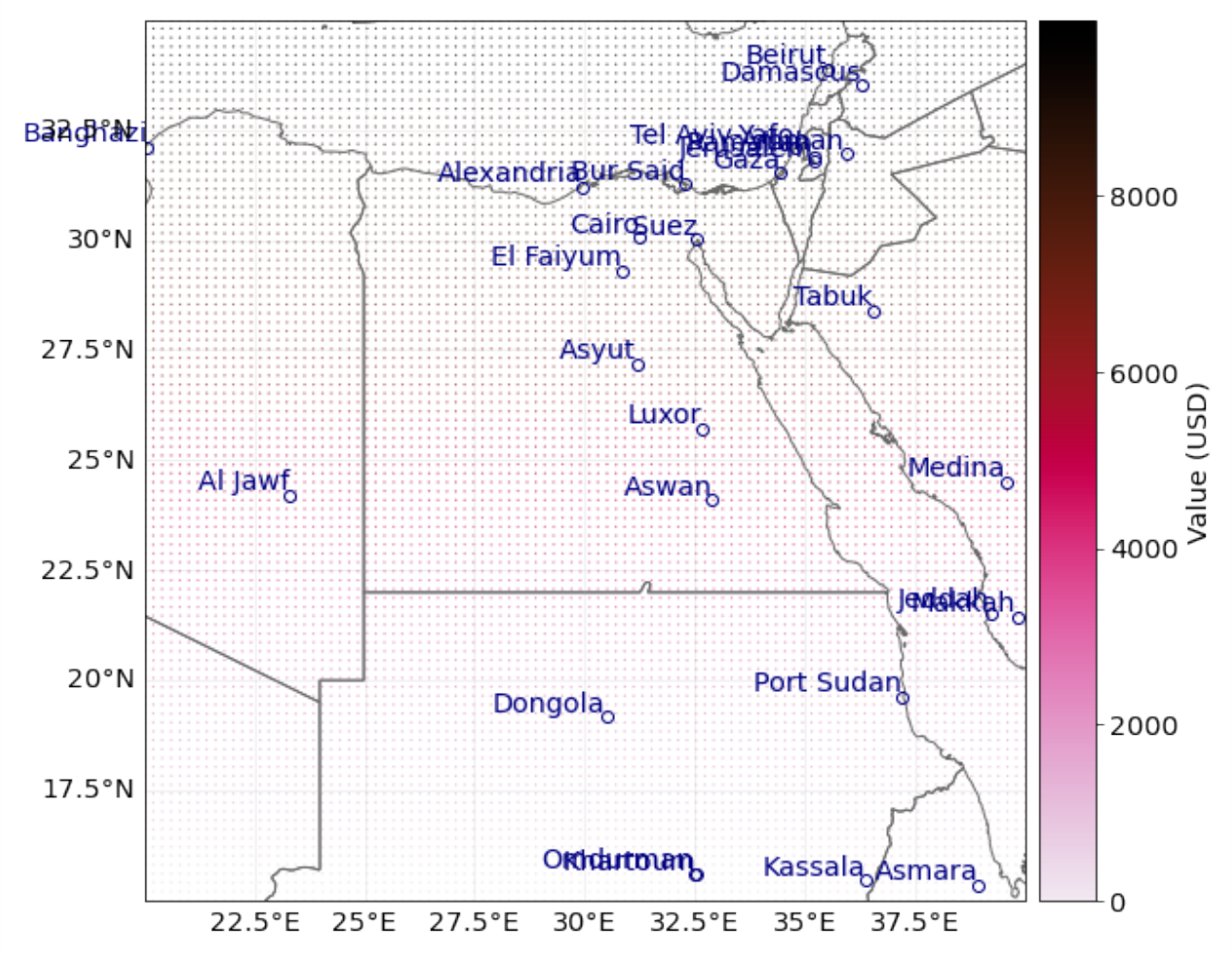
The method `plot_scatter()` uses cartopy and matplotlib's scatter function to represent the points values over a 2d map. As usal, it returns the figure and axes, which can be modify aftwerwards.

The method `plot_raster()` rasterizes the points into the given resolution. Use the `save_tiff` option to save the resulting `tiff` file and the `res_raster`option to re-set the raster's resolution.

Finally, the method `plot_basemap()` plots the scatter points over a satellite image using contextily library.

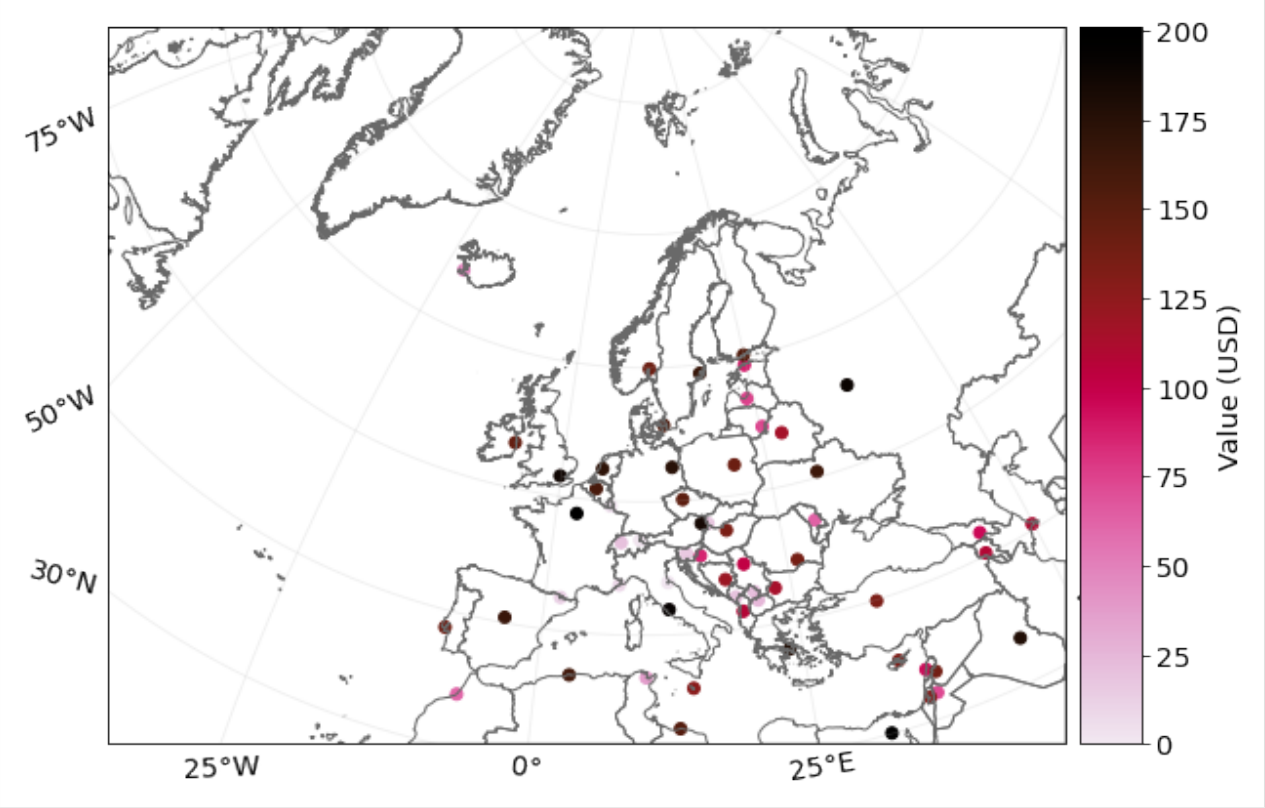
[21]:
```
# Example 1: plot_hexbin method
print('\x1b[1;03;30;30m' + 'Plotting exp_df.' + '\x1b[0m')
axs = exp.plot_hexbin()

# further methods to check out:
# axs.set_xlim(15, 45) to modify x-axis borders, axs.set_ylim(10, 40) to modify y-axis␣
↪borders
# further keyword arguments to play around with: pop_name, buffer, gridsize, ...
```

```
Plotting exp_df.
```



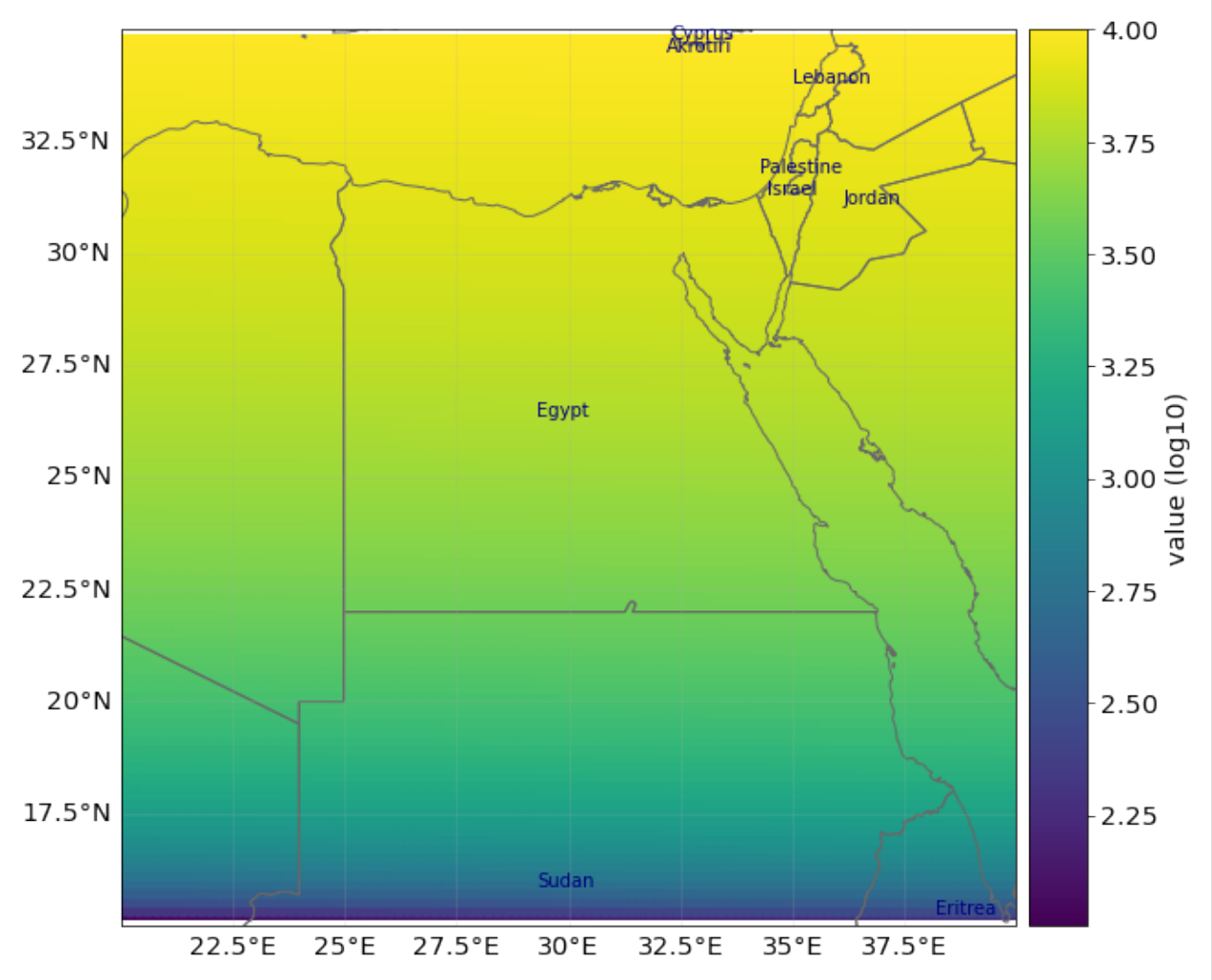
[22]:
```
# Example 2: plot_scatter method

exp_gpd.to_crs('epsg:3035', inplace=True)
exp_gpd.plot_scatter(pop_name=False)
```

[22]:
```
<GeoAxesSubplot:>
```



[23]:
```
# Example 3: plot_raster method
from climada.util.plot import add_cntry_names # use climada's plotting utilities
ax = exp.plot_raster() # plot with same resolution as data
add_cntry_names(ax, [exp.gdf.longitude.min(), exp.gdf.longitude.max(), exp.gdf.latitude.
↪min(), exp.gdf.latitude.max()])

# use keyword argument save_tiff='filepath.tiff' to save the corresponding raster in tiff␣
↪format
# use keyword argument raster_res='desired number' to change resolution of the raster.
```

```
2021-06-04 17:07:42,654 - climada.util.coordinates - INFO - Raster from resolution 0.
↪20202020202019355 to 0.20202020202019355.
```

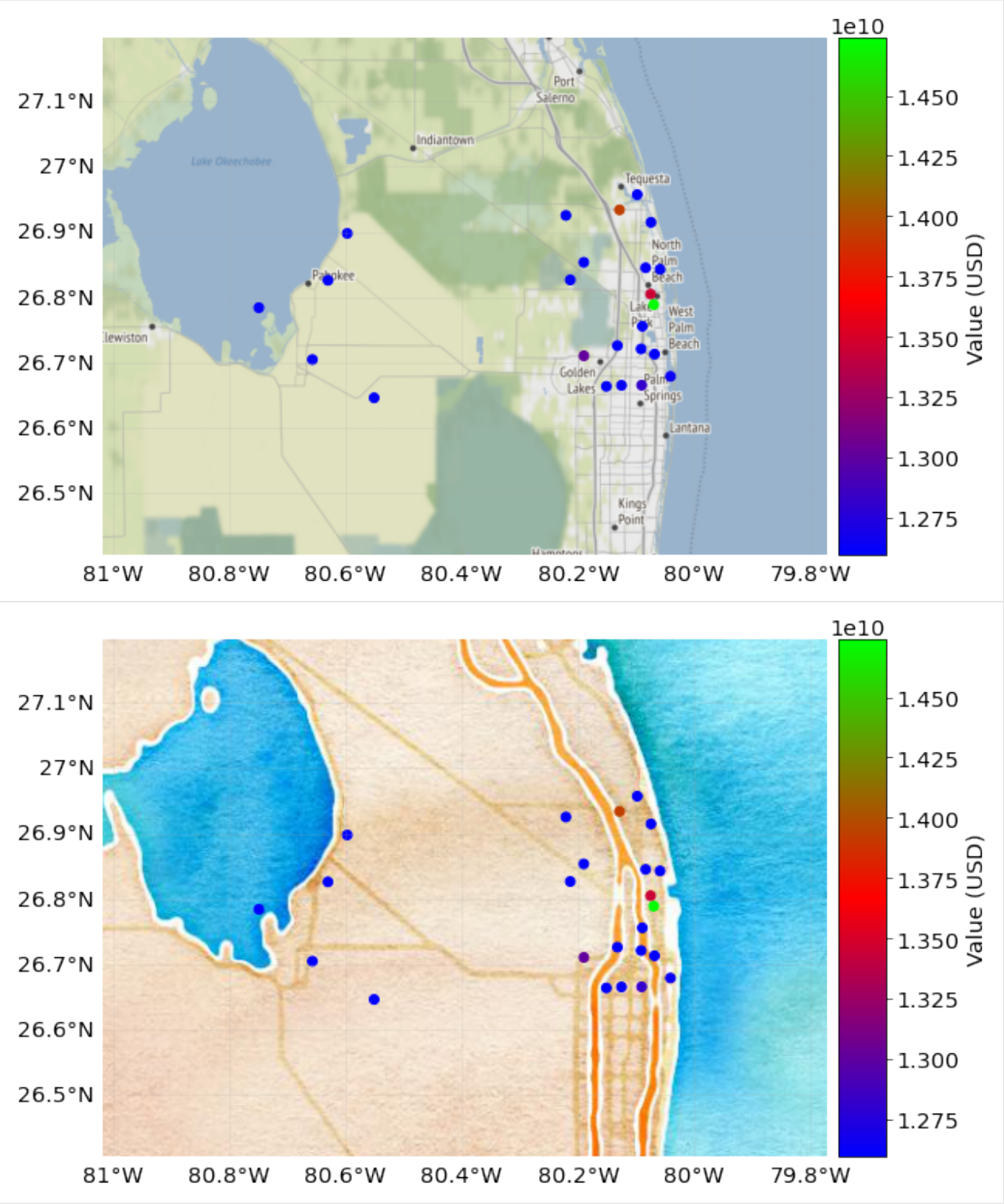


[24]:
```
# Example 4: plot_basemap method
import contextily as ctx
# select the background image from the available ctx.sources
ax = exp_templ.plot_basemap(buffer=30000, cmap='brg') # using open street map
ax = exp_templ.plot_basemap(buffer=30000, url=ctx.sources._T_WATERCOLOR, cmap='brg',␣
↪zoom=9) # set image zoom
```

```
2021-06-04 17:07:51,154 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
```

```
/Users/zeliestalhanske/miniconda3/envs/climada_env/lib/python3.8/site-packages/
↪contextily/tile.py:265: FutureWarning: The url format using 'tileX', 'tileY', 'tileZ'␣
↪as placeholders is deprecated. Please use '{x}', '{y}', '{z}' instead.
  warnings.warn(
```

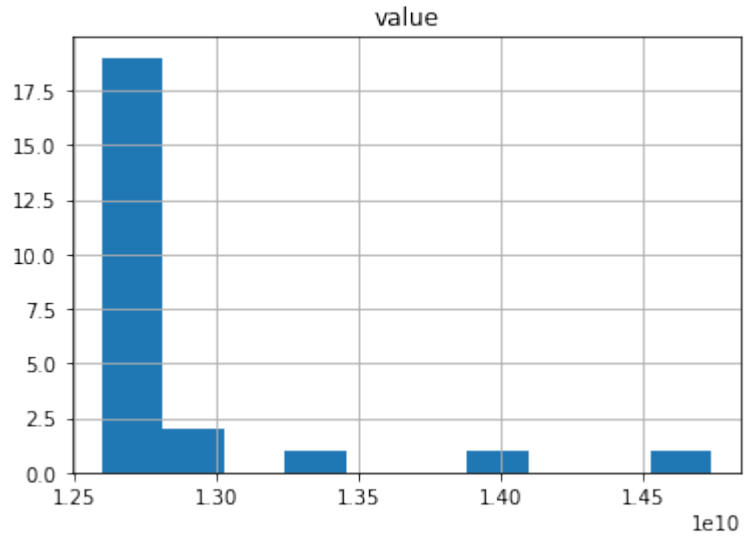
```
2021-06-04 17:07:55,046 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
2021-06-04 17:07:55,124 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
2021-06-04 17:07:58,604 - climada.entity.exposures.base - INFO - Setting latitude and␣
↪longitude attributes.
```



Since `Exposures` is a `GeoDataFrame`, any function for visualization from geopandas can be used. Check making maps and examples gallery.

[25]:
```
# other visualization types
exp_templ.gdf.hist(column='value')
```

[25]:
```
array([[<AxesSubplot:title={'center':'value'}>]], dtype=object)
```



Part 4: Write (Save) Exposures Exposures can be saved in any format available for `GeoDataFrame` (see fiona.supported_drivers) and `DataFrame` (pandas IO tools). Take into account that in many of these formats the metadata (e.g. variables `ref_year`, `value_unit` and `tag`) will not be saved. Use instead the format hdf5 provided by `Exposures` methods `write_hdf5()` and `from_hdf5()` to handle all the data.

[26]:
```
import fiona; fiona.supported_drivers
from climada import CONFIG
results = CONFIG.local_data.save_dir.dir()

# GeoDataFrame default: ESRI shape file in current path. metadata not saved!
exp_templ.gdf.to_file(results.joinpath('exp_templ'))

# DataFrame save to csv format. geometry writen as string, metadata not saved!
exp_templ.gdf.to_csv(results.joinpath('exp_templ.csv'), sep='\t')
```

[27]:
```
# write as hdf5 file
exp_templ.write_hdf5(results.joinpath('exp_temp.h5'))
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/IPython/core/interactiveshell.py:3441:␣
↪PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block2_values] [items->Index([
↪'geometry'], dtype='object')]

  exec(code_obj, self.user_global_ns, self.user_ns)
```

Finallly, as with any Python object, use climada's save option to save it in pickle format.

[28]:
```
# save in pickle format
from climada.util.save import save
# this generates a results folder in the current path and stores the output there
save('exp_templ.pkl.p', exp_templ) # creates results folder and stores there
```

Part 5: Dask - improving performance for big exposure Dask is used in some methods of CLIMADA and can be

activated easily by proving the scheduler.

[29]:
```
# set_geometry_points is expensive for big exposures
# for small amount of data, the execution time might be even greater when using dask
exp.gdf.drop(columns=['geometry'], inplace=True)
print(exp.gdf.head())
%time exp.set_geometry_points(scheduler='processes')
print(exp.gdf.head())
```

```
   value  latitude  longitude  impf_TC
0      0      15.0  20.000000        1
1      1      15.0  20.202020        1
2      2      15.0  20.404040        1
3      3      15.0  20.606061        1
4      4      15.0  20.808081        1
CPU times: user 243 ms, sys: 116 ms, total: 359 ms
Wall time: 2.52 s
   value  latitude  longitude  impf_TC                    geometry
0      0      15.0  20.000000        1  POINT (20.00000 15.00000)
1      1      15.0  20.202020        1  POINT (20.20202 15.00000)
2      2      15.0  20.404040        1  POINT (20.40404 15.00000)
3      3      15.0  20.606061        1  POINT (20.60606 15.00000)
4      4      15.0  20.808081        1  POINT (20.80808 15.00000)
```

5.3 LitPop class

5.3.1 Introduction

`LitPop` is an *Exposures*-type class. It is used to initiate grided exposure data with estimates of either asset value, economic activity or population based on nightlight intensity and population count data.

Background

The modeling of economic disaster risk on a global scale requires high-resolution maps of exposed asset values. We have developed a generic and scalable method to downscale national asset value estimates proportional to a combination of nightlight intensity ("Lit") and population data ("Pop").

Asset exposure value is disaggregated to the grid points proportionally to $Lit^m Pop^n$, computed at each grid cell:

$Lit^m Pop^n = Lit^m * Pop^n$, with $exponents = [m, n] \in^+$ (Default values are $m = n = 1$).

For more information please refer to the related publication (https://doi.org/10.5194/essd-12-817-2020) and data archive (https://doi.org/10.3929/ethz-b-000331316).

How to cite: Eberenz, S., Stocker, D., Röösli, T., and Bresch, D. N.: *Asset exposure data for global physical risk assessment*, Earth Syst. Sci. Data, 12, 817–833, https://doi.org/10.5194/essd-12-817-2020, 2020.

Input data

Note: All required data except for the population data from Gridded Population of the World (GPW) is downloaded automatically when an `LitPop.set_*` method is called.

Nightlight intensity

Black Marble annual composite of the VIIRS day-night band (Grayscale) at 15 arcsec resolution is downloaded from the NASA Earth Observatory: https://earthobservatory.nasa.gov/Features/NightLights (available for 2012 and 2016 at 15 arcsec resolution (~500m)). The first time a nightlight image is used, it is downloaded and stored locally. This might take some time.

Population count

Gridded Population of the World (GPW), v4: Population Count, v4.10, v4.11 or later versions (2000, 2005, 2010, 2015, 2020), available from http://sedac.ciesin.columbia.edu/data/collection/gpw-v4/sets/browse.

The GPW file of the year closest to the requested year (`reference_year`) is required. To download GPW data a (free) login for the NASA SEDAC website is required.

Direct download links are avilable, also for older versions, i.e.: - v4.11: http://sedac.ciesin.columbia.edu/downloads/data/gpw-v4/gpw-v4-population-count-rev11/gpw-v4-population-count-rev11_2015_30_sec_tif.zip - v4.10: http://sedac.ciesin.columbia.edu/downloads/data/gpw-v4/gpw-v4-population-count-rev10/gpw-v4-population-count-rev10_2015_30_sec_tif.zip, - Overview over all versions of GPW v.4: https://beta.sedac.ciesin.columbia.edu/data/collection/gpw-v4/sets/browse

The population data from GWP needs to be downloaded manually as TIFF from this site and placed in the `SYSTEM_DIR` folder of your climada installation.

Downloading existing LitPop asset exposure data

Readily computed LitPop asset exposure data based on $Lit^1 Pop^1$ for 224 countries, distributing produced capital / non-financial wealth of 2014 at a resolution of 30 arcsec can be downloaded from the ETH Research Repository: https://doi.org/10.3929/ethz-b-000331316. The dataset contains gridded data for more than 200 countries as CSV files.

5.3.2 Attributes

The `LitPop` class inherits from `` `Exposures <climada_entity_Exposures.ipynb#Exposures-class>`__ ``. It adds the following attributes:

```
exponents : Defining powers (m, n) with which nightlights and population go into Lit**m␣
↪* Pop**n.
fin_mode : Socio-economic indicator to be used as total asset value for diaggregation.
gpw_version : Version number of GPW population data, e.g. 11 for v4.11
```

fin_mode

The choice of `fin_mode` is crucial. Implemented choices are: * `'pc'`: produced capital (Source: World Bank), incl. manufactured or built assets such as machinery, equipment, and physical structures. The pc-data is stored in the subfolder *data/system/Wealth-Accounts_CSV/*. Source: https://datacatalog.worldbank.org/dataset/wealth-accounting * `'pop'`: population count (source: GPW, same as gridded population) * `'gdp'`: gross-domestic product (Source: World Bank) * `'income_group'`: gdp multiplied by country's income group+1 * `'nfw'`: non-financial household wealth (Source: Credit Suissey) * `'tw'`: total household wealth (Source: Credit Suisse) * `'norm'`: normalized, total value of country or region is 1. * `'none'`: None – LitPop per pixel is returned unchanged

Regarding the GDP (nominal GDP at current USD) and income group values, they are obtained from the World Bank using the pandas-datareader API. If a value is missing, the value of the closest year is considered. When no values are provided from the World Bank, we use the Natural Earth repository values.

5.3.3 Key Methods

- `from_countries`: set exposure for one or more countries, see section `from_countries` below.
- `from_nightlight_intensity`: wrapper around `from_countries` and `from_shape` to load nightlight data to exposure.
- `from_population`: wrapper around `from_countries` and `from_shape_population` to load pure population data to exposure. This can be used to initiate a population exposure set.
- `from_shape_and_countries`: given a shape and a list of countries, exposure is initiated for the countries and then cropped to the shape. See section *Set custom shapes* below.
- `from_shape`: given any shape or geometry and an estimate of total values, exposure is initiated for the shape directly. See section *Set custom shapes* below.

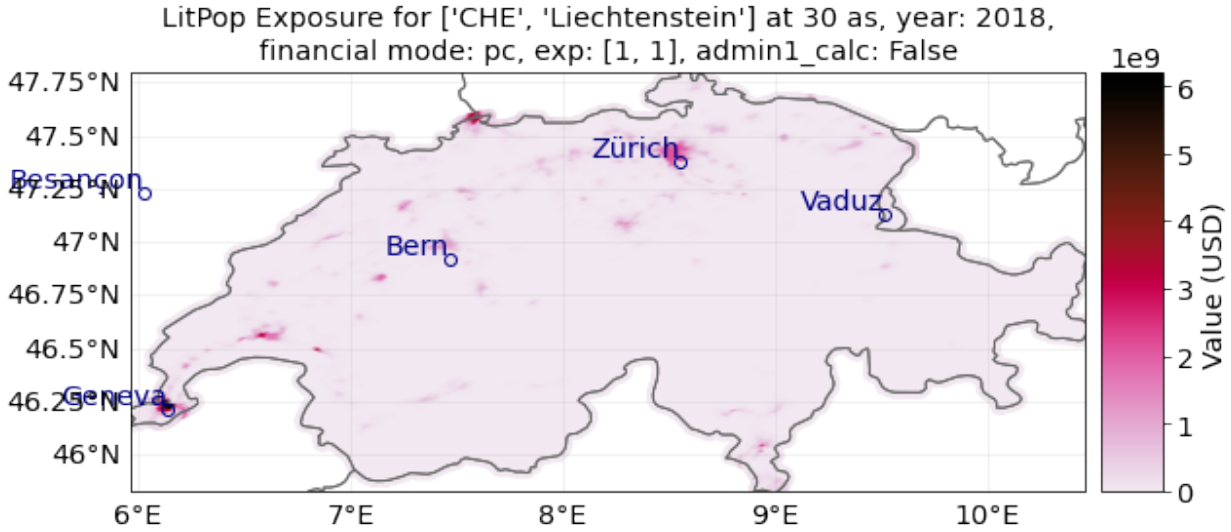
[2]:
```
# Import class LitPop:
from climada.entity import LitPop
```

from_countries

In the following, we will create exposure data sets and plots for a variety of countries, comparing different settings. #### Default Settings Per default, the exposure entity was initiated using the default parameters, i.e. a resolution of 30 arcsec, produced capital 'pc' as total asset value and using the exponents $(1, 1)$.

[5]:
```
# Initiate a default LitPop exposure entity for Switzerland and Liechtenstein (ISO3-
↪Codes 'CHE' and 'LIE'):
try:
    exp = LitPop.from_countries(['CHE', 'Liechtenstein']) # you can provide either␣
↪single countries or a list of countries
except FileExistsError as err:
    print("Reason for error: The GPW population data has not been downloaded, c.f.␣
↪section 'Input data' above.")
    raise err
exp.plot_scatter()

# Note that `exp.gdf.region_id` is a number identifying each country:
print('\n Region IDs (`region_id`) in this exposure:')
print(exp.gdf.region_id.unique())
```

```
2021-10-19 17:03:20,108 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2021-10-19 17:03:23,876 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2021-10-19 17:03:23,953 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2021-10-19 17:03:24,476 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.

 Region IDs (`region_id`) in this exposure:
[756 438]
```



fin_mode, resolution and exponents

Instead on produced capital, we can also downscale other available macroeconomic indicators as estimates of asset value. The indicator can be set via the parameter `fin_mode`, either to 'pc', 'pop', 'gdp', 'income_group', 'nfw', 'tw', 'norm', or 'none'. See descriptions of each alternative above in the introduction.

We can also change the resolution via `res_arcsec` and the `exponents`.

The default resolution is 30 arcsec ≈ 1 km. A resolution of 3600 arcsec = 1 degree corresponds to roughly 110 km close to the equator.

from_population

Let's initiate an exposure instance with the financial mode "income_group" and at a resoultion of 120 arcsec (roughly 4 km).

[3]:
```
# Initiate a LitPop exposure entity for Costa Rica with varied resolution, fin_mode, and␣
↪exponents:
exp = LitPop.from_countries('Costa Rica', fin_mode='income_group', res_arcsec=120,␣
↪exponents=(1,1)) # change the parameters and see what happens...
# exp = LitPop.from_countries('Costa Rica', fin_mode='gdp', res_arcsec=90, exponents=(3,0))
↪# example of variation
```

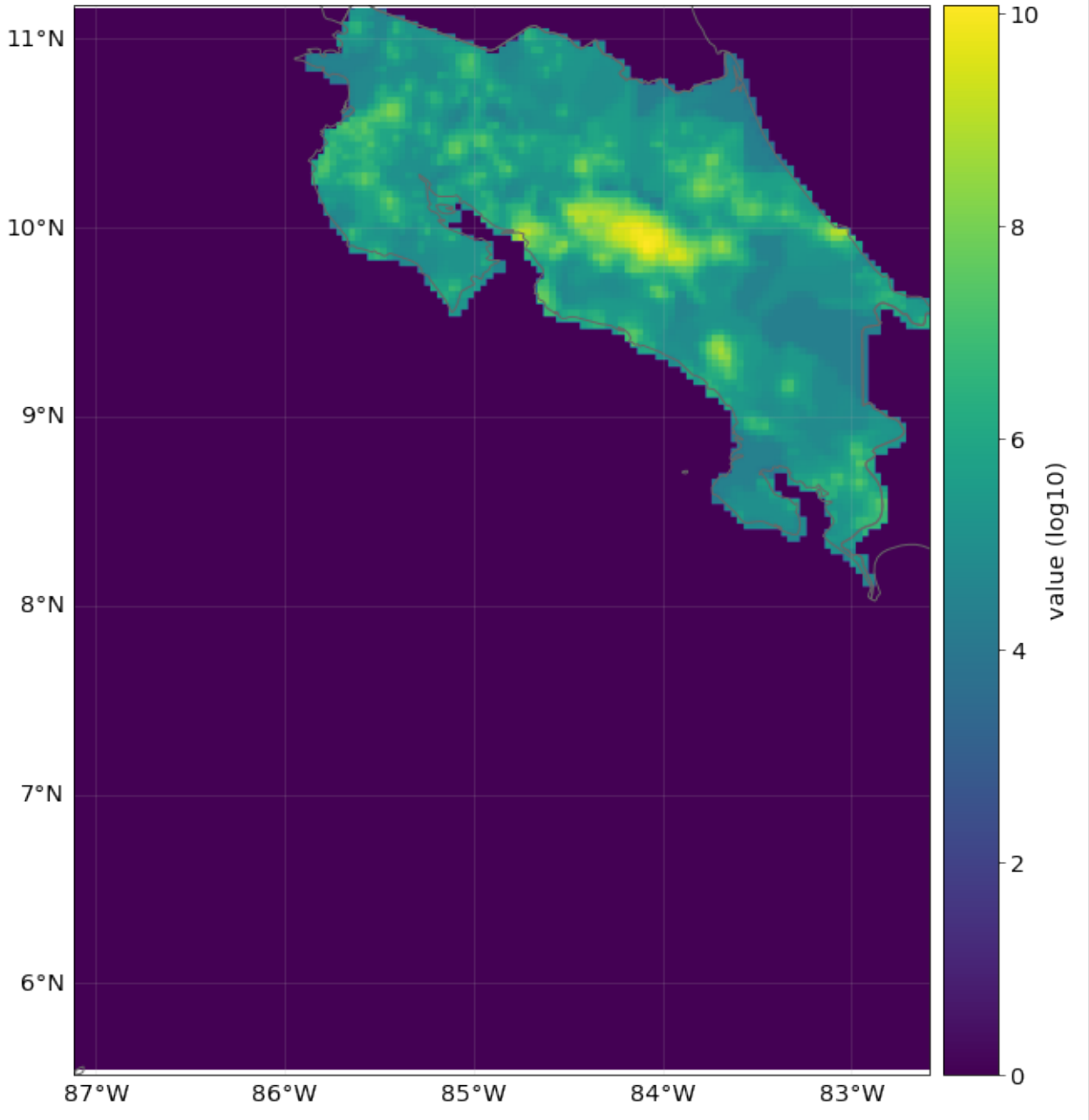
(continues on next page)

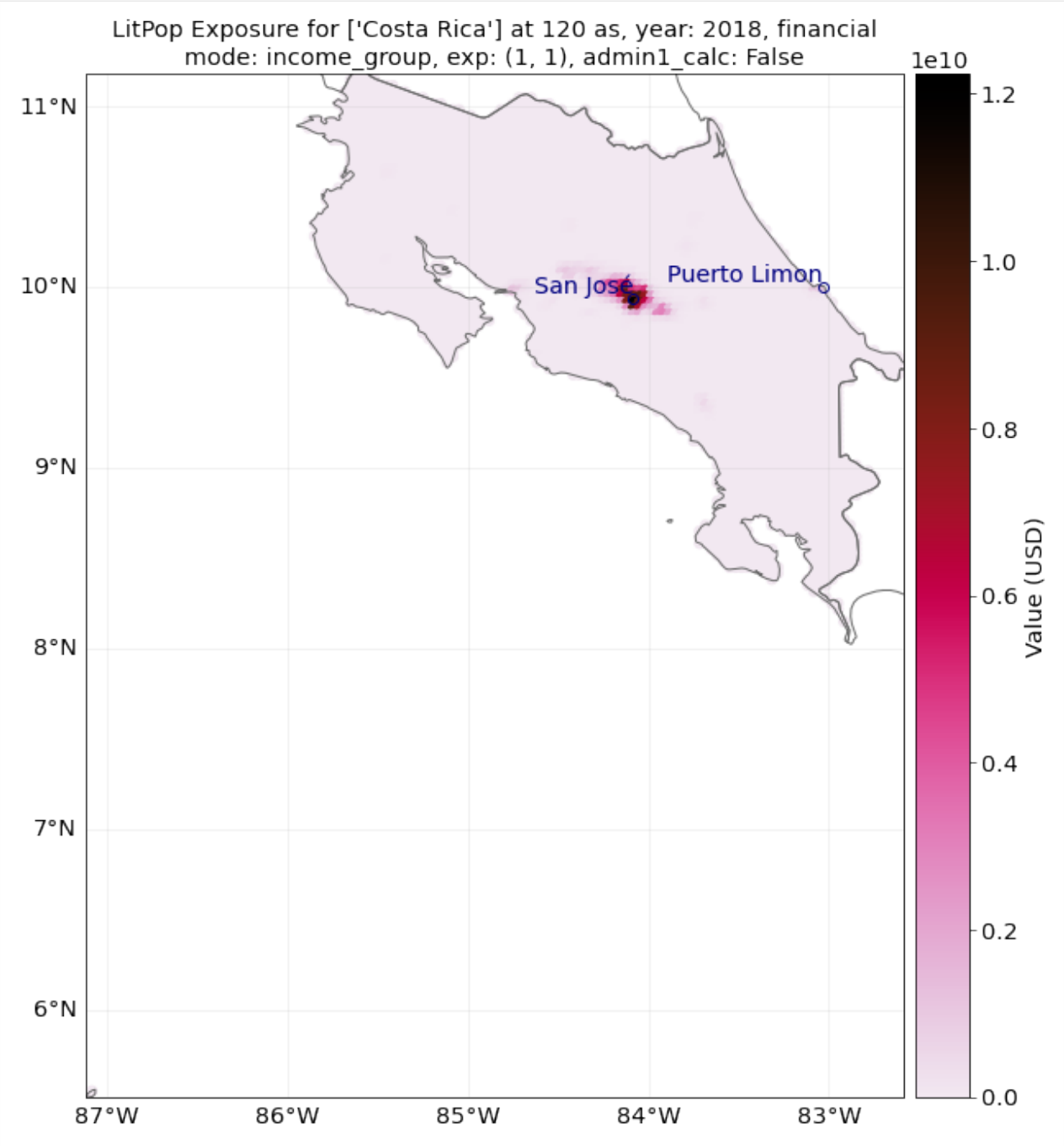
(continued from previous page)

```
exp.plot_raster() # note the log scale of the colorbar
exp.plot_scatter()
```

```
2021-10-19 17:03:26,671 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

[3]:
```
<GeoAxesSubplot:title={'center':"LitPop Exposure for ['Costa Rica'] at 120 as, year:␣
↪2018, financial\nmode: income_group, exp: (1, 1), admin1_calc: False"}>
```




LitPop Exposure for ['Costa Rica'] at 120 as, year: 2018, financial
mode: income_group, exp: (1, 1), admin1_calc: False
1e10
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Value (USD)
San Jose
Puerto Limon
11°N
10°N
9°N
8°N
7°N
6°N
87°W
86°W
85°W
84°W
83°W

Reference year

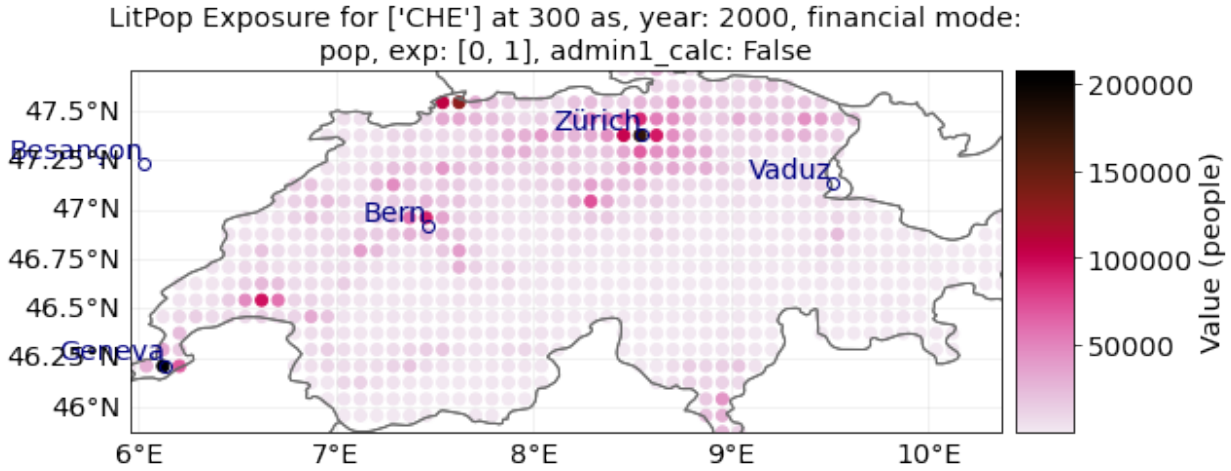
Additionally, we can change the year our exposure is supposed to represent. For this, nightlight and population data are used that are closest to the requested years. Macroeconomic indicators like produced capital are interpolated from available data or scaled proportional to GDP.

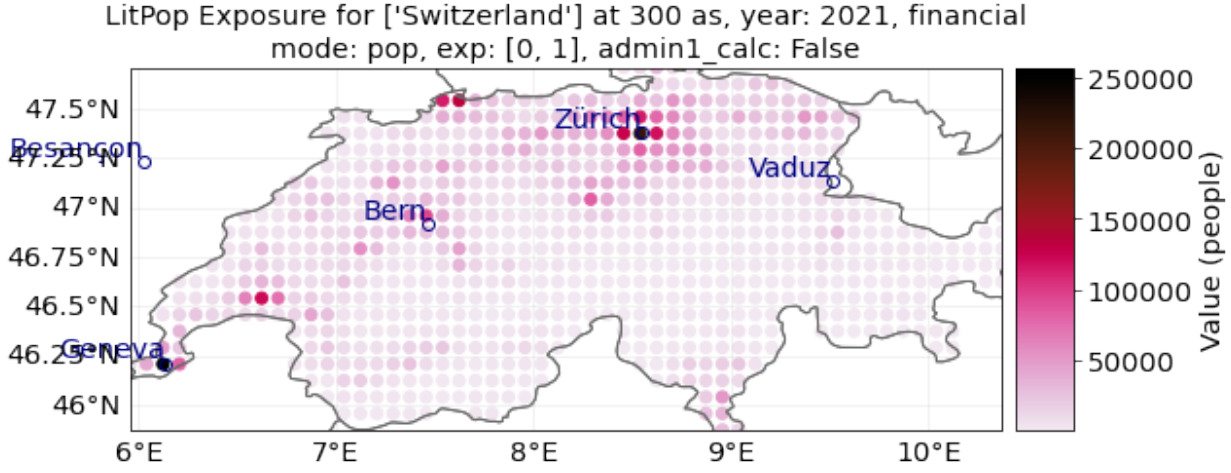
Let's load a population exposure map for Switzerland in 2000 and 2021 with a resolution of 300 arcsec:

[4]:
```
pop_2000 = LitPop.from_countries('CHE', fin_mode='pop', res_arcsec=300, exponents=(0,1),␣
↪reference_year=2000)
# Alternatively, we ca use `from_population`:
pop_2021 = LitPop.from_population(countries='Switzerland', res_arcsec=300, reference_
↪year=2021)
# Since no population data for 2021 is available, the closest data point, 2020, is used␣
↪(see LOGGER.warning)
pop_2000.plot_scatter()
pop_2021.plot_scatter()
"""Note the difference in total values on the color bar."""
```

```
2021-10-19 17:03:31,884 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2021. Using nearest available year for GPW data: 2020
```

[4]:
```
'Note the difference in total values on the color bar.'
```





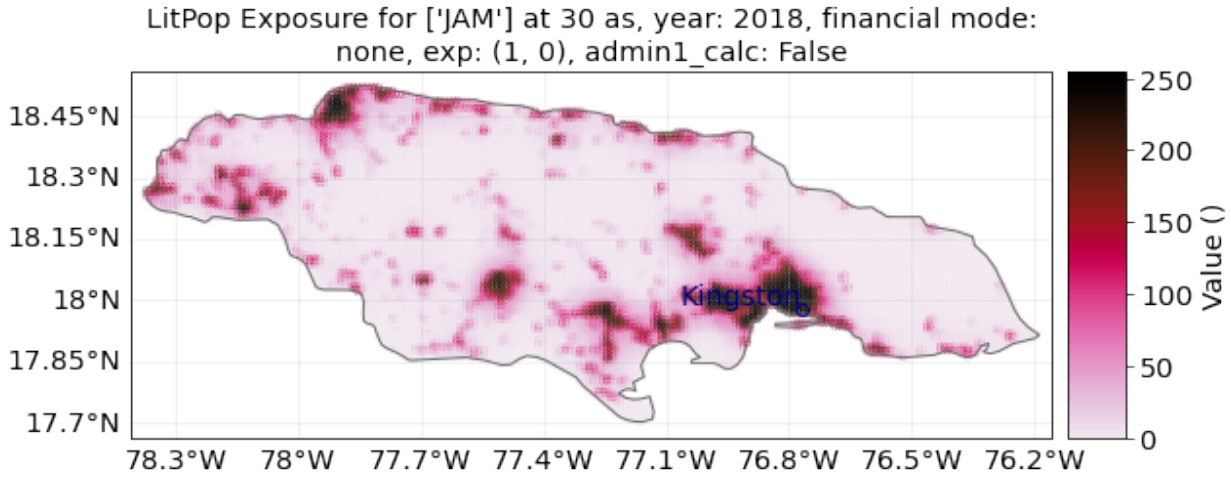
from_nightlight_intensity and from_population

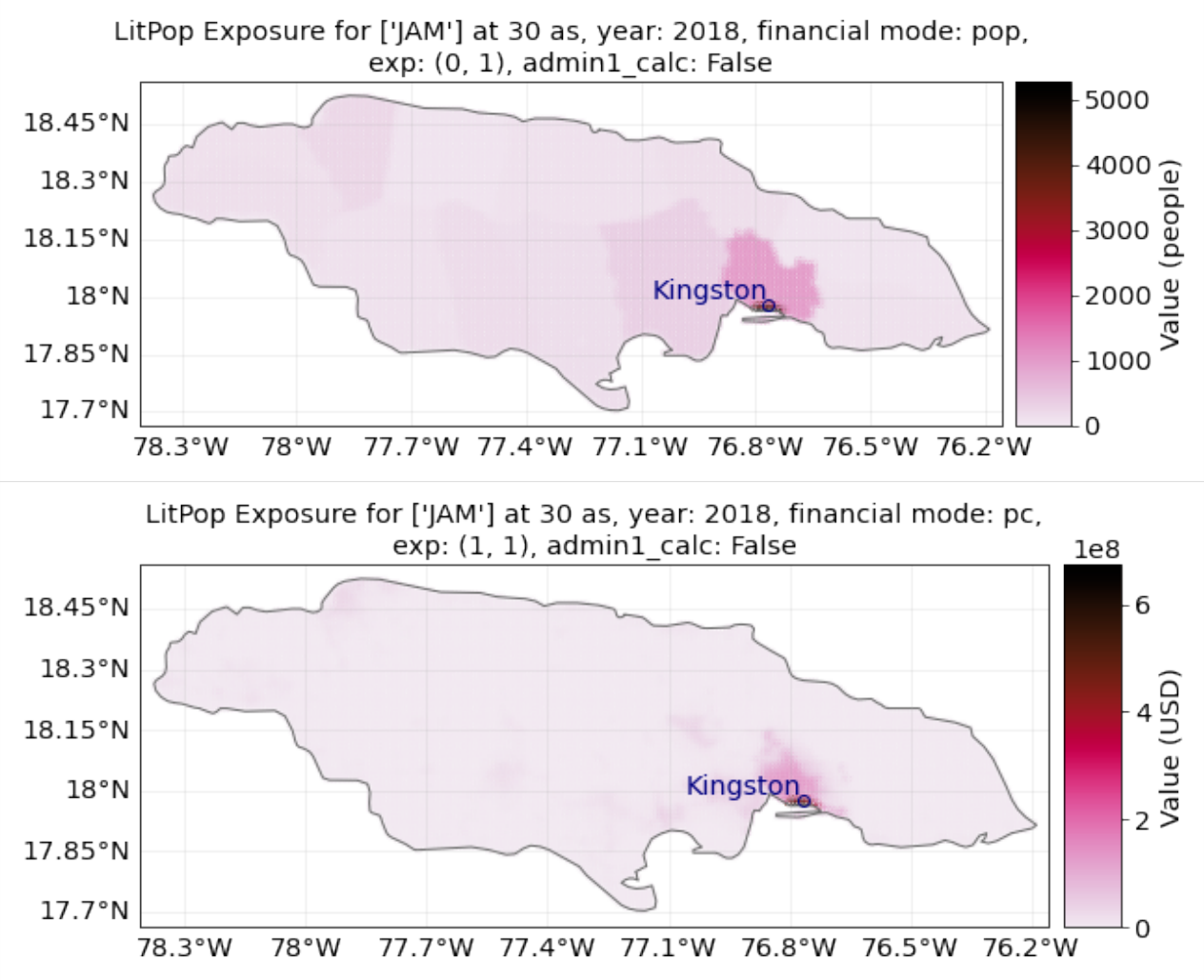
These wrapper methods can be used to produce exposures that are showing purely nightlight intensity or purely population count.

[5]:
```
res = 30
country = 'JAM' # Try different countries, i.e. 'JAM', 'CHE', 'RWA', 'MEX'
markersize = 4 # for plotting
buffer_deg=.04

exp_nightlights = LitPop.from_nightlight_intensity(countries=country, res_arcsec=res) #␣
↪nightlight intensity
exp_nightlights.plot_hexbin(linewidth=markersize, buffer=buffer_deg)
# Compare to the population map:
exp_population = LitPop().from_population(countries=country, res_arcsec=res)
exp_population.plot_hexbin(linewidth=markersize, buffer=buffer_deg)
# Compare to default LitPop exposures:
exp = LitPop.from_countries('JAM', res_arcsec=res)
exp.plot_hexbin(linewidth=markersize, buffer=buffer_deg)
```

```
2021-10-19 17:03:34,705 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2021-10-19 17:03:35,308 - climada.entity.exposures.litpop.litpop - WARNING - Note: set_
↪nightlight_intensity sets values to raw nightlight intensity, not to USD. To␣
↪disaggregate asset value proportionally to nightlights^m, call from_countries or from_
↪shape with exponents=(m,0).
2021-10-19 17:03:39,867 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2021-10-19 17:03:44,032 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

[5]:
```
<GeoAxesSubplot:title={'center':"LitPop Exposure for ['JAM'] at 30 as, year: 2018,␣
↪financial mode: pc,\nexp: (1, 1), admin1_calc: False"}>
```





For **Switzerland**, population is resolved on the 3rd administrative level, with 2538 distinct geographical units. Therefore, the purely population-based map is highly resolved.

For **Jamaica**, population is only resolved on the 1st administrative level, with only 14 distinct geographical units. Therefore, the purely population-based map shows large monotonous patches. The combination of Lit and Pop results in a concentration of asset value estimates around the capital city Kingston.

5.3.4 Init LitPop-Exposure from custom shapes

The methods `LitPop.from_shape_and_countries` and `LitPop.from_shape` initiate a LitPop-exposure instance for a given custom shape instead of a country. This can be used to initiate exposure for admin1-regions, i.e. cantons, states, districts, - but also for bounding boxes etc.

The difference between the two methods is that for `from_shape_and_countries`, the exposure for one or more whole countries is initiated first and then it is cropped to the shape. Please make sure that the shape is contained in the given countries. With `from_shape`, the shape is initiated directly which is much more resource efficient but requires a `total_value` to be provided by the user.

A population exposure for a custom shape can be initiated directly via `from_population` without providing `total_value`.

Using `LitPop.from_shape_and_countries` and `LitPop.from_shape` we initiate LitPop exposures for Florida:

[6]:
```
import time
import climada.util.coordinates as u_coord
import climada.entity.exposures.litpop as lp

country_iso3a = 'USA'
state_name = 'Florida'
reslution_arcsec = 600
"""First, we need to get the shape of Florida:"""
admin1_info, admin1_shapes = u_coord.get_admin1_info(country_iso3a)
admin1_info = admin1_info[country_iso3a]
admin1_shapes = admin1_shapes[country_iso3a]
admin1_names = [record['name'] for record in admin1_info]
print(admin1_names)
for idx, name in enumerate(admin1_names):
    if admin1_names[idx]==state_name:
        break
print('Florida index: ' + str(idx))

"""Secondly, we estimate the `total_value`"""
# `total_value` required user input for `from_shape`, here we assume 5% of total value of␣
↪the whole USA:
total_value = 0.05 * lp._get_total_value_per_country(country_iso3a, 'pc', 2020)

"""Then, we can initiate the exposures for Florida:"""
start = time.process_time()
exp = LitPop.from_shape(admin1_shapes[idx], total_value, res_arcsec=600, reference_
↪year=2020)
print(f'\n Runtime `from_shape` : {time.process_time() - start:1.2f} sec.\n')
exp.plot_scatter(vmin=100, buffer=.5)
```

```
['Minnesota', 'Washington', 'Idaho', 'Montana', 'North Dakota', 'Michigan', 'Maine',
↪'Ohio', 'New Hampshire', 'New York', 'Vermont', 'Pennsylvania', 'Arizona', 'California
↪', 'New Mexico', 'Texas', 'Alaska', 'Louisiana', 'Mississippi', 'Alabama', 'Florida',
↪'Georgia', 'South Carolina', 'North Carolina', 'Virginia', 'District of Columbia',
↪'Maryland', 'Delaware', 'New Jersey', 'Connecticut', 'Rhode Island', 'Massachusetts',
↪'Oregon', 'Hawaii', 'Utah', 'Wyoming', 'Nevada', 'Colorado', 'South Dakota', 'Nebraska
↪', 'Kansas', 'Oklahoma', 'Iowa', 'Missouri', 'Wisconsin', 'Illinois', 'Kentucky',
↪'Arkansas', 'Tennessee', 'West Virginia', 'Indiana']
Florida index: 20
```

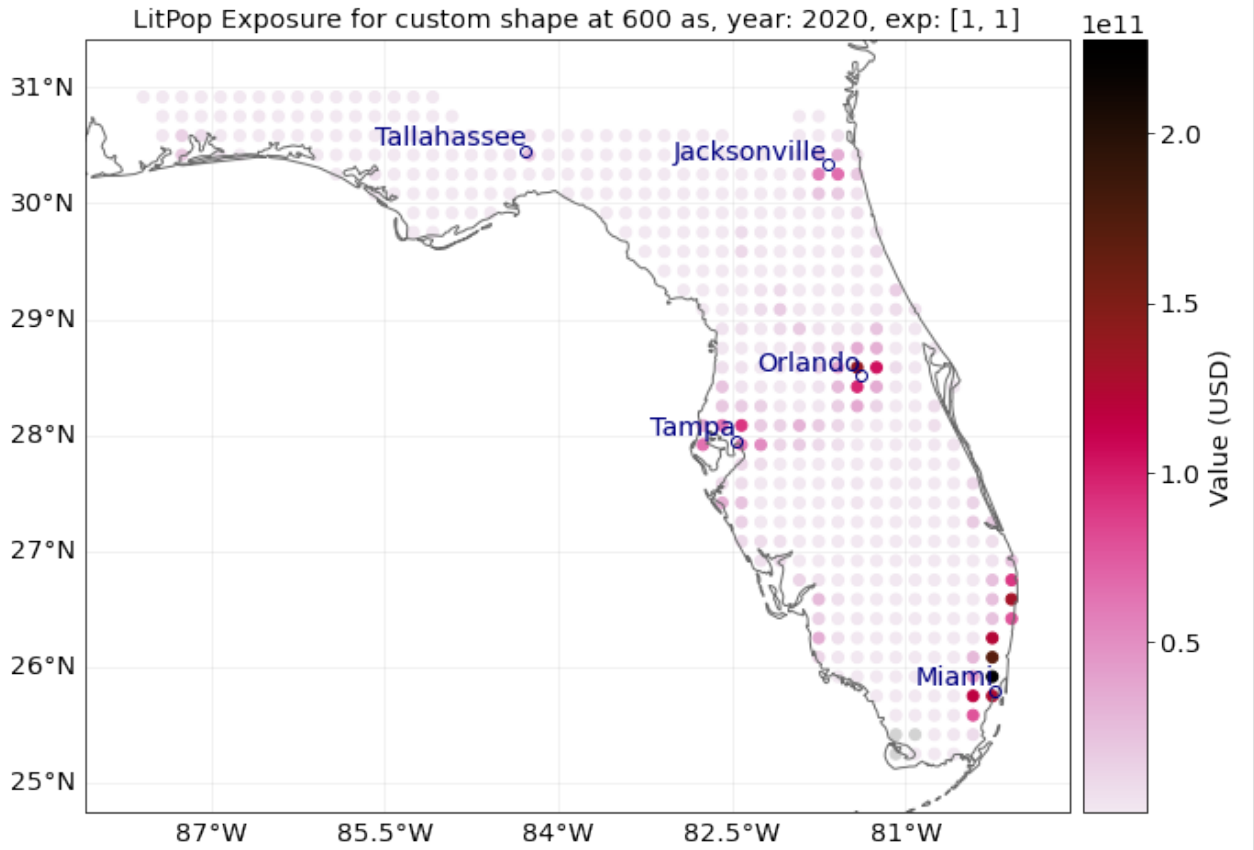
```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
```

```
 Runtime `from_shape` : 9.01 sec.
```

(continues on next page)

(continued from previous page)

[6]:
```
<GeoAxesSubplot:title={'center':'LitPop Exposure for custom shape at 600 as, year: 2020,␣
↪exp: [1, 1]'}>
```



[7]:
```
# `from_shape_and_countries` does not require `total_value`, but is slower to compute␣
↪than `from_shape`,
# because first, the exposure for the whole USA is initiated:
start = time.process_time()
exp = LitPop.from_shape_and_countries(admin1_shapes[idx], country_iso3a, res_arcsec=600,␣
↪reference_year=2020)
print(f'\n Runtime `from_shape_and_countries` : {time.process_time() - start:1.2f} sec.\n
↪')
exp.plot_scatter(vmin=100, buffer=.5)
"""Note the differences in computational speed and total value between the two approaches
↪"""
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.
```

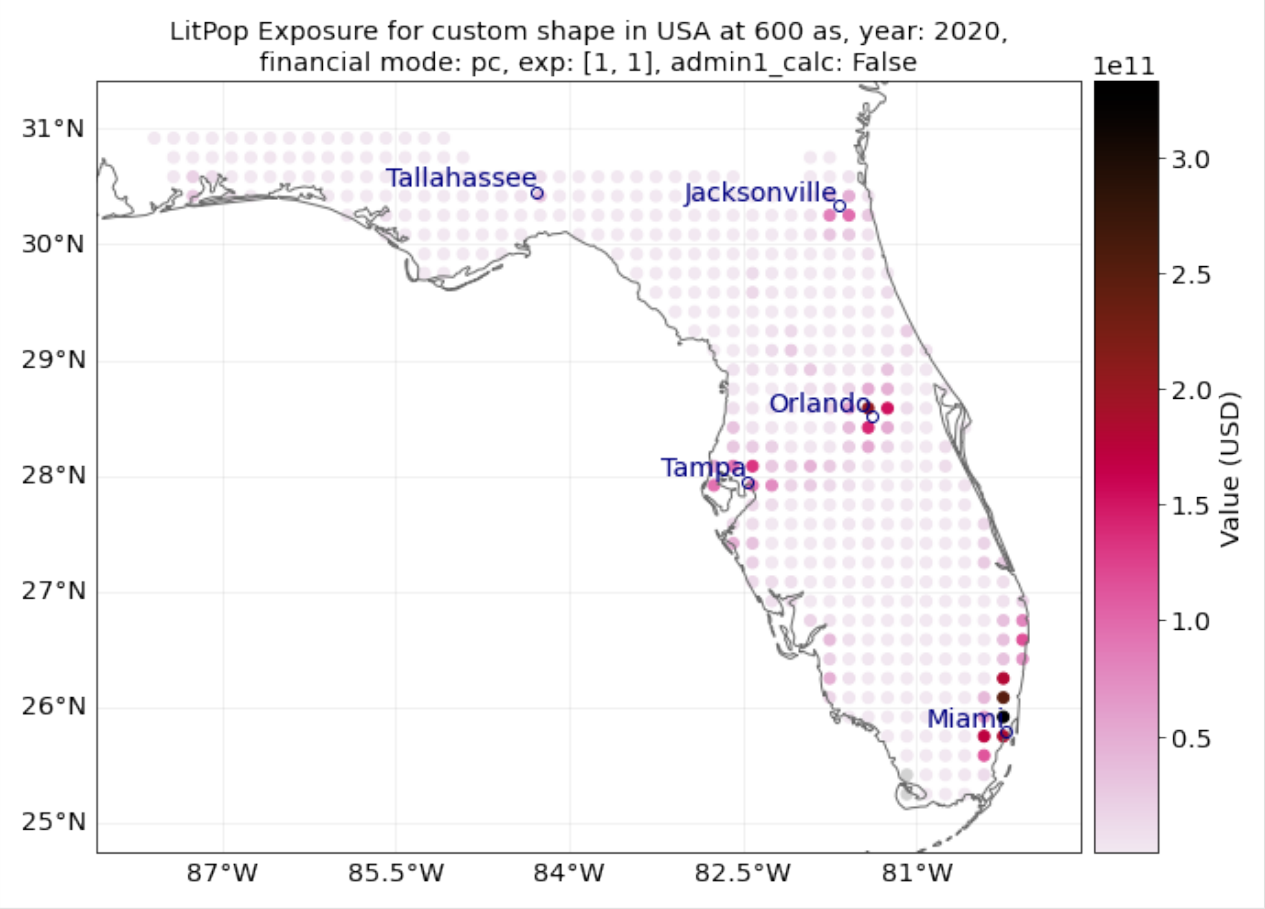
(continues on next page)

(continued from previous page)

```
countries['area'] = countries.geometry.area
```

```
Runtime `from_shape_and_countries` : 24.49 sec.
```

[7]:
```
'Note the differences in computational speed and total value between the two approaches'
```



You can also define your own shape as a Polygon:

[8]:
```
import time
from shapely.geometry import Polygon

"""initiate LitPop exposures for a geographical box around the city of Zurich:"""
bounds = (8.41, 47.25, 8.70, 47.47) # (min_lon, max_lon, min_lat, max_lat)
total_value=1000 # required user input for `from_shape`, here we just assume USD 1000 of␣
↪total value
shape = Polygon([
                (bounds[0], bounds[3]),
                (bounds[2], bounds[3]),
                (bounds[2], bounds[1]),
                (bounds[0], bounds[1])
                ])
import time
```

(continues on next page)

(continued from previous page)

```
start = time.process_time()
exp = LitPop.from_shape(shape, total_value)
print(f'\n Runtime `from_shape` : {time.process_time() - start:1.2f} sec.\n')
exp.plot_scatter()
# `from_shape_and_countries` does not require `total_value`, but is slower to compute:
start = time.process_time()
exp = LitPop.from_shape_and_countries(shape, 'Switzerland')
print(f'\n Runtime `from_shape_and_countries` : {time.process_time() - start:1.2f} sec.\n
↪')
exp.plot_scatter()
"""Note the difference in total value between the two exposure sets!"""

"""For comparison, initiate population exposure for a geographical box around the city␣
↪of Zurich:"""
start = time.process_time()
exp_pop = LitPop.from_population(shape=shape)
print(f'\n Runtime `from_population` : {time.process_time() - start:1.2f} sec.\n')
exp_pop.plot_scatter()

"""Population exposure for a custom shape can be initiated directly via `set_population`␣
↪without providing `total_value`"""
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
```

```
 Runtime `from_shape` : 0.51 sec.

2021-10-19 17:04:24,606 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020

 Runtime `from_shape_and_countries` : 3.18 sec.
```

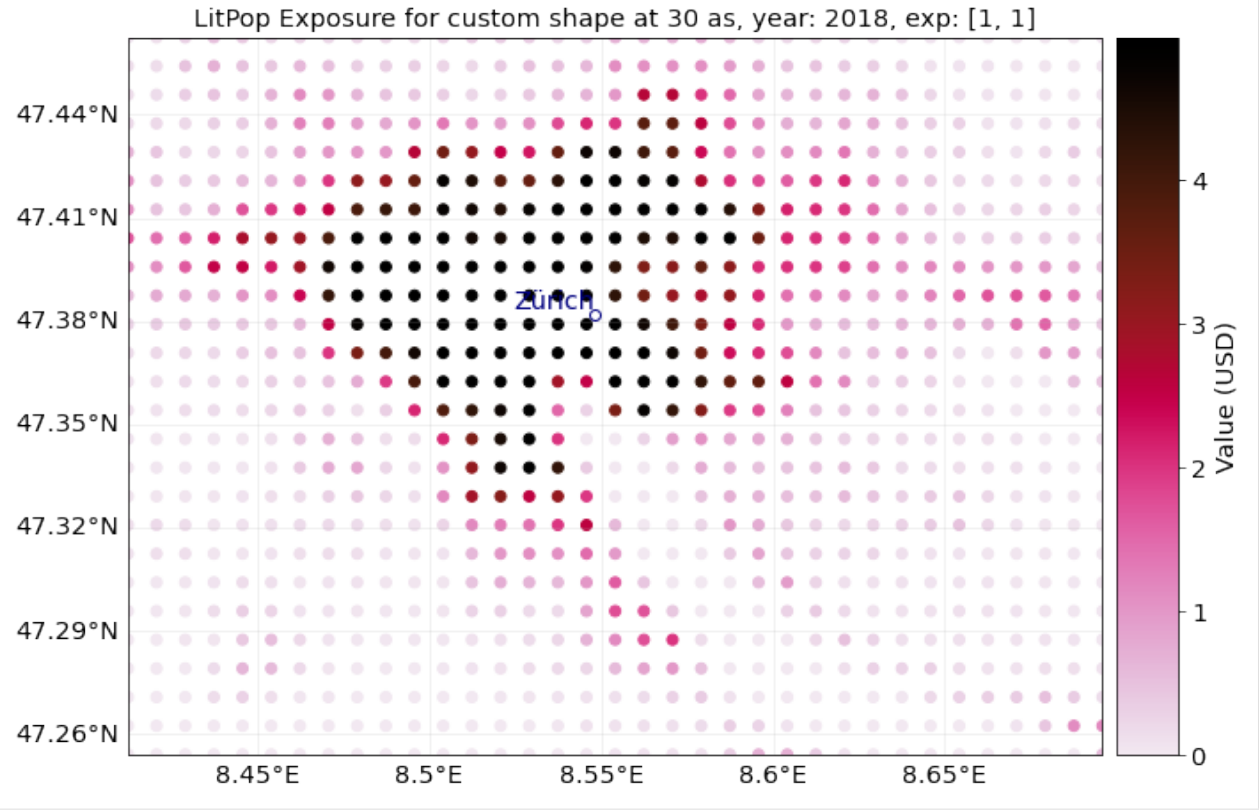
```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
```

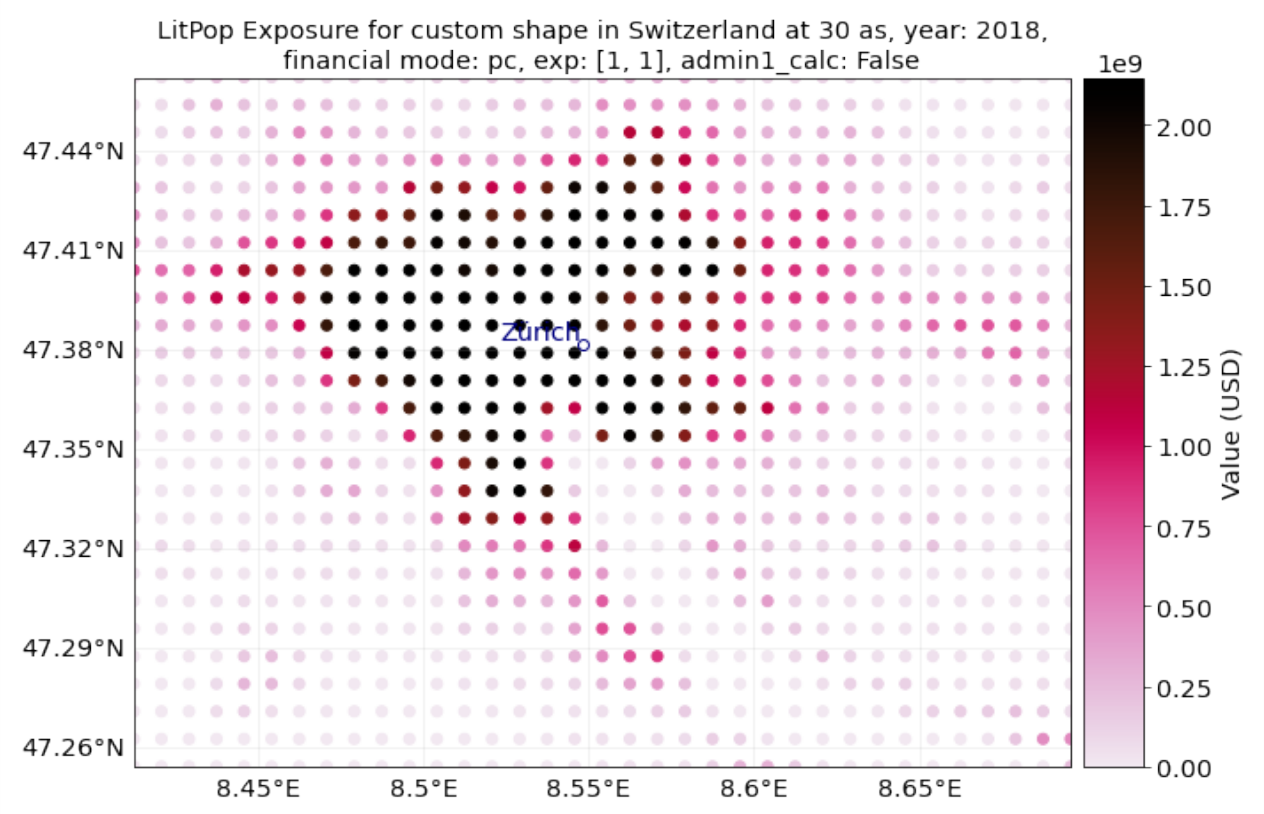
(continues on next page)

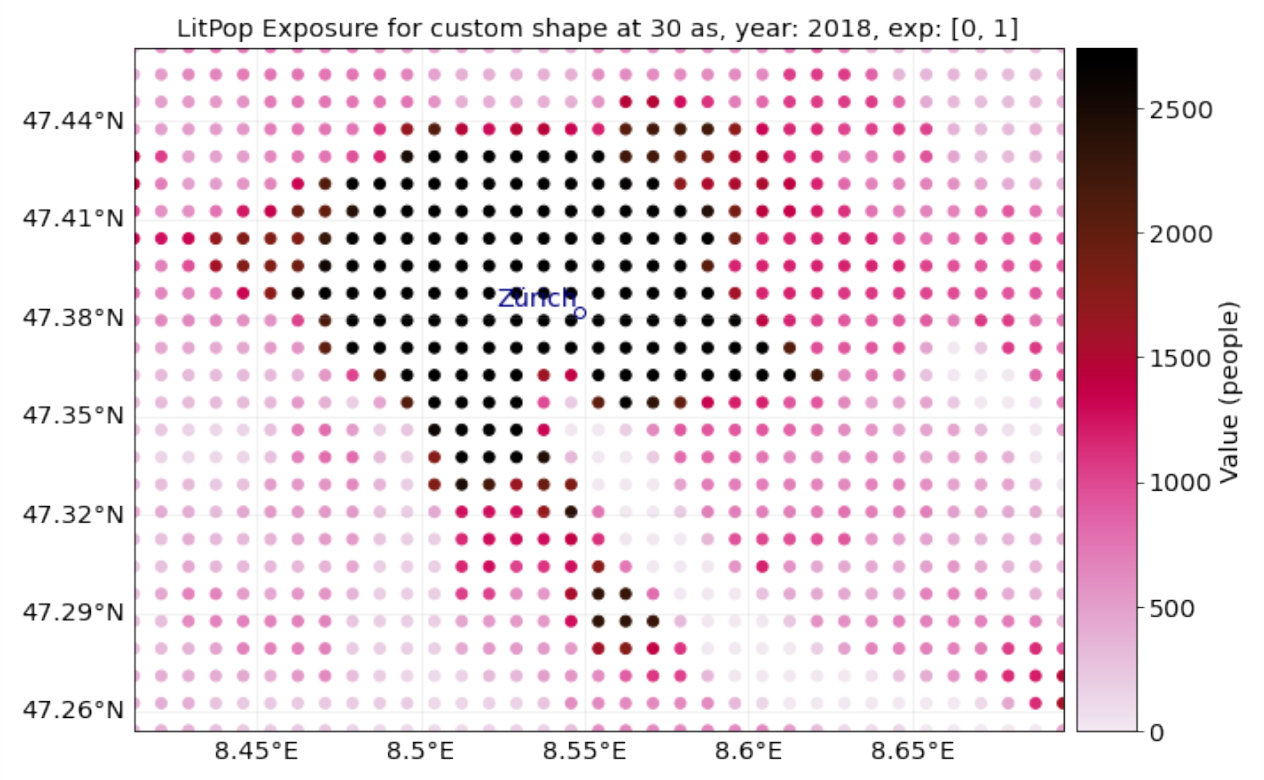
(continued from previous page)

```
Runtime `from_population` : 0.75 sec.
```

```
[8]: 'Population exposure for a custom shape can be initiated directly via `set_population`␣
↪without providing `total_value`'
```




LitPop Exposure for custom shape in Switzerland at 30 as, year: 2018,
financial mode: pc, exp: [1, 1], admin1_calc: False
1e9
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Value (USD)
47.44°N
47.41°N
47.38°N
47.35°N
47.32°N
47.29°N
47.26°N
Zurich
8.45°E
8.5°E
8.55°E
8.6°E
8.65°E


LitPop Exposure for custom shape at 30 as, year: 2018, exp: [0, 1]
2500
2000
1500
1000
500
0
Value (people)
47.44°N
47.41°N
47.38°N
47.35°N
47.32°N
47.29°N
47.26°N
Zurich
8.45°E
8.5°E
8.55°E
8.6°E
8.65°E

Sub-national (admin-1) GDP as intermediate downscaling layer

In order to improve downscaling for countries with large regional differences within, a subnational breakdown of GDP can be used as an intermediate downscaling layer wherever available.

The sub-national (admin-1) GDP-breakdown needs to be added manually as a ".xls"-file to the folder `data/system/GSDP` in the CLIMADA-directory. Currently, such data is provided for more than 10 countries, including USA, India, and China.

The xls-file requires at least the following columns (with names specified in row 1): - `State_Province`: Names of admin-1 regions, i.e. states, cantons, provinces. Names need to match the naming of admin-1 shapes in the data used by the python package `cartopy.io` (c.f. `shapereader.natural_earth(name='admin_1_states_provinces')`) - `GSDP_ref`: value of sub-national GDP to be used (absolute or relative values) - `Postal`, optional: Alternative identifier of region, if names do not match wioth cartopy. Needs to correspond to the Postal-identifiers used in the shapereader of `cartopy.io`.

Please note that while admin1-GDP will per definition improve the downscaling of *GDP*, it might not neccessarily improve the downscaling quality for other asset bases like produced capital (pc). #### How To: The intermediadte downscaling layer can be activated with the parameter `admin1_calc=True`.

[9]:
```
# Initiate GDP-Entity for Switzerland, with and without admin1_calc:

ent_adm0 = LitPop.from_countries('CHE', res_arcsec=120, fin_mode='gdp', admin1_
↪calc=False)
ent_adm0.set_geometry_points()

ent_adm1 = LitPop.from_countries('CHE', res_arcsec=120, fin_mode='gdp', admin1_calc=True)

ent_adm0.check()
ent_adm1.check()
print('Done.')
```

```
2021-10-19 17:04:31,363 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
```

(continues on next page)

(continued from previous page)

```
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.
```

(continues on next page)

(continued from previous page)

```
  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
```

(continues on next page)

(continued from previous page)

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
```

(continues on next page)

(continued from previous page)

```
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.
```

(continues on next page)

(continued from previous page)

```
  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

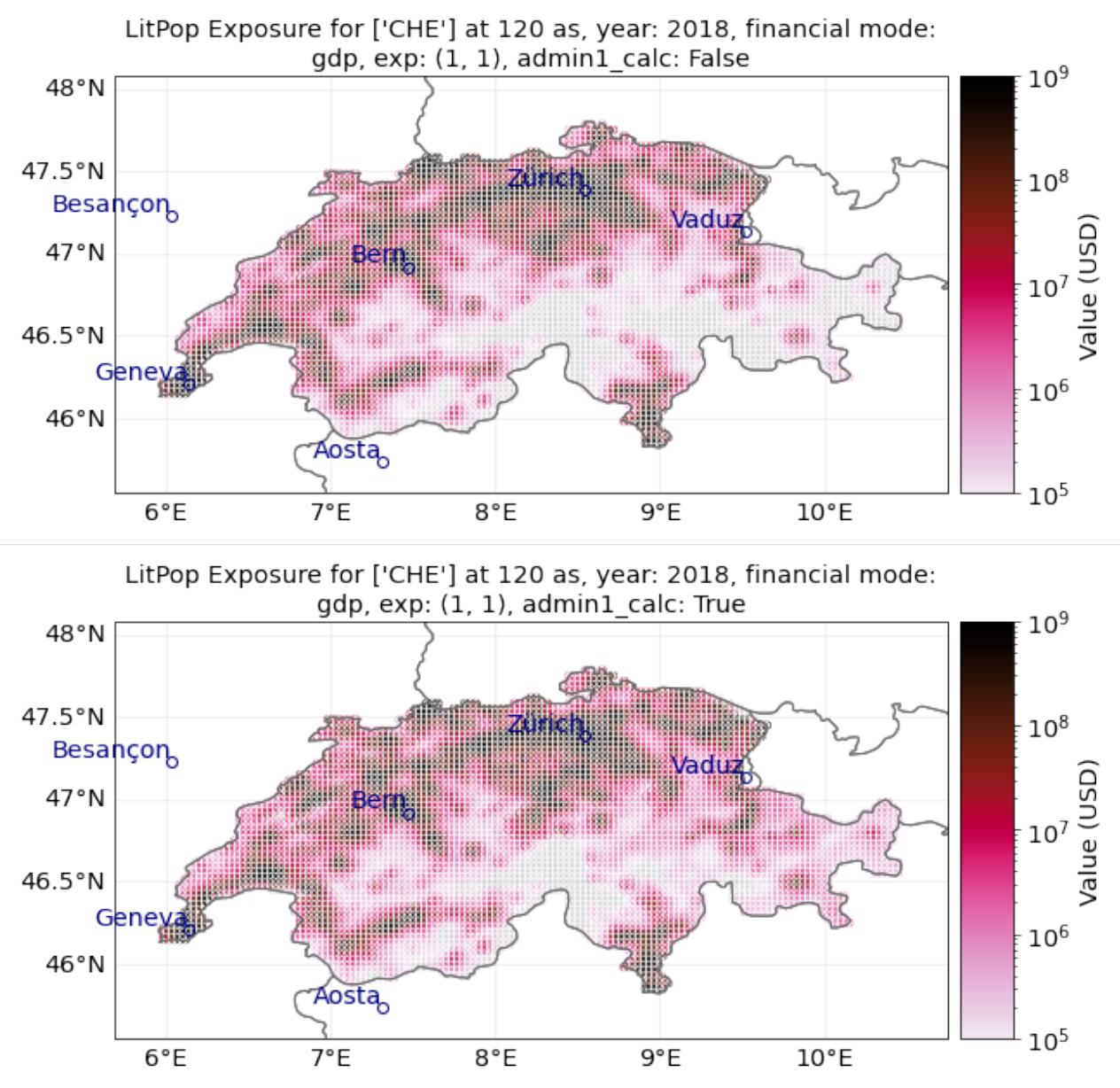
```
Done.
```

```
$CLIMADA_SRC/climada/util/coordinates.py:1129: UserWarning: Geometry is in a geographic␣
↪CRS. Results from 'area' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  countries['area'] = countries.geometry.area
```

[10]:
```
# Plotting:
from matplotlib import colors
norm=colors.LogNorm(vmin=1e5, vmax=1e9) # setting range for the log-normal scale
markersize = 5
ent_adm0.plot_hexbin(buffer=.3, norm=norm, linewidth=markersize)
ent_adm1.plot_hexbin(buffer=.3, norm=norm, linewidth=markersize)

print('admin-0: First figure')
print('admin-1: Second figure')
'"Do you spot the small differences in Graubünden (eastern Switzerland)?"'
```

```
admin-0: First figure
admin-1: Second figure
```

[10]:
```
'Do you spot the small differences in Graubünden (eastern Switzerland)?'
```



5.4 Impact Functions

5.4.1 What is an impact function?

An impact function relates the percentage of damage in the exposurethe to the hazard intensity, also commonly referred to as a "vulnerability curves" in the modelling community. Every hazard and exposure types are characterized by an impact function.

5.4.2 What is the difference between ImpactFunc and ImpactFuncSet?

An ImpactFunc is a class for a single impact function. E.g. a function that relates the percentage of damage of a reinforced concrete building (exposure) to the wind speed of a tropical cyclone (hazard intensity).

An ImpactFuncSet class is a container that contains multiple ImpactFunc. For instance, there are 100 ImpactFunc represent 100 types of buildings exposed to tropical cyclone's wind damage. These 100 ImpactFunc are all gathered in an ImpactFuncSet.

5.4.3 What does an ImpactFunc look like in CLIMADA?

The ImpacrFunc class requires users to define the following attributes.

Mandatory attributes	Data Type	Description
haz_type	(str)	Hazard type acronym (e.g. 'TC')
id	(int or str)	Unique id of the impact function. Exposures of the same type will refer to the same impact function id
name	(str)	Name of the impact function
intensity	(np.array)	Intensity values
intensity_unit	(str)	Unit of the intensity
mdd	(np.array)	Mean damage (impact) degree for each intensity (numbers in [0,1])
paa	(np.array)	Percentage of affected assets (exposures) for each intensity (numbers in [0,1])

Users may use ImpactFunc.check() to check that the attributes have been set correctly. The mean damage ratio mdr (mdr=mdd*paa) is calculated by the method ImpactFunc.calc_mdr().

5.4.4 What does an ImpactFuncSet look like in CLIMADA?

The ImpactFuncSet class contains all the ImpactFunc classes. Users are not required to define any attributes in ImpactFuncSet.

To add an ImpactFunc into an ImpactFuncSet, simply use the method ImpactFuncSet.append(ImpactFunc). If the users only has one impact function, they should generate an ImpactFuncSet that contains one impact function. ImpactFuncSet is to be used in the *impact calculation*.

Tag stores information about the data. E.g. the original file name of the impact functions and descriptions.

Attributes	Data Type	Description
tag	Tag	Information about the source data
_data	(dict)	Contains ImpactFunc classes. Not suppossed to be directly accessed. Use the class methods instead.

5.4.5 Structure of the tutorial

Part 1: Defining `ImpactFunc` from your own data

Part 2: Loading `ImpactFunc` from CLIMADA in-built impact functions

Part 3: Add `ImpactFunc` into the container `ImpactFuncSet`

Part 4: Read and write `ImpactFuncSet` into Excel sheets

Part 5: Loading `ImpactFuncSet` from CLIMADA in-built impact functions

Part 1: Defining `ImpactFunc` from your own data

The essential attributes are listed in the table above. The following example shows you how to define an `ImpactFunc` from scratch, and using the method `ImpactFunc.calc_mdr()` to calculate the mean damage ratio.

5.4.6 Generate a dummy impact function from scratch.

Here we generate an impact function with random dummy data for illustrative reasons. Assuming this impact function is a function that relates building damage to tropical cyclone (TC) wind, with an arbitrary id 3.

[1]:
```
import numpy as np
from climada.entity import ImpactFunc

# We initialise a dummy ImpactFunc for tropical cyclone wind damage to building.
# Giving the ImpactFunc an arbitrary id 3.
imp_fun = ImpactFunc()
imp_fun.haz_type = 'TC'
imp_fun.id = 3
imp_fun.name = 'TC building damage'
# provide unit of the hazard intensity
imp_fun.intensity_unit = 'm/s'
# provide values for the hazard intensity, mdd, and paa
imp_fun.intensity = np.linspace(0, 100, num=15)
imp_fun.mdd = np.concatenate((np.array([0]),np.sort(np.random.rand(14))), axis=0)
imp_fun.paa = np.concatenate((np.array([0]),np.sort(np.random.rand(14))), axis=0)
# check if the all the attributes are set correctly
imp_fun.check()
```

[2]:
```
# Calculate the mdr at hazard intensity 18.7 m/s
print('Mean damage ratio at intensity 18.7 m/s: ', imp_fun.calc_mdr(18.7))
```

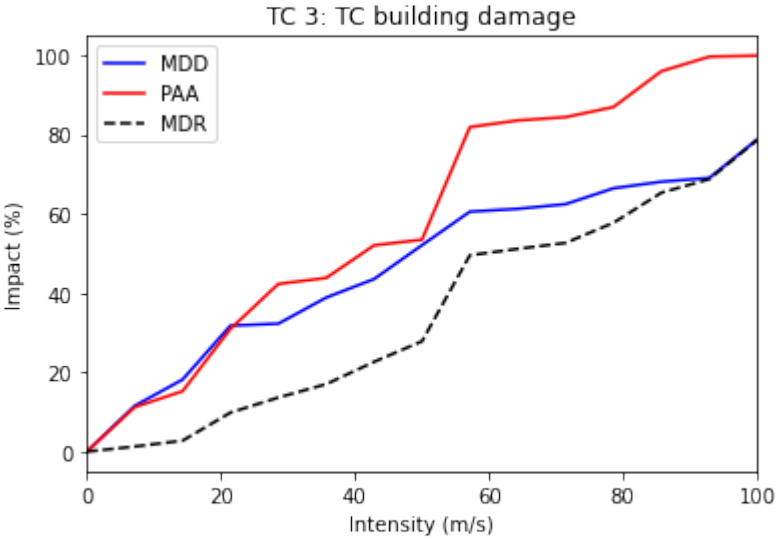
```
Mean damage ratio at intensity 18.7 m/s:  0.02395797020194174
```

5.4.7 Visualise the Impact function

The method `plot()` uses the matplotlib's axes plot function to visualise the impact function. It returns a figure and axes, which can be modified by users.

[3]:
```
# plot impact function
imp_fun.plot()
```

[3]:
```
<AxesSubplot:title={'center':'TC 3: TC building damage'}, xlabel='Intensity (m/s)',␣
↪ylabel='Impact (%)'>
```



Part 2: Loading impact functions from CLIMADA in-built impact functions

In CLIMADA there is several defined impact functions that users can directly load and use them. However, users should be aware of the applications of the impact functions to types of assets, reading the background references of the impact functions are strongly recommended. Currently available perils include tropical cyclones, river floods, European windstorm, crop yield, and drought. Continuous updates of perils are available. Here we use the impact function of tropical cyclones as an example.

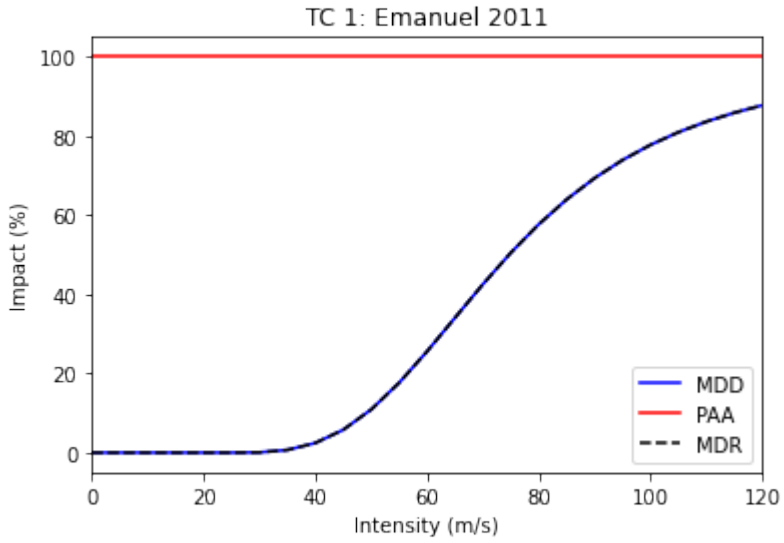
5.4.8 Loading CLIMADA in-built impact function for tropical cyclones

`ImpfTropCyclone` is a derivated class of `ImpactFunc`. This in-built impact function estimates the insured property damages by tropical cyclone wind in USA, forllowing the reference paper Emanuel (2011).

To generate the this impact function, method `set_emanual_usa()` is used.

[4]:
```
from climada.entity import ImpfTropCyclone

# Here we generate the impact function for TC damage using the formula of Emanuel 2011
impFunc_emanuel_usa = ImpfTropCyclone.from_emanuel_usa()
# plot the impact function
impFunc_emanuel_usa.plot()
```

[4]:
```
<AxesSubplot:title={'center':'TC 1: Emanuel 2011'}, xlabel='Intensity (m/s)', ylabel=
↪'Impact (%)'>
```



Part 3: Add `ImpactFunc` into the container `ImpactFuncSet`

`ImpactFuncSet` is a container of multiple `ImpactFunc`, it is part of the arguments in `impact.calc()` (see *the impact tutorial*).

Here we generate 2 aribtrary impact funstions and add them into an `ImpactFuncSet` class. To add them into the container, simply use the method `ImpactFuncSet.append(ImpactFunc)`.

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from climada.entity import ImpactFunc, ImpactFuncSet

# generate the 1st arbitrary impact function
imp_fun_1 = ImpactFunc()
imp_fun_1.haz_type = 'TC'
imp_fun_1.id = 1
imp_fun_1.name = 'TC Default Damage Function'
imp_fun_1.intensity_unit = 'm/s'
imp_fun_1.intensity = np.linspace(0, 100, num=10)
imp_fun_1.mdd = np.concatenate((np.array([0]),np.sort(np.random.rand(9))), axis=0)
imp_fun_1.paa = np.concatenate((np.array([0]),np.sort(np.random.rand(9))), axis=0)
imp_fun_1.check()

# generate the 2nd arbitrary impact function
imp_fun_3 = ImpactFunc()
imp_fun_3.haz_type = 'TC'
imp_fun_3.id = 3
imp_fun_3.name = 'TC Building Damage'
imp_fun_3.intensity_unit = 'm/s'
imp_fun_3.intensity = np.linspace(0, 100, num=15)
imp_fun_3.mdd = np.concatenate((np.array([0]),np.sort(np.random.rand(14))), axis=0)
imp_fun_3.paa = np.concatenate((np.array([0]),np.sort(np.random.rand(14))), axis=0)
imp_fun_1.check()

# add the 2 impact functions into ImpactFuncSet
```

(continues on next page)

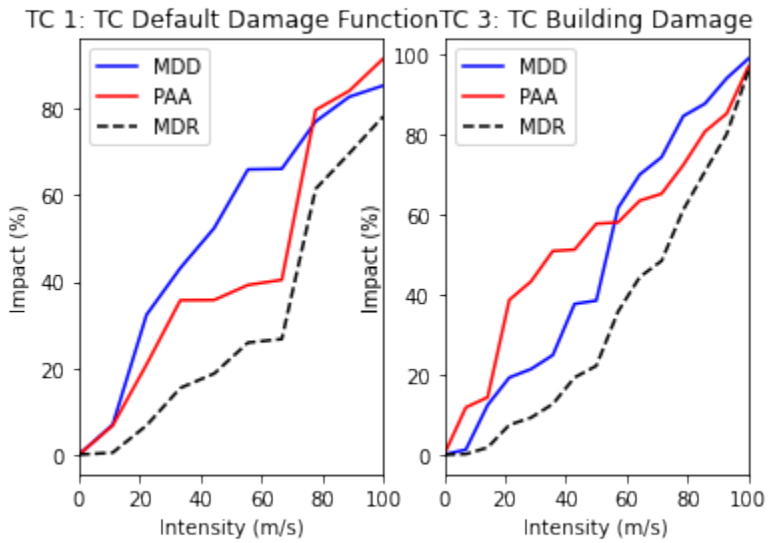
(continued from previous page)

```
imp_fun_set = ImpactFuncSet()
imp_fun_set.append(imp_fun_1)
imp_fun_set.append(imp_fun_3)
```

5.4.9 Plotting all the impact functions in an `ImpactFuncSet`

The method `plot()` in `ImpactFuncSet` also uses the the matplotlib's axes plot function to visualise the impact functions, returning a figure with all the subplots of impact functions. Users may modify these plots.

[6]:
```
# plotting all the impact functions in impf_set
axes = imp_fun_set.plot()
```

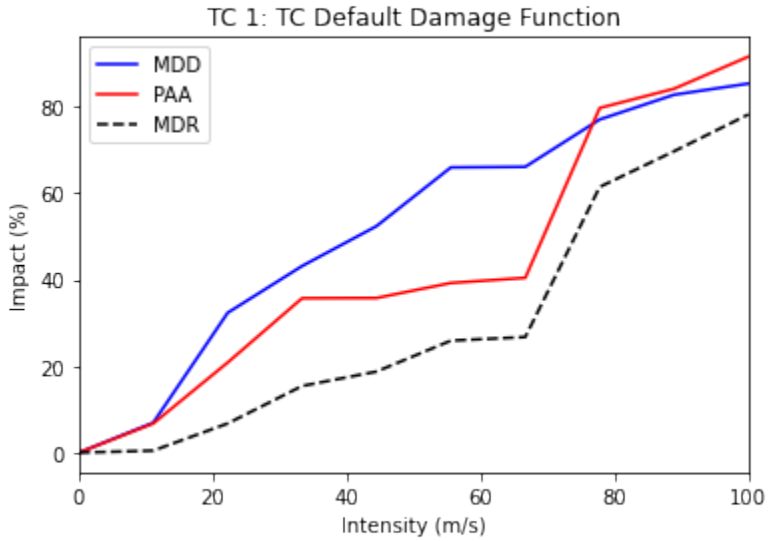


5.4.10 Retrieving an impact function from the `ImpactFuncSet`

User may want to retrive a particular impact function from `ImpactFuncSet`. Using the method `get_func(haz_type, id)`, it returns an `ImpactFunc` class of the desired impact function. Below is an example of extracting the TC impact function with id 1, and using `plot()` to visualise the function.

[7]:
```
# extract the TC impact function with id 1
impf_tc_1 = imp_fun_set.get_func('TC', 1)
# plot the impact function
impf_tc_1.plot()
```

[7]:
```
<AxesSubplot:title={'center':'TC 1: TC Default Damage Function'}, xlabel='Intensity (m/s)
↪', ylabel='Impact (%)'>
```



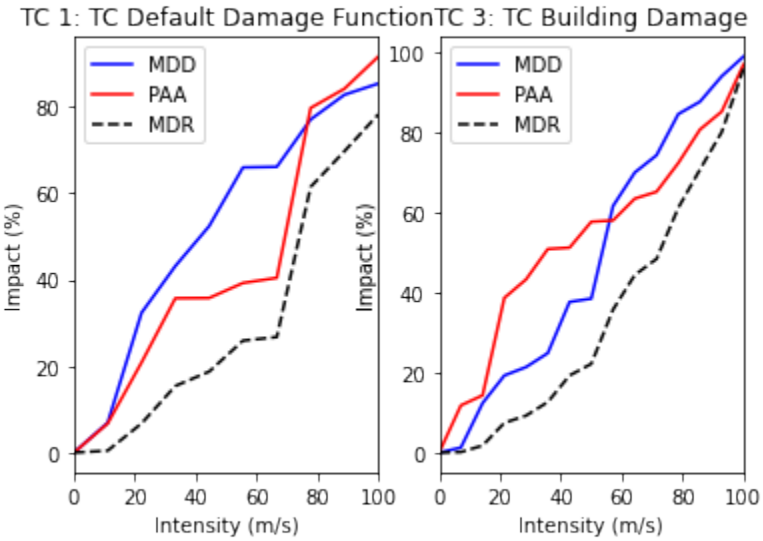
5.4.11 Removing an impact function from the ImpactFunctionSet

If there is an unwanted impact function from the `ImpactFuncSet`, we may remove it using the method `remove_func(haz_type, id)` to remove it from the set.

For example, in the previous generated impact function set `imp_fun_set` contains an unwanted TC impact function with id 3, we might thus would like to remove that from the set.

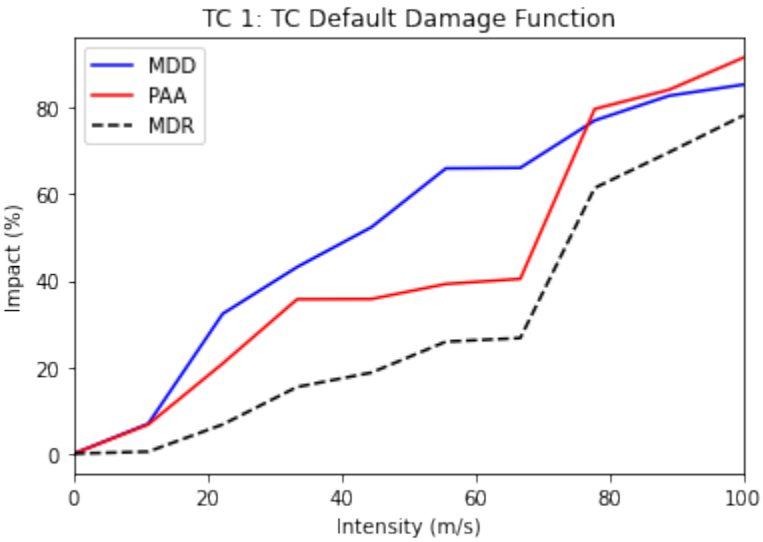
[8]:
```
# first plotting all the impact functions in the impact function set to see what is in␣
↪there:
imp_fun_set.plot()
```

[8]:
```
array([<AxesSubplot:title={'center':'TC 1: TC Default Damage Function'}, xlabel=
↪'Intensity (m/s)', ylabel='Impact (%)'>,
       <AxesSubplot:title={'center':'TC 3: TC Building Damage'}, xlabel='Intensity (m/s)
↪', ylabel='Impact (%)'>],
      dtype=object)
```



[9]:
```
# removing the TC impact function with id 3
imp_fun_set.remove_func('TC', 3)
# plot all the remaining impact functions in imp_fun_set
imp_fun_set.plot()
```

[9]:
```
<AxesSubplot:title={'center':'TC 1: TC Default Damage Function'}, xlabel='Intensity (m/s)
↪', ylabel='Impact (%)'>
```



Part 4: Read and write `ImpactFuncSet` into Excel sheets

Users may load impact functions to an `ImpactFuncSet` class from an excel sheets, or to write the `ImpactFuncSet` into an exvel. This section will give an example of how to do it.

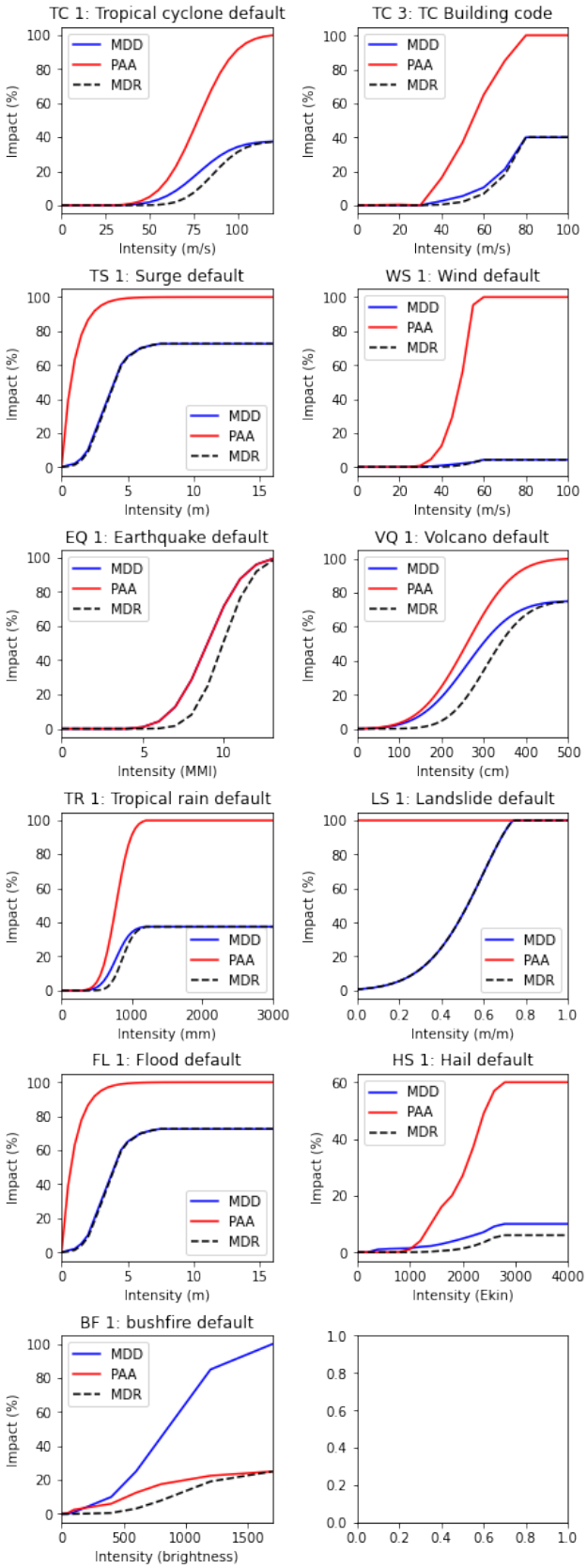
5.4.12 Reading impact functions from an Excel file

Impact functions defined in an excel file following the template provided in sheet `impact_functions` of `climada_python/data/system/entity_template.xlsx` can be ingested directly using the method `from_excel()`.

[10]:
```
from climada.entity import ImpactFuncSet
from climada.util import ENT_TEMPLATE_XLS
import matplotlib.pyplot as plt

# provide absolute path of the input excel file
file_name = ENT_TEMPLATE_XLS
# fill ImpactFuncSet from Excel file
imp_set_xlsx = ImpactFuncSet.from_excel(file_name)

# plot all the impact functions from the ImpactFuncSet
print('Read file:', imp_set_xlsx.tag.file_name)
imp_set_xlsx.plot()
# adjust the plots
plt.tight_layout()
plt.subplots_adjust(right=1., top=4., hspace=0.4, wspace=0.4)
```

```
Read file: /Users/ckropf/climada/data/entity_template.xlsx
```

```
/var/folders/r5/6rbkr9r16mg86237m11wqn500000gn/T/ipykernel_32650/833411030.py:14:␣
↪UserWarning: Tight layout not applied. tight_layout cannot make axes height small␣
↪enough to accommodate all axes decorations
  plt.tight_layout()
```


TC 1: Tropical cyclone default
TC 3: TC Building code
TS 1: Surge default
WS 1: Wind default
EQ 1: Earthquake default
VQ 1: Volcano default
TR 1: Tropical rain default
LS 1: Landslide default
FL 1: Flood default
HS 1: Hail default
BF 1: bushfire default
MDD
PAA
MDR
Impact (%)
Intensity (m/s)
Intensity (m)
Intensity (MMI)
Intensity (cm)
Intensity (mm)
Intensity (m/m)
Intensity (Ekin)
Intensity (brightness)

5.4.13 Write impact functions

Users may write the impact functions in Excel format using `write_excel()` method.

[11]:
```
from climada.entity import ImpactFuncSet
from climada.util import ENT_TEMPLATE_XLS

# provide absolute path of the output excel file
file_name_save = ENT_TEMPLATE_XLS

# write imp_set_xlsx into an excel file
imp_set_xlsx.write_excel('tutorial_impf_set.xlsx')
```

5.4.14 Alternative saveing format

Alternatively, users may also save the impact functions into pickle format, using CLIMADA in-built function `save()`.

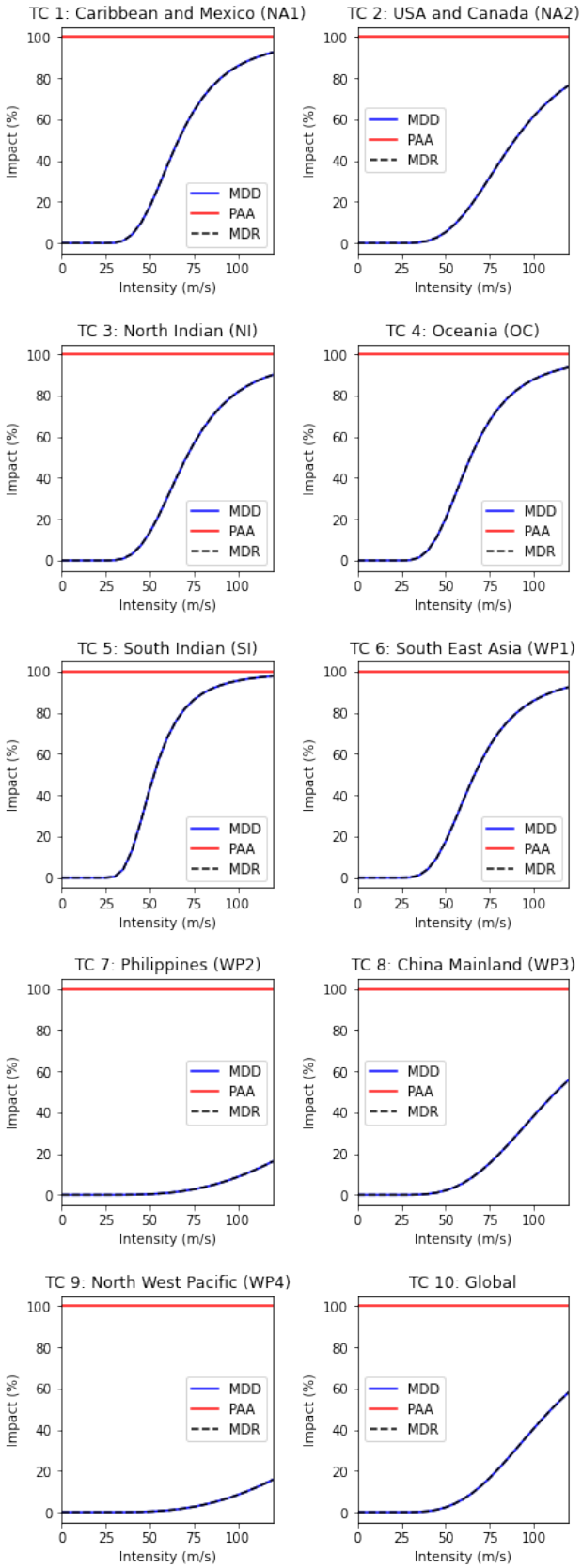
[12]:
```
from climada.util.save import save

# this generates a results folder in the current path and stores the output there
save('tutorial_impf_set.p', imp_set_xlsx)
```

Part 5: Loading `ImpactFuncSet` from CLIMADA in-built impact functions

Similar to *Part 3*, some of the impact functions are available as `ImpactFuncSet` classes. Users may load them from the CLIMADA modules.

Here we use the example of the calibrated impact functions of TC wind damages per region to property damages, following the reference Eberenz et al. (2021). Method `from_calibrated_regional_ImpfSet()` returns a set of default calibrated impact functions for TC for different regions.

[13]:
```
from climada.entity.impact_funcs.trop_cyclone import ImpfSetTropCyclone
import matplotlib.pyplot as plt

# generate the default calibrated TC impact functions for different regions
imp_fun_set_TC = ImpfSetTropCyclone.from_calibrated_regional_ImpfSet()

# plot all the impact functions
imp_fun_set_TC.plot()
# adjust the plots
plt.tight_layout()
plt.subplots_adjust(right=1., top=4., hspace=0.4, wspace=0.4)
```


TC 1: Caribbean and Mexico (NA1)
TC 2: USA and Canada (NA2)
TC 3: North Indian (NI)
TC 4: Oceania (OC)
TC 5: South Indian (SI)
TC 6: South East Asia (WP1)
TC 7: Philippines (WP2)
TC 8: China Mainland (WP3)
TC 9: North West Pacific (WP4)
TC 10: Global
Impact (%)
Intensity (m/s)
MDD
PAA
MDR

5.5 DiscRates class

Discount rates are used to calculate the net present value of any future or past value. They are thus used to compare amounts paid (costs) and received (benefits) in different years. A project is economically viable (attractive), if the net present value of benefits exceeds the net present value of costs - a const-benefit ratio < 1.

There are several important implications that come along with discount rates. Namely, that higher discount rates lead to smaller net present values of future impacts (costs). As a consequence of that, climate action and mitigation measures can be postboned. In the literature higher discount rates are typically justified by the expectation of continued exponential growth of the economy. The most widley used interest rate in climate change economics is 1.4% as propsed by the Stern Review (2006). Neoliberal economists around Nordhaus (2007) claim that rates should be higher, around 4.3%. Environmental economists argue that future costs shouldn't be discounted at all. This is especially true for non-monetary variables such as ecosystems or human lifes, where no price tag should be applied out of ethical reasons. This discussion has a long history, reaching back to the 18th century: "Some things have a price, or relative worth, while other things have a dignity, or inner worth" (Kant, 1785).

This class contains the discount rates for every year and discounts given values. Its attributes are:

- tag (Tag): information about the source data
- years (np.array): years
- rates (np.array): discount rates for each year (between 0 and 1)

[1]:
```
from climada.entity import DiscRates
help(DiscRates)
```

```
Help on class DiscRates in module climada.entity.disc_rates.base:

class DiscRates(builtins.object)
 |  DiscRates(years=None, rates=None, tag=None)
 |
 |  Defines discount rates and basic methods. Loads from
 |  files with format defined in FILE_EXT.
 |
 |  Attributes
 |  ---------
 |  tag: Tag
 |      information about the source data
 |  years: np.array
 |      list of years
 |  rates: np.array
 |      list of discount rates for each year (between 0 and 1)
 |
 |  Methods defined here:
 |
 |  __init__(self, years=None, rates=None, tag=None)
 |      Fill discount rates with values and check consistency data
 |
 |      Parameters
 |      ----------
 |      years : numpy.ndarray(int)
 |          Array of years. Default is numpy.array([]).
 |      rates : numpy.ndarray(float)
 |          Discount rates for each year in years.
 |          Default is numpy.array([]).
```

(continues on next page)

(continued from previous page)

```
|          Note: rates given in float, e.g., to set 1% rate use 0.01
|      tag : climate.entity.tag
|          Metadata. Default is None.
|
|  append(self, disc_rates)
|      Check and append discount rates to current DiscRates. Overwrite
|      discount rate if same year.
|
|      Parameters
|      ----------
|      disc_rates: climada.entity.DiscRates
|          DiscRates instance to append
|
|      Raises
|      ------
|      ValueError
|
|  check(self)
|      Check attributes consistency.
|
|      Raises
|      ------
|      ValueError
|
|  clear(self)
|      Reinitialize attributes.
|
|  net_present_value(self, ini_year, end_year, val_years)
|      Compute net present value between present year and future year.
|
|      Parameters
|      ----------
|      ini_year: float
|          initial year
|      end_year: float
|          end year
|      val_years: np.array
|          cash flow at each year btw ini_year and end_year (both included)
|
|      Returns
|      -------
|          net_present_value: float
|              net present value between present year and future year.
|
|  plot(self, axis=None, figsize=(6, 8), **kwargs)
|      Plot discount rates per year.
|
|      Parameters
|      ----------
|      axis: matplotlib.axes._subplots.AxesSubplot, optional
|          axis to use
|      figsize: tuple(int, int), optional
```

(continues on next page)

(continued from previous page)

```
|          size of the figure. The default is (6,8)
|      kwargs: optional
|          keyword arguments  passed to plotting function axis.plot
|
|      Returns
|      -------
|      axis: matplotlib.axes._subplots.AxesSubplot
|          axis handles of the plot
|
|  read_excel(self, *args, **kwargs)
|      This function is deprecated, use DiscRates.from_excel instead.
|
|  read_mat(self, *args, **kwargs)
|      This function is deprecated, use DiscRates.from_mats instead.
|
|  select(self, year_range)
|      Select discount rates in given years.
|
|      Parameters
|      ----------
|      year_range: np.array(int)
|          continuous sequence of selected years.
|
|      Returns: climada.entity.DiscRates
|          The selected discrates in the year_range
|
|  write_excel(self, file_name, var_names={'sheet_name': 'discount', 'col_name': {'year
↪': 'year', 'disc': 'discount_rate'}})
|      Write excel file following template.
|
|      Parameters
|      ----------
|      file_name: str
|          filename including path and extension
|      var_names: dict, optional
|          name of the variables in the file. The Default is
|          DEF_VAR_EXCEL = {'sheet_name': 'discount',
|             'col_name': {'year': 'year', 'disc': 'discount_rate'}}
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  from_excel(file_name, description='', var_names={'sheet_name': 'discount', 'col_name
↪': {'year': 'year', 'disc': 'discount_rate'}}) from builtins.type
|      Read excel file following template and store variables.
|
|      Parameters
|      ----------
|      file_name: str
|          filename including path and extension
|      description: str, optional
|          description of the data. The default is ''
```

(continues on next page)

(continued from previous page)

```
|      var_names: dict, optional
|          name of the variables in the file. The Default is
|          DEF_VAR_EXCEL = {'sheet_name': 'discount',
|             'col_name': {'year': 'year', 'disc': 'discount_rate'}}
|
|      Returns
|      -------
|      climada.entity.DiscRates :
|          The disc rates from excel
|
|  from_mat(file_name, description='', var_names={'sup_field_name': 'entity', 'field_
↪name': 'discount', 'var_name': {'year': 'year', 'disc': 'discount_rate'}}) from␣
↪builtins.type
|      Read MATLAB file generated with previous MATLAB CLIMADA version.
|
|      Parameters
|      ----------
|      file_name: str
|          filename including path and extension
|      description: str, optional
|          description of the data. The default is ''
|      var_names: dict, optional
|          name of the variables in the file. The Default is
|          DEF_VAR_MAT = {'sup_field_name': 'entity', 'field_name': 'discount',
|             'var_name': {'year': 'year', 'disc': 'discount_rate'}}
|
|      Returns
|      -------
|      climada.entity.DiscRates :
|          The disc rates from matlab
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

An example of use - we define discount rates and apply them on a coastal protection scheme which initially costs 100 mn. USD plus 75'000 USD maintance each year, starting after 10 years. Net present value of the project can be calculated as displayed:

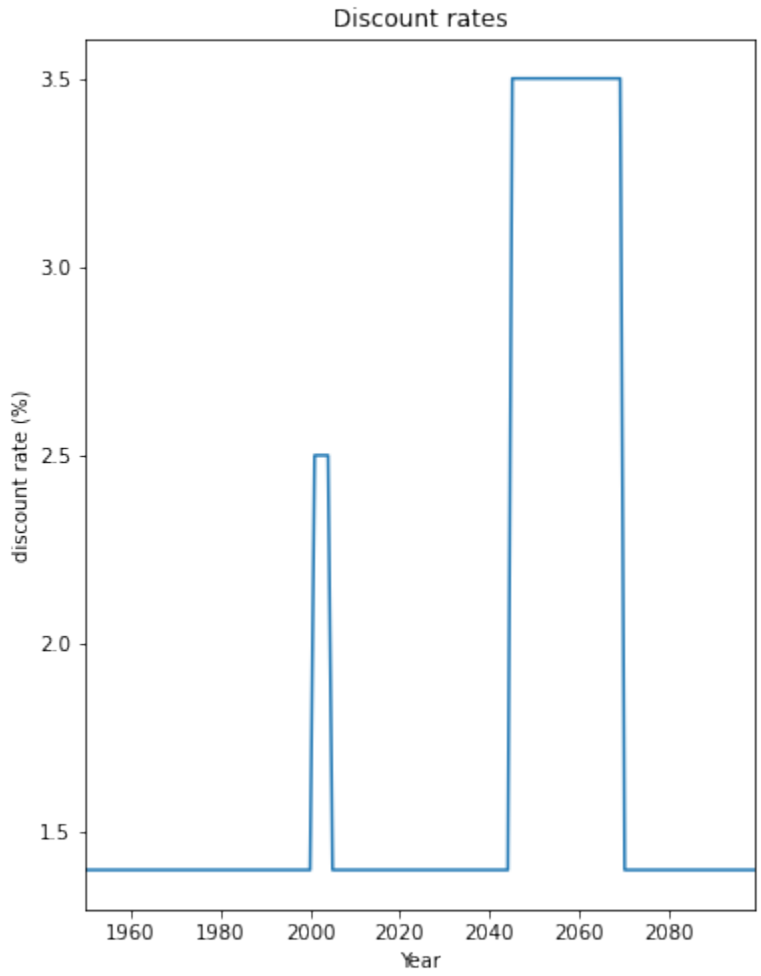
[3]:
```
%matplotlib inline
import numpy as np
from climada.entity import DiscRates

# define discount rates
years = np.arange(1950, 2100)
rates = np.ones(years.size) * 0.014
```

(continues on next page)

(continued from previous page)

```
rates[51:55] = 0.025
rates[95:120] = 0.035
disc = DiscRates(years=years, rates=rates)
disc.plot()

# Compute net present value between present year and future year.
ini_year = 2019
end_year = 2050
val_years = np.zeros(end_year-ini_year+1)
val_years[0] = 100000000 # initial investment
val_years[10:] = 75000 # maintenance from 10th year
npv = disc.net_present_value(ini_year, end_year, val_years)
print('net present value: {:.5e}'.format(npv))
```

```
net present value: 1.01231e+08
```



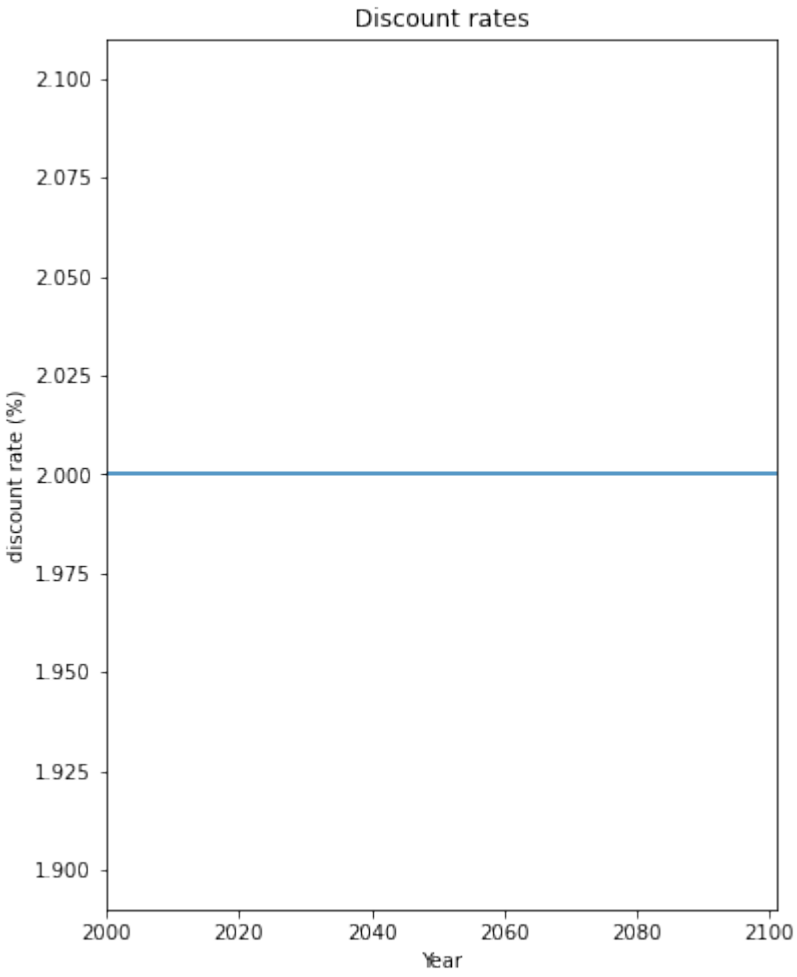
Read discount rates of an Excel file

Discount rates defined in an excel file following the template provided in sheet `discount` of `climada_python/data/system/entity_template.xlsx` can be ingested directly using the method `from_excel()`.

[4]:
```
from climada.entity import DiscRates
from climada.util import ENT_TEMPLATE_XLS

# Fill DataFrame from Excel file
file_name = ENT_TEMPLATE_XLS # provide absolute path of the excel file
disc = DiscRates.from_excel(file_name)
print('Read file:', disc.tag.file_name)
disc.plot()
```

```
Read file: /Users/ckropf/climada/data/entity_template.xlsx
```

[4]:
```
<AxesSubplot:title={'center':'Discount rates'}, xlabel='Year', ylabel='discount rate (%)
↪'>
```



5.5.1 Write discount rates

Discount rates defined in an excel file following the template provided in sheet `discount` of `climada_python/data/system/entity_template.xlsx` can be ingested directly using the method `from_excel()`.

[5]:
```
from climada.entity import DiscRates
from climada.util import ENT_TEMPLATE_XLS

# Fill DataFrame from Excel file
file_name = ENT_TEMPLATE_XLS # provide absolute path of the excel file
disc = DiscRates.from_excel(file_name)

# write file
disc.write_excel('results/tutorial_disc.xlsx')
```

Pickle can always be used as well:

[5]:
```
from climada.util.save import save
# this generates a results folder in the current path and stores the output there
save('tutorial_disc.p', disc)
```

5.6 How to use polygons or lines as exposure

Exposure in CLIMADA are usually represented as individual points or a raster of points. See *Exposures* tutorial to learn how to fill and use exposures. In this tutorial we show you how to use CLIMADA Impf you have your exposure in the form of shapes/polygons or in the form of lines.

The approach follows three steps: 1. transform your polygon or line in a set of points 2. do the impact calculation in CLIMADA with that set of points 3. transform the calculated Impact back to your polygon or line

5.6.1 Polygons

Polygons or shapes are a common geographical representation of countries, states etc. as for example in NaturalEarth. Here we want to show you how to deal with exposure information as polygons.

Lets assume we have the following data given. The polygons of the admin-1 regions of the netherlands and an exposure value each. We want to know the Impact of Lothar on each admin-1 region.

[1]:
```
from cartopy.io import shapereader
from climada_petals.entity.exposures.black_marble import country_iso_geom

# open the file containing the Netherlands admin-1 polygons
shp_file = shapereader.natural_earth(resolution='10m',
                                     category='cultural',
                                     name='admin_0_countries')
shp_file = shapereader.Reader(shp_file)

# extract the NL polygons
prov_names = {'Netherlands': ['Groningen', 'Drenthe',
                              'Overijssel', 'Gelderland',
                                            'Limburg', 'Zeeland',
                                            'Noord-Brabant', 'Zuid-Holland',
```

(continues on next page)

(continued from previous page)

```
                                                'Noord-Holland', 'Friesland',
                                                'Flevoland', 'Utrecht']
                                }
polygon_Netherlands, polygons_prov_NL = country_iso_geom(prov_names,
                                shp_file)

# assign a value to each admin-1 area (assumption 100'000 USD per inhabitant)
population_prov_NL = {'Drenthe':493449, 'Flevoland':422202,
                  'Friesland':649988, 'Gelderland':2084478,
                  'Groningen':585881, 'Limburg':1118223,
                  'Noord-Brabant':2562566, 'Noord-Holland':2877909,
                  'Overijssel':1162215, 'Zuid-Holland':3705625,
                  'Utrecht':1353596, 'Zeeland':383689}
value_prov_NL = {n: 100000 * population_prov_NL[n] for n in population_prov_NL.keys()}
```

Assume a unImpform distribution of values within your polygons

This helps you in the case you have a given total exposure value per polygon and we assume this value is distributed evenly within the polygon.

We can now perform the three steps for this example:

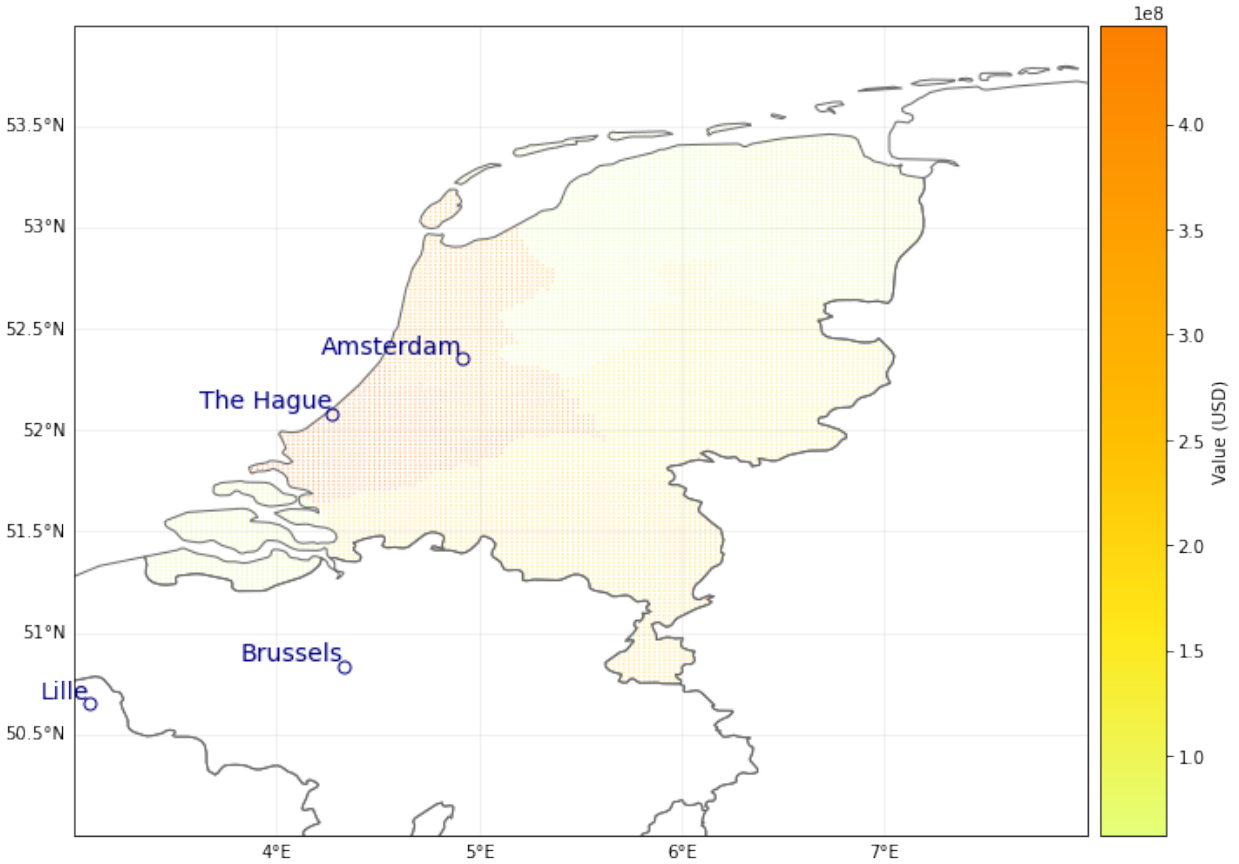
[2]:
```
import numpy as np
from pandas import DataFrame
from climada.entity import Exposures
from climada.util.coordinates import coord_on_land


### 1. transform your polygon or line in a set of points
# create exposure with points
exp_df = DataFrame()
n_exp = 200*200
lat, lon = np.mgrid[50 : 54 : complex(0, np.sqrt(n_exp)),
                    3 : 8 : complex(0, np.sqrt(n_exp))]
exp_df['latitude'] = lat.flatten() # provide latitude
exp_df['longitude'] = lon.flatten() # provide longitude
exp_df['impf_WS'] = np.ones(n_exp, int) # provide impact functions

# now we assign each point a province and a value, Impf the points are within one of the␣
↪polygons defined above
exp_df['province'] = ''
exp_df['value'] = np.ones((exp_df.shape[0],))*np.nan
for prov_name_i, prob_polygon_i in zip(prov_names['Netherlands'],polygons_prov_NL['NLD
↪']):
    in_geom = coord_on_land(lat=exp_df['latitude'],
                                            lon=exp_df['longitude'],
                                            land_geom=prob_polygon_i)
    np.put(exp_df['province'].values, np.where(in_geom)[0], prov_name_i)
    np.put(exp_df['value'].values, np.where(in_geom)[0], value_prov_NL[prov_name_i]/
↪sum(in_geom))
```

(continues on next page)

(continued from previous page)

```
exp_df = Exposures(exp_df)
exp_df.set_geometry_points() # set geometry attribute (shapely Points) from GeoDataFrame␣
↪from latitude and longitude
exp_df.check()
exp_df.plot_hexbin()
```

[2]:
```
<GeoAxesSubplot:>
```



[3]:
```
from climada.hazard.storm_europe import StormEurope
from climada.util.constants import WS_DEMO_NC
from climada.entity.impact_funcs.storm_europe import ImpfStormEurope
from climada.entity.impact_funcs import ImpactFuncSet
from climada.engine import Impact

### 2. do the impact calculation in CLIMADA with that set of points
# define hazard
storms = StormEurope.from_footprints(WS_DEMO_NC, description='test_description')
# define impact function
impact_func = ImpfStormEurope.from_welker()
impact_function_set = ImpactFuncSet()
impact_function_set.append(impact_func)
# calculate hazard
impact_NL = Impact()
```

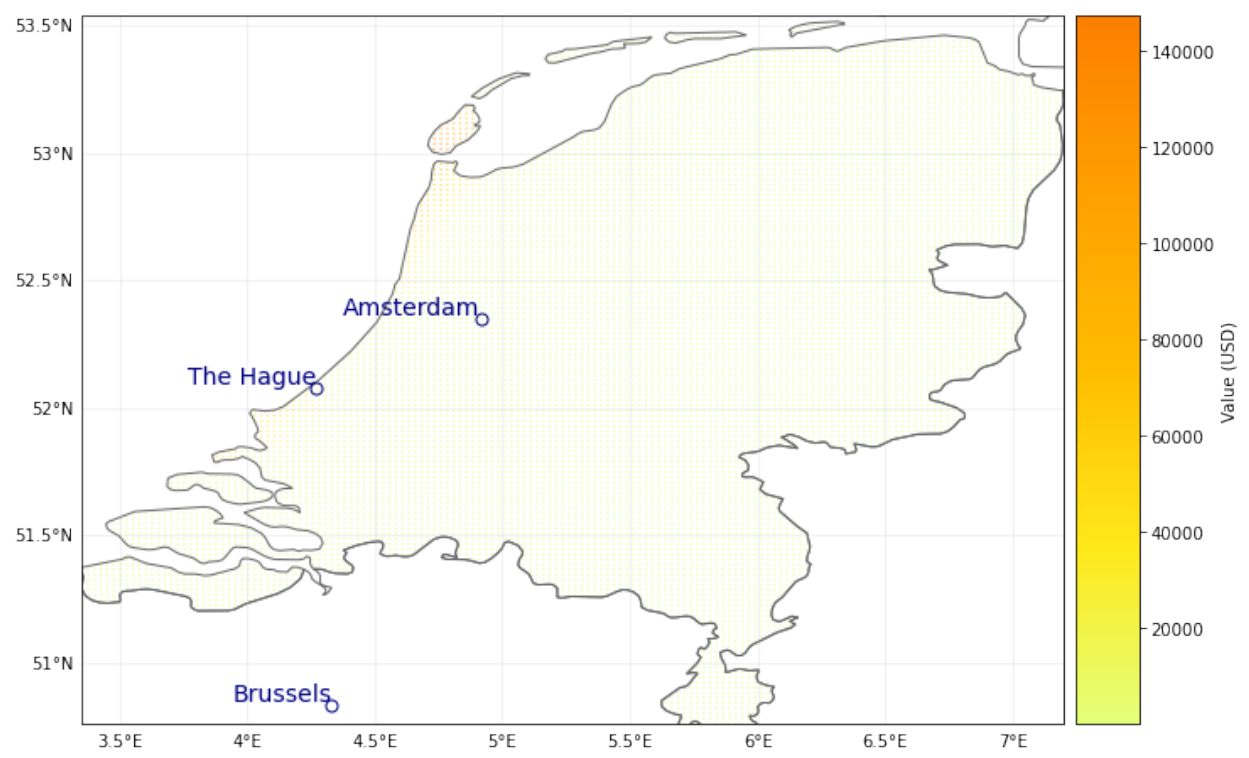
(continues on next page)

(continued from previous page)

```
impact_NL.calc(exp_df, impact_function_set, storms, save_mat=True)
impact_NL.plot_hexbin_impact_exposure()
```

```
$CLIMADA_SRC/climada/hazard/centroids/centr.py:822: UserWarning: Geometry is in a␣
↪geographic CRS. Results from 'buffer' are likely incorrect. Use 'GeoSeries.to_crs()'␣
↪to re-project geometries to a projected CRS before this operation.

  xy_pixels = self.geometry.buffer(res / 2).envelope
```

[3]: <GeoAxesSubplot:>



[4]:
```
import pandas as pd

### 3. transform the calculated Impact back to your polygon or line
impact_at_province_raw = pd.DataFrame(np.mean(impact_NL.imp_mat.todense().transpose(),␣
↪axis=1),
                                      index=exp_df.gdf['province'])
impact_at_province = impact_at_province_raw.groupby(impact_at_province_raw.index).sum()
print(impact_at_province)
```

```
                         0
province
              0.000000e+00
Drenthe       6.852537e+04
Flevoland     2.716078e+05
Friesland     7.782136e+05
Gelderland    4.056456e+05
Groningen     1.150089e+05
Limburg       2.559739e+05
```

(continues on next page)

(continued from previous page)

```
Noord-Brabant  7.391625e+05
Noord-Holland  5.908293e+06
Overijssel     1.588620e+05
Utrecht        4.846288e+05
Zeeland        4.069767e+05
Zuid-Holland   2.893966e+06
```

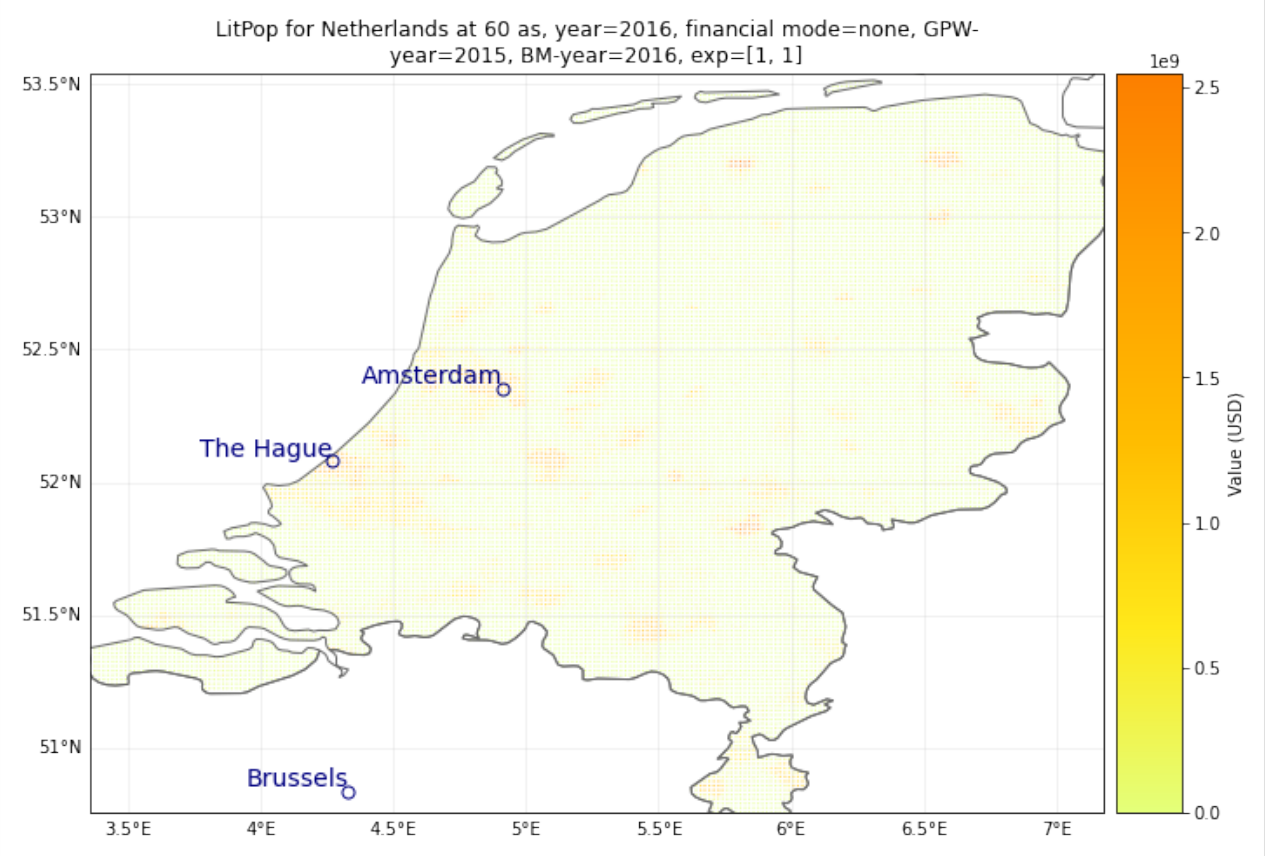
Use the LitPop module to disaggregate your exposure values

Instead of a unImpform distribution, another geographical distribution can be chosen to disaggregate within the value within each polygon. We show here the example of *LitPop*. The same three steps apply:

[5]:
```
import numpy as np
from climada.entity import LitPop
from climada.util.coordinates import coord_on_land


### 1. transform your polygon or line in a set of points
# create exposure with points
exp_df_lp = LitPop.from_countries('Netherlands',res_arcsec = 60, fin_mode = 'none')
exp_df_lp.gdf['impf_WS'] = np.ones(exp_df_lp.gdf.shape[0], int) # provide impact␣
↪functions

# now we assign each point a province and a value, Impf the points are within one of the␣
↪polygons defined above
exp_df_lp.gdf['province'] = ''
for prov_name_i, prob_polygon_i in zip(prov_names['Netherlands'],polygons_prov_NL['NLD
↪']):
    in_geom = coord_on_land(lat=exp_df_lp.gdf['latitude'],
                                            lon=exp_df_lp.gdf['longitude'],
                                            land_geom=prob_polygon_i)
    np.put(exp_df_lp.gdf['province'].values, np.where(in_geom)[0], prov_name_i)
    exp_df_lp.gdf['value'][np.where(in_geom)[0]] =  \
                exp_df_lp.gdf['value'][np.where(in_geom)[0]] * value_prov_NL[prov_name_
↪i]/sum(exp_df_lp.gdf['value'][np.where(in_geom)[0]])
exp_df_lp.gdf = exp_df_lp.gdf.drop(np.where(exp_df_lp.gdf['province']=='')[0]) #drop␣
↪carribean islands for this example
exp_df_lp.set_geometry_points()
exp_df_lp.check()
exp_df_lp.plot_hexbin()
```

```
2021-10-19 16:54:04,710 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

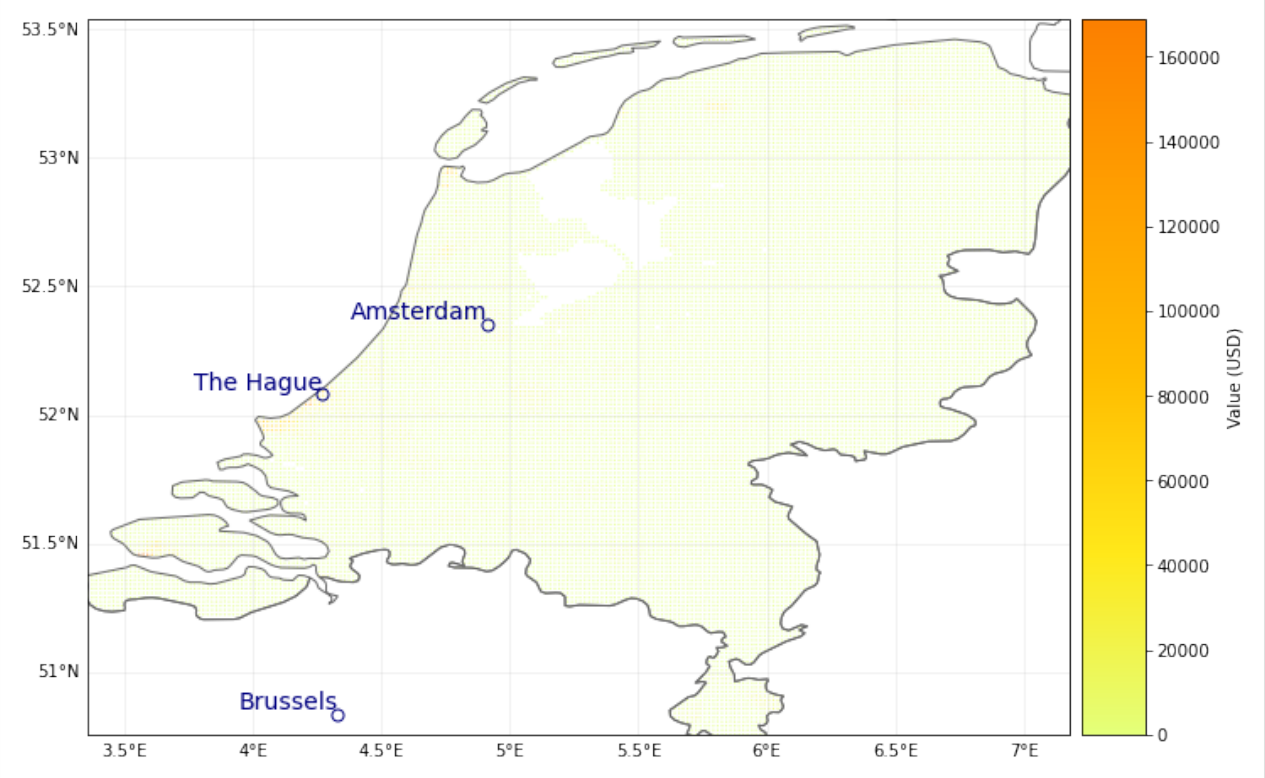
[5]:
```
<GeoAxesSubplot:title={'center':"LitPop Exposure for ['Netherlands'] at 60 as, year:␣
↪2018, financial\nmode: none, exp: (1, 1), admin1_calc: False"}>
```



[6]:
```
from climada.hazard.storm_europe import StormEurope
from climada.util.constants import WS_DEMO_NC
from climada.entity.impact_funcs.storm_europe import ImpfStormEurope
from climada.entity.impact_funcs import ImpactFuncSet
from climada.engine import Impact

### 2. do the impact calculation in CLIMADA with that set of points
# define hazard
storms = StormEurope.from_footprints(WS_DEMO_NC, description='test_description')
# define impact function
impact_func = ImpfStormEurope.from_welker()
impact_function_set = ImpactFuncSet()
impact_function_set.append(impact_func)
# calculate hazard
impact_NL = Impact()
impact_NL.calc(exp_df_lp, impact_function_set, storms, save_mat=True)
impact_NL.plot_hexbin_impact_exposure()
```

```
$CLIMADA_SRC/climada/hazard/centroids/centr.py:822: UserWarning: Geometry is in a␣
↪geographic CRS. Results from 'buffer' are likely incorrect. Use 'GeoSeries.to_crs()'␣
↪to re-project geometries to a projected CRS before this operation.

  xy_pixels = self.geometry.buffer(res / 2).envelope
```

[6]:
```
<GeoAxesSubplot:>
```



[7]:
```
import pandas as pd

### 3. transform the calculated Impact back to your polygon or line
impact_at_province_raw = pd.DataFrame(np.mean(impact_NL.imp_mat.todense().transpose(),␣
↪axis=1),
                                      index=exp_df_lp.gdf['province'])
impact_at_province_lp = impact_at_province_raw.groupby(impact_at_province_raw.index).
↪sum()
print(impact_at_province_lp)
```

```
                         0
province
Drenthe         6.500784e+04
Flevoland       1.684639e+05
Friesland       4.575450e+05
Gelderland      3.345681e+05
Groningen       1.604080e+05
Limburg         3.647827e+05
Noord-Brabant   6.253040e+05
Noord-Holland   1.819285e+06
Overijssel      1.094167e+05
Utrecht         4.495409e+05
Zeeland         7.701477e+05
Zuid-Holland    2.823286e+06
```

Comparison of both modelled impacts:

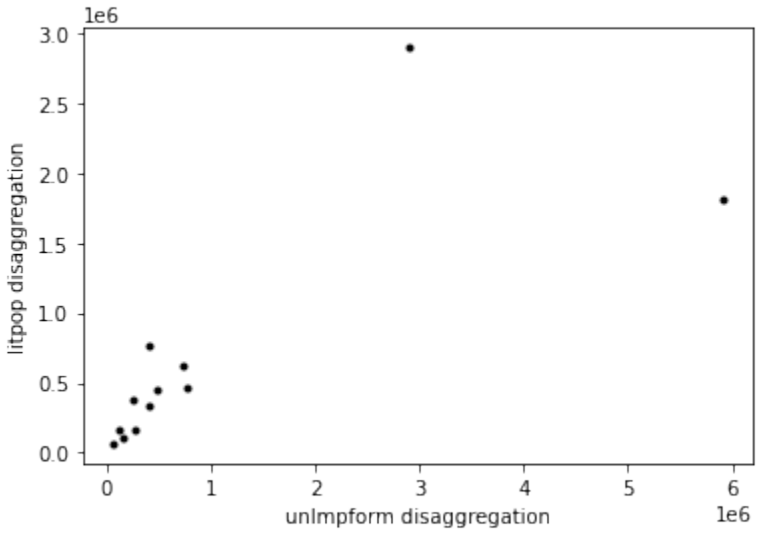
[8]:
```
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.plot(impact_at_province[impact_at_province.index!=''],impact_at_province_lp, '.k')
plt.xlabel('unImpform disaggregation')
plt.ylabel('litpop disaggregation')
```

[8]:
```
Text(0, 0.5, 'litpop disaggregation')
```



Further statistical analysis of hazard on polygon level

imagine that you need access to the hazard centroids in oder to provide some statistical analysis on the province level

[9]:
```
# this provides the wind speed value for each event at the corresponding exposure
import scipy

exp_df_lp.gdf[:5]
l1, l2, vals = scipy.sparse.find(storms.intensity)
# provide columns for both events
exp_df_lp.gdf['wind_0'] = 0
exp_df_lp.gdf['wind_1'] = 0
for evt, idx, val in zip(l1, l2, vals):
    if evt == 0:
        exp_df_lp.gdf['wind_0'].values[exp_df_lp.gdf['centr_WS'] == idx] = val
    else:
        exp_df_lp.gdf['wind_1'].values[exp_df_lp.gdf['centr_WS'] == idx] = val
```

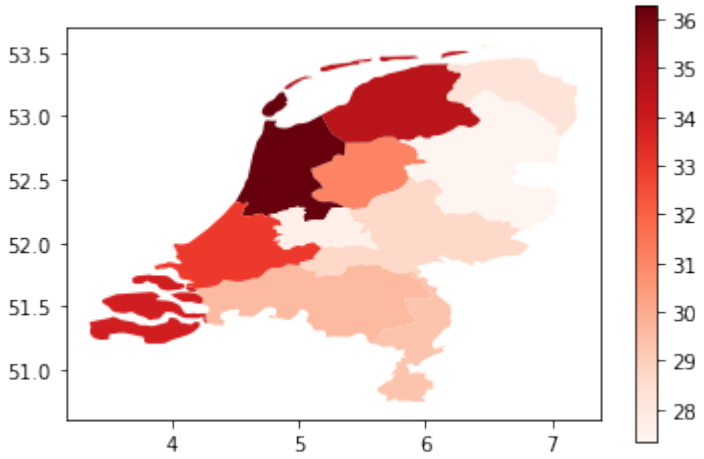
[10]:
```
# now you can perform additional statistical analysis and aggregate it to the province␣
↪level
import pandas as pd
import geopandas as gpd

exp_province_raw = exp_df_lp.copy()

def f(x):  # define function for statistical aggregation with pandas
    d = {}
```

(continues on next page)

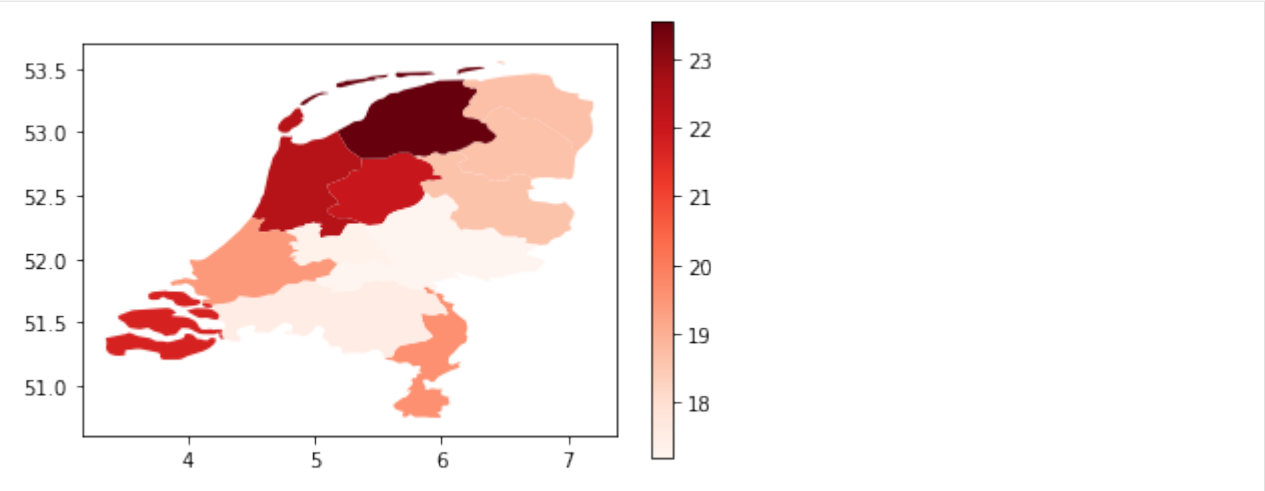
(continued from previous page)

```
    d['value'] = x['value'].sum()
    d['wind_0'] = x['wind_0'].max()
    d['wind_1'] = x['wind_1'].mean()
    # one could also be interested in centroid of max wind with respect to province
    #d['centr_WS'] = x.loc[x.index[x['wind_0'].max()],'centr_WS']
    return pd.Series(d, index=['value', 'wind_0', 'wind_1'])

exp_province = exp_province_raw.gdf.groupby('province').apply(f).reset_index()  # Result␣
↪is not a GeoDataFrame anymore
# add geometries to DataFrame and plot results
exp_province=gpd.GeoDataFrame(exp_province, geometry=None)
for prov,poly in zip(list(prov_names.values())[0],polygons_prov_NL['NLD']):
    exp_province.loc[exp_province.index[exp_province['province']== prov],'geometry']=␣
↪gpd.GeoDataFrame(geometry=[poly]).geometry.values
exp_province
print('Plot maximum wind per province for first event')
exp_province.plot(column='wind_0', cmap='Reds', legend=True)
plt.show()
print('Plot mean wind per province for second event')
exp_province.plot(column='wind_1', cmap='Reds', legend=True)
plt.show()
```

```
Plot maximum wind per province for first event
```




```
Plot mean wind per province for second event
```



5.6.2 Lines

Lines are common geographical representation of transport infrastructure like streets, train tracks or powerlines etc.

[11]:
```
# under construction. here follows an example on how to deal with lines
```

5.7 Adaptation Measures

Adaptation measures are defined by parameters that alter the exposures, hazard or impact functions. Risk transfer options are also considered. Single measures are defined in the `Measure` class, which can be aggregated to a `MeasureSet`.

5.7.1 Measure class

A measure is characterized by the following attributes:

Related to measure's description: * name (str): name of the action * haz_type (str): related hazard type (peril), e.g. TC * color_rgb (np.array): integer array of size 3. Gives color code of this measure in RGB * cost (float): discounted cost (in same units as assets). Needs to be provided by the user. See the example provided in `climada_python/data/system/entity_template.xlsx` sheets `_measures_details` and `_discounting_sheet` to see how the discounting is done.

Related to a measure's impact: * hazard_set (str): file name of hazard to use * hazard_freq_cutoff (float): hazard frequency cutoff * exposure_set (str): file name of exposure to use * hazard_inten_imp (tuple): parameter a and b of hazard intensity change * mdd_impact (tuple): parameter a and b of the impact over the mean damage degree * paa_impact (tuple): parameter a and b of the impact over the percentage of affected assets * imp_fun_map (str): change of impact function id, e.g. '1to3' * exp_region_id (int): region id of the selected exposures to consider ALL the previous parameters * risk_transf_attach (float): risk transfer attachment. Applies to the whole exposure. * risk_transf_cover (float): risk transfer cover. Applies to the whole exposure.

Paramters description:

`hazard_set` and `exposures_set` provide the file names in h5 format (generated by CLIMADA) of the hazard and exposures to use as a result of the implementation of the measure. These might be further modified when applying the other parameters.

`hazard_inten_imp`, `mdd_impact` and `paa_impact` transform the impact functions linearly as follows:

```
intensity = intensity*hazard_inten_imp[0] + hazard_inten_imp[1]
mdd = mdd*mdd_impact[0] + mdd_impact[1]
paa = paa*paa_impact[0] + paa_impact[1]
```

`hazard_freq_cutoff` modifies the hazard by putting 0 intensities to the events whose impact exceedance frequency are greater than `hazard_freq_cutoff`.

`imp_fun_map` indicates the ids of the impact function to replace and its replacement. The `impf_XX` variable of `Exposures` with the affected impact function id will be correspondingly modified (`XX` refers to the `haz_type` of the measure).

`exp_region_id` will apply all the previous changes only to the `region_id` indicated. This means that only the exposures with that `region_id` and the hazard's centroids close to them will be modified with the previous changes, the other regions will remain unaffected to the measure.

`risk_transf_attach` and `risk_transf_cover` are the deductible and coverage of any event to happen.

Methods description:

The method `check()` validates the attibutes. `apply()` applies the measure to a given exposure, impact function and hazard, returning their modified values. The parameters related to insurability (risk_transf_attach and risk_transf_cover) affect the resulting impact and are therefore not applied in the `apply()` method yet.

`calc_impact()` calls to `apply()`, applies the insurance parameters and returns the final impact and risk transfer of the measure. This method is called from the `CostBenefit` class.

The method `apply()` allows to visualize the effect of a measure. Here are some examples:

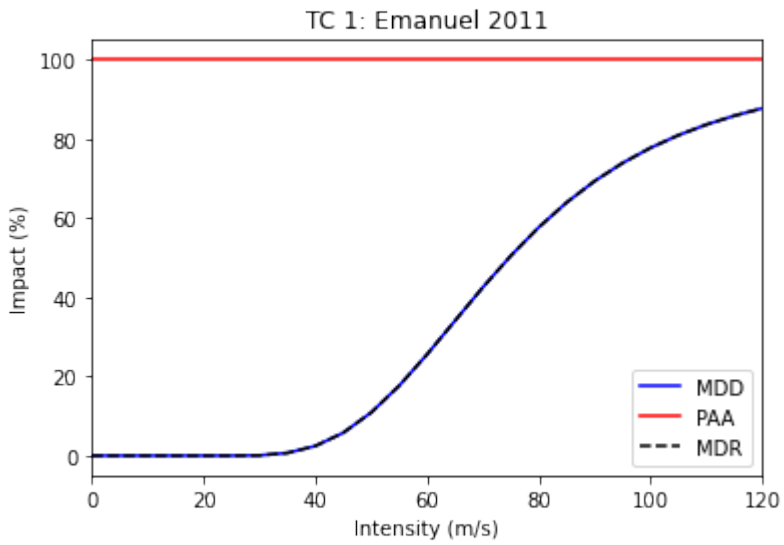
[1]:
```
# effect of mdd_impact, paa_impact, hazard_inten_imp
%matplotlib inline
import numpy as np
from climada.entity import ImpactFuncSet, ImpfTropCyclone, Exposures
from climada.entity.measures import Measure
from climada.hazard import Hazard

# define measure
meas = Measure()
meas.name = 'Mangrove'
meas.haz_type = 'TC'
meas.color_rgb = np.array([1, 1, 1])
meas.cost = 500000000
meas.mdd_impact = (1, 0)
meas.paa_impact = (1, -0.15)
meas.hazard_inten_imp = (1, -10) # reduces intensity by 10

# impact functions
impf_tc = ImpfTropCyclone.from_emanuel_usa()
impf_all = ImpactFuncSet()
impf_all.append(impf_tc)
impf_all.plot()

# dummy Hazard and Exposures
haz = Hazard('TC') # this measure does not change hazard
exp = Exposures() # this measure does not change exposures

# new impact functions
```

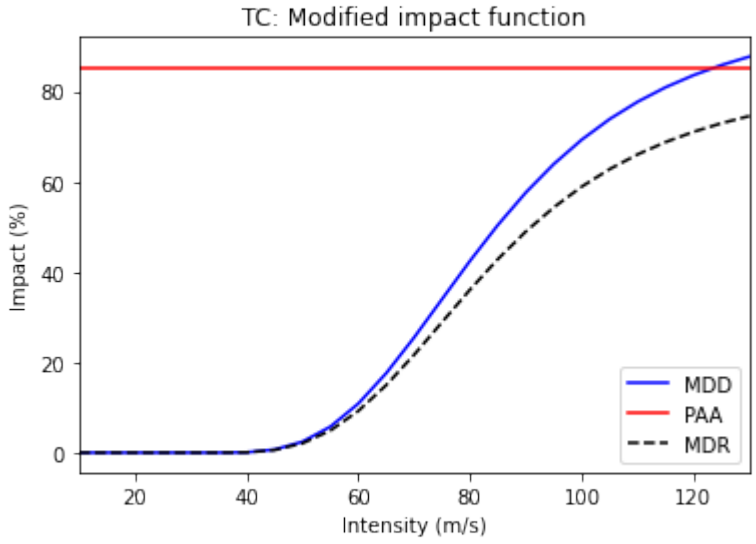
(continues on next page)

(continued from previous page)

```
new_exp, new_impfs, new_haz = meas.apply(exp, impf_all, haz)
axes = new_impfs.plot()
axes.set_title('TC: Modified impact function')
```

[1]:
```
Text(0.5, 1.0, 'TC: Modified impact function')
```





[2]:
```
# effect of hazard_freq_cutoff
import numpy as np
from climada.entity import ImpactFuncSet, ImpfTropCyclone, Exposures
from climada.entity.measures import Measure
from climada.hazard import Hazard
from climada.engine import Impact

from climada.util import HAZ_DEMO_H5, EXP_DEMO_H5

# define measure
```

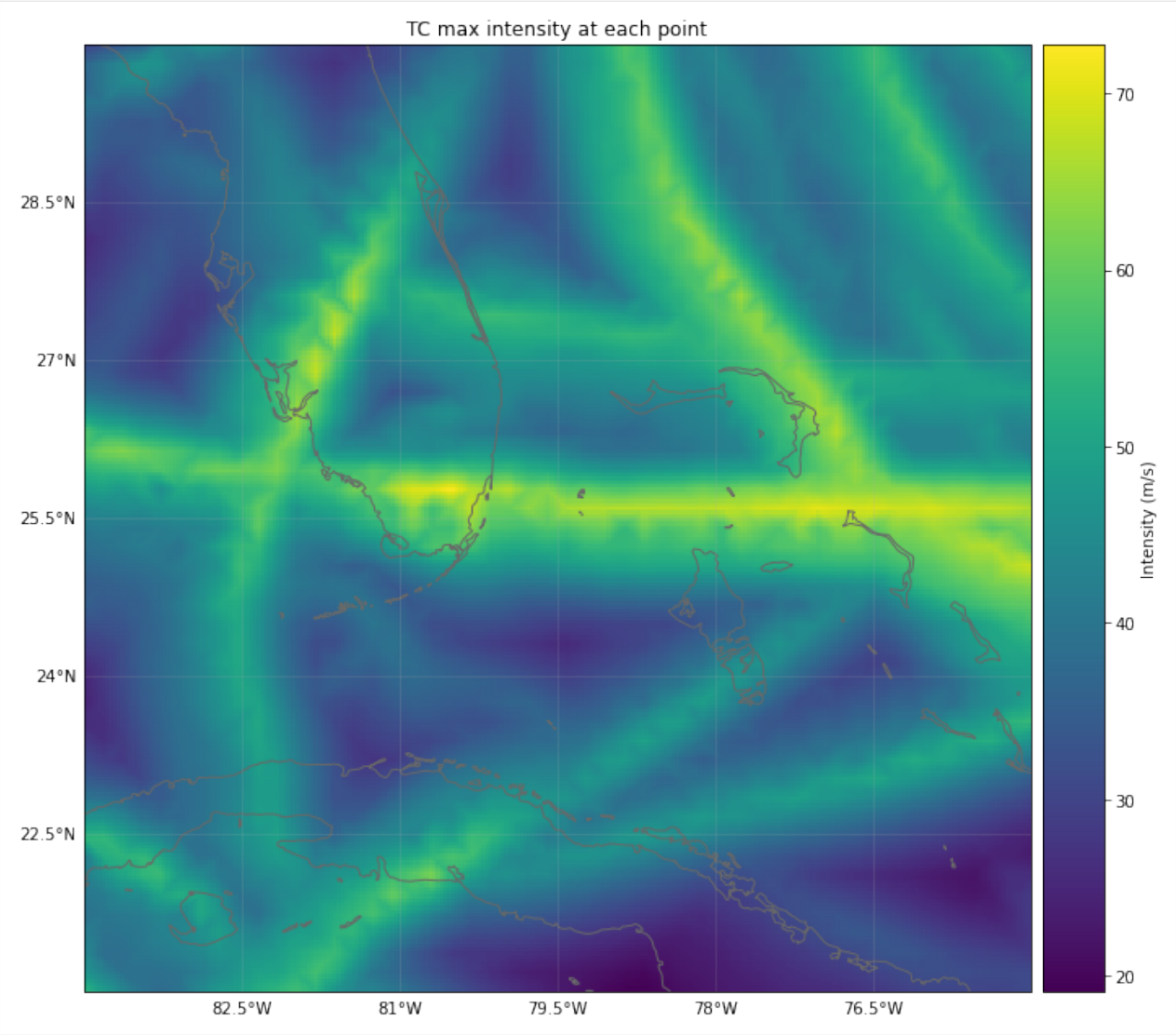
(continues on next page)

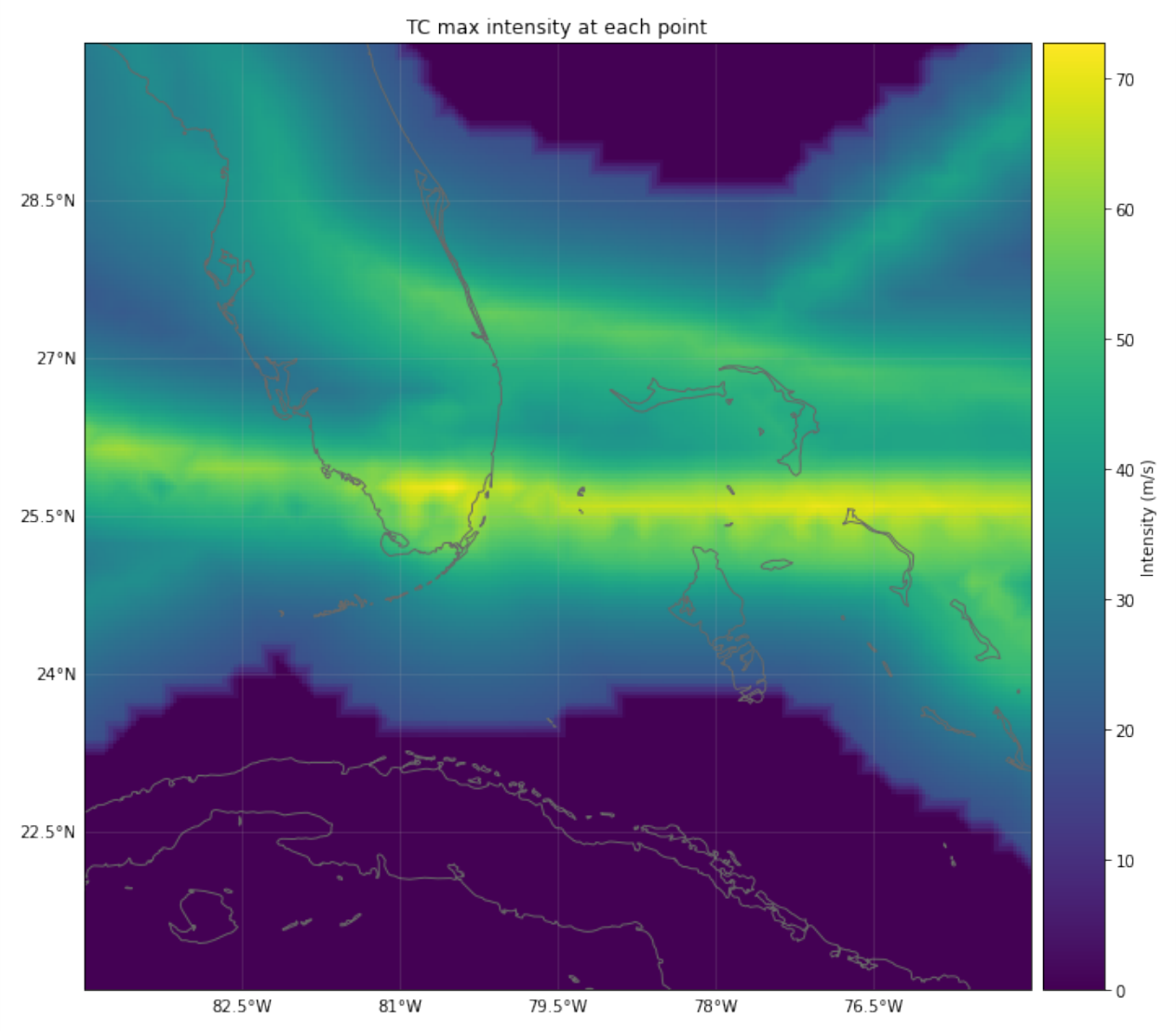
(continued from previous page)

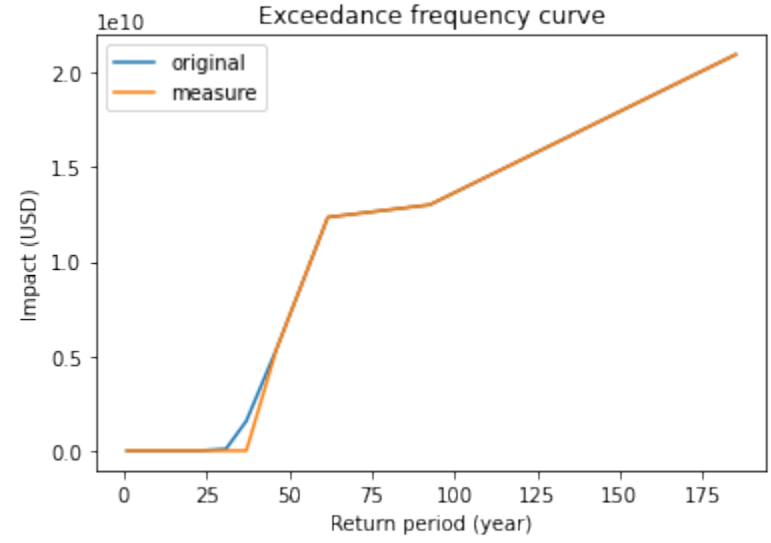
```
meas = Measure()
meas.name = 'Mangrove'
meas.haz_type = 'TC'
meas.color_rgb = np.array([1, 1, 1])
meas.cost = 500000000
meas.hazard_freq_cutoff = 0.0255

# impact functions
impf_tc = ImpfTropCyclone.from_emanuel_usa()
impf_all = ImpactFuncSet()
impf_all.append(impf_tc)

# Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
haz.check()

# Exposures
exp = Exposures.from_hdf5(EXP_DEMO_H5)
exp.check()

# new hazard
new_exp, new_impfs, new_haz = meas.apply(exp, impf_all, haz)
# if you look at the maximum intensity per centroid: new_haz does not contain the event␣
↪with smaller impact (the most frequent)
haz.plot_intensity(0)
new_haz.plot_intensity(0)
# you might also compute the exceedance frequency curve of both hazard
imp = Impact()
imp.calc(exp, impf_all, haz)
ax = imp.calc_freq_curve().plot(label='original')

new_imp = Impact()
new_imp.calc(new_exp, new_impfs, new_haz)
new_imp.calc_freq_curve().plot(axis=ax, label='measure') # the damages for events with␣
↪return periods > 1/0.0255 ~ 40 are 0
ax.legend()
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

[2]:
```
<matplotlib.legend.Legend at 0x7f433756d970>
```


TC max intensity at each point
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
70
60
50
40
30
20
Intensity (m/s)


TC max intensity at each point
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
70
60
50
40
30
20
10
0
Intensity (m/s)


Exceedance frequency curve
1e10
original
measure
2.0
1.5
1.0
0.5
0.0
Impact (USD)
0
25
50
75
100
125
150
175
Return period (year)

[3]:
```
# effect of exp_region_id
import numpy as np
from climada.entity import ImpactFuncSet, ImpfTropCyclone, Exposures
from climada.entity.measures import Measure
from climada.hazard import Hazard
from climada.engine import Impact

from climada.util import HAZ_DEMO_H5, EXP_DEMO_H5

# define measure
meas = Measure()
meas.name = 'Building code'
meas.haz_type = 'TC'
meas.color_rgb = np.array([1, 1, 1])
meas.cost = 500000000
meas.hazard_freq_cutoff = 0.00455
meas.exp_region_id = [1] # apply measure to points close to exposures with region_id=1

# impact functions
impf_tc = ImpfTropCyclone.from_emanuel_usa()
impf_all = ImpactFuncSet()
impf_all.append(impf_tc)

# Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
haz.check()

# Exposures
exp = Exposures.from_hdf5(EXP_DEMO_H5)
#exp['region_id'] = np.ones(exp.shape[0])
exp.check()
# all exposures have region_id=1
exp.plot_hexbin(buffer=1.0)

# new hazard
new_exp, new_impfs, new_haz = meas.apply(exp, impf_all, haz)
# the cutoff has been apllied only in the region of the exposures
haz.plot_intensity(0)
new_haz.plot_intensity(0)

# the exceddance frequency has only been computed for the selected exposures before␣
↪doing the cutoff.
# since we have removed the hazard of the places with exposure, the new exceedance␣
↪frequency curve is zero.
imp = Impact()
imp.calc(exp, impf_all, haz)
imp.calc_freq_curve().plot()

new_imp = Impact()
new_imp.calc(new_exp, new_impfs, new_haz)
new_imp.calc_freq_curve().plot()
```

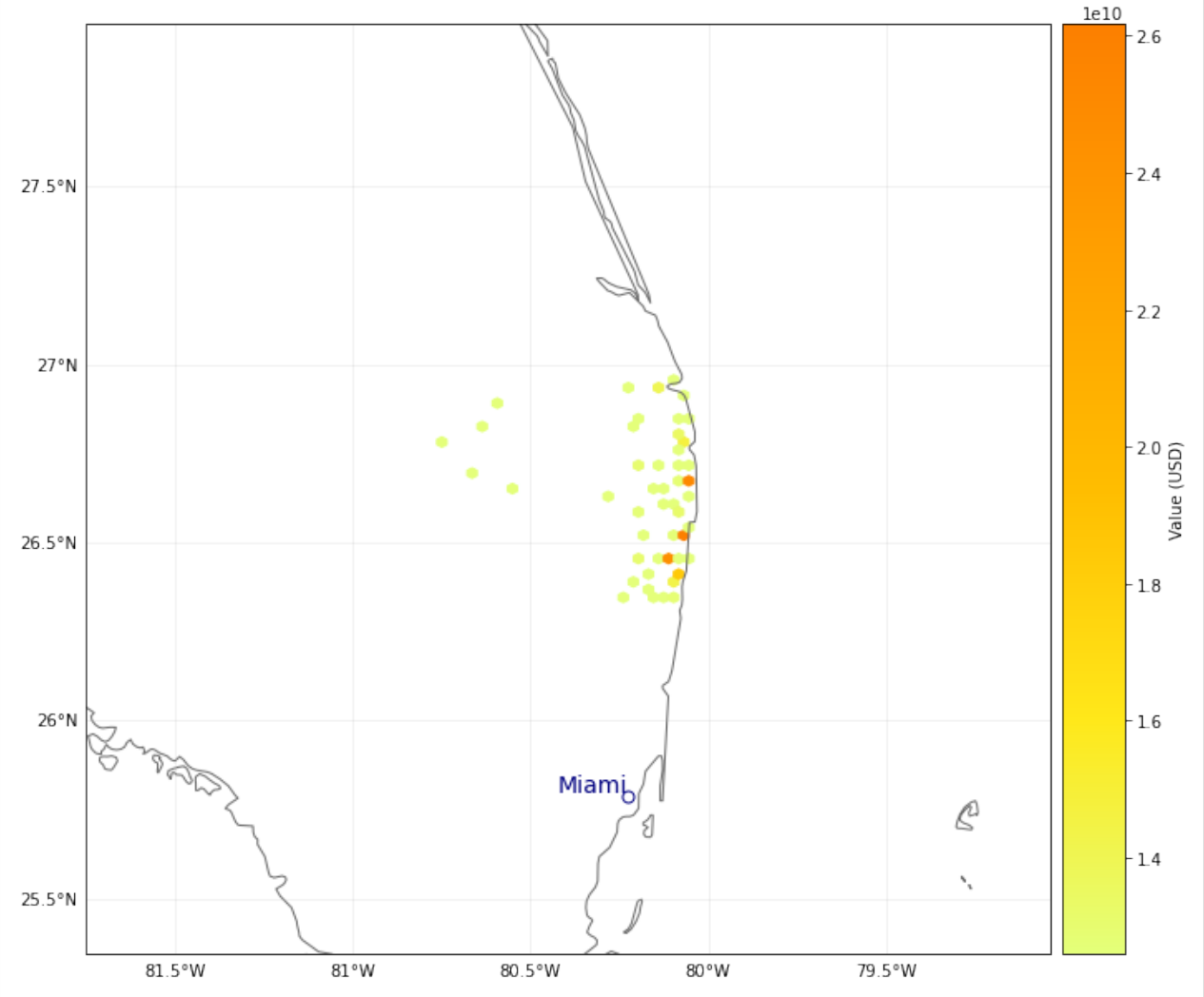
```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
```

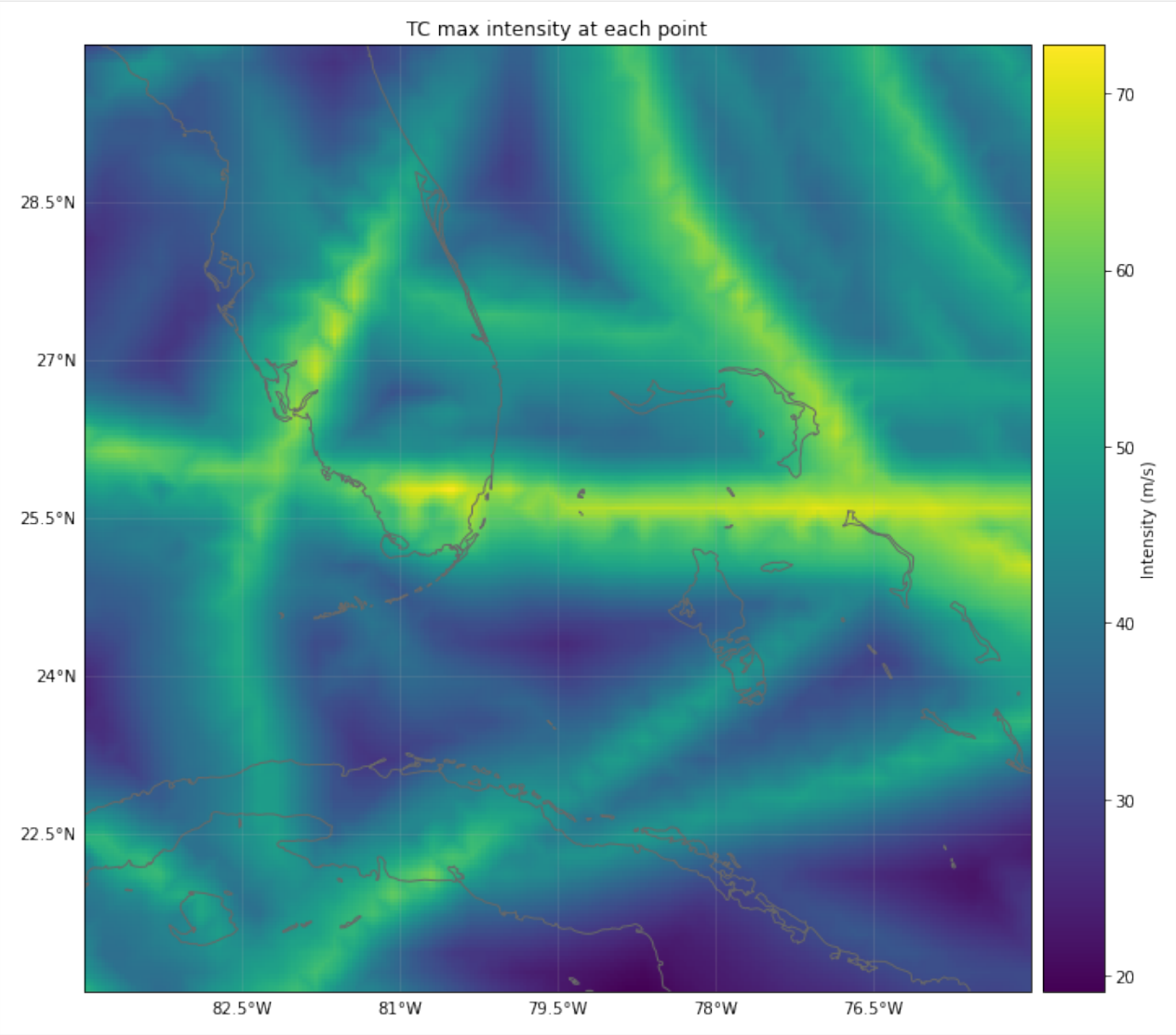
(continues on next page)

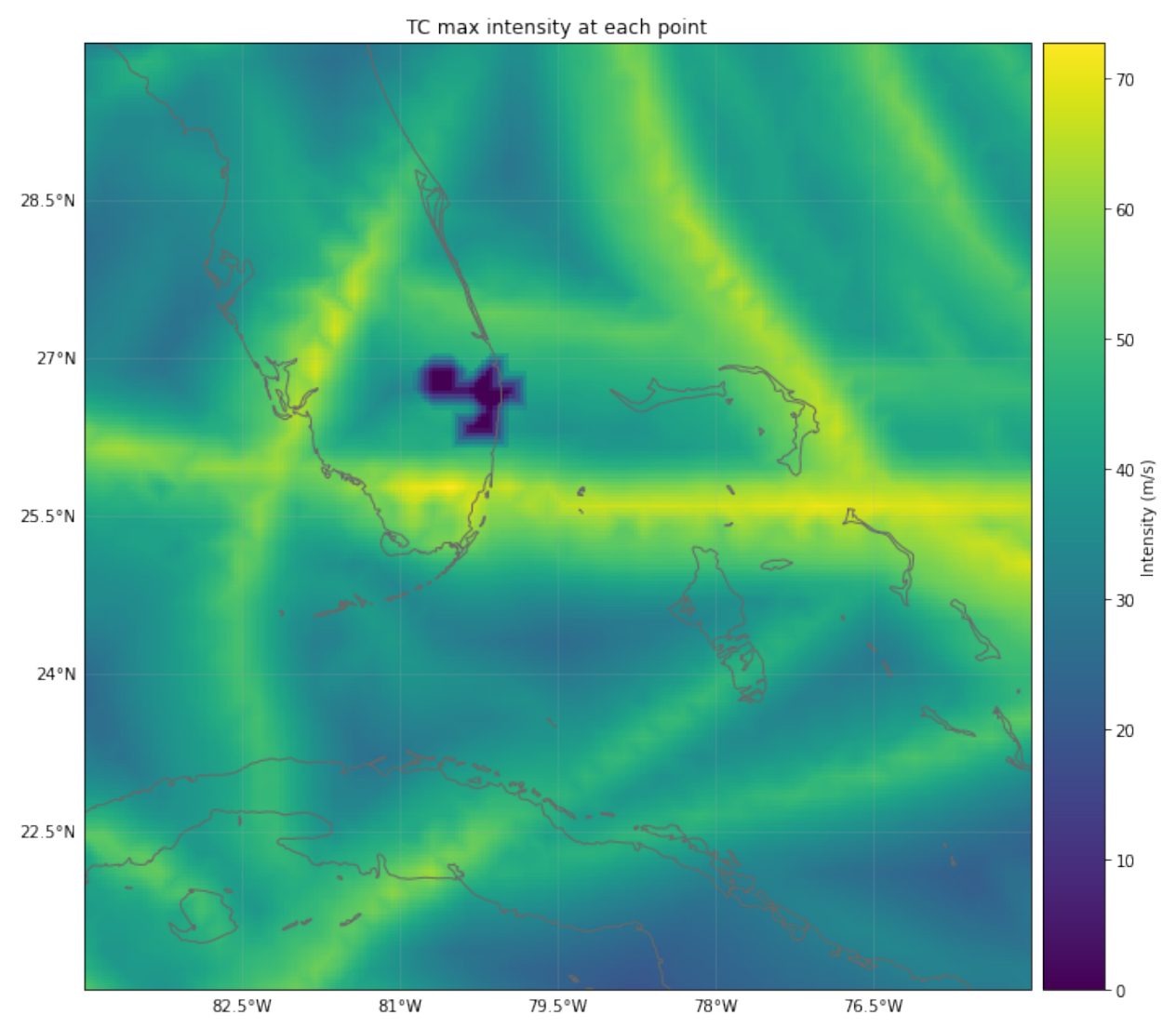
(continued from previous page)

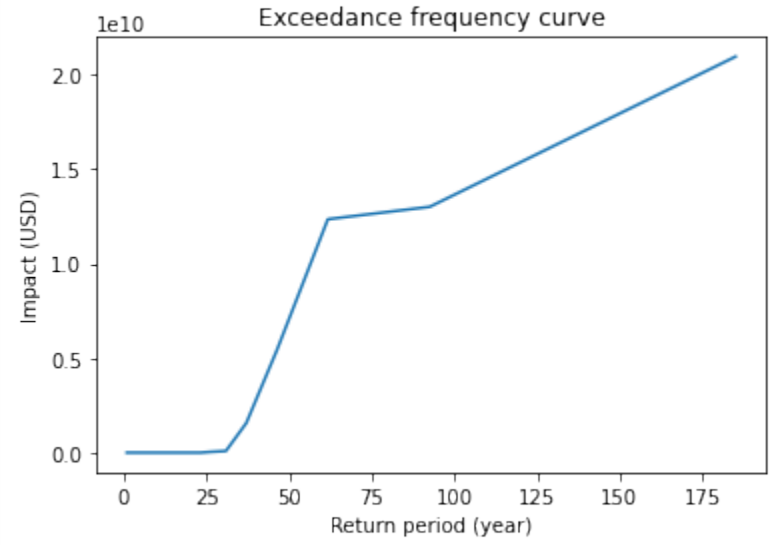
```
  return _prepare_from_string(" ".join(pjargs))
```

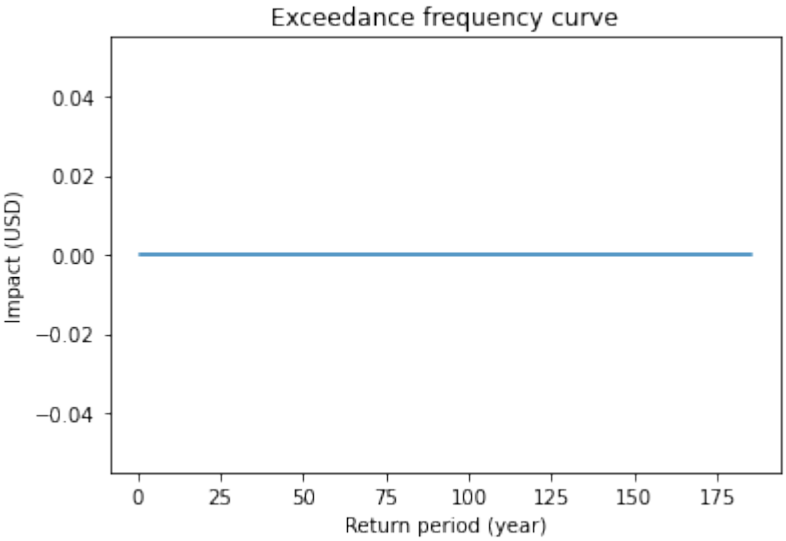
[3]:
```
<AxesSubplot:title={'center':'Exceedance frequency curve'}, xlabel='Return period (year)
↪', ylabel='Impact (USD)'>
```




TC max intensity at each point
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
70
60
50
40
30
20
Intensity (m/s)


TC max intensity at each point
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
Intensity (m/s)
0
10
20
30
40
50
60
70


Exceedance frequency curve
1e10
Impact (USD)
0.0
0.5
1.0
1.5
2.0
0
25
50
75
100
125
150
175
Return period (year)



[4]:
```
# effect of risk_transf_attach and risk_transf_cover
import numpy as np
from climada.entity import ImpactFuncSet, ImpfTropCyclone, Exposures
from climada.entity.measures import Measure
from climada.hazard import Hazard
from climada.engine import Impact

from climada.util import HAZ_DEMO_H5, EXP_DEMO_H5

# define measure
meas = Measure()
meas.name = 'Insurance'
meas.haz_type = 'TC'
meas.color_rgb = np.array([1, 1, 1])
meas.cost = 500000000
meas.risk_transf_attach = 5.0e8
meas.risk_transf_cover = 1.0e9

# impact functions
impf_tc = ImpfTropCyclone.from_emanuel_usa()
impf_all = ImpactFuncSet()
impf_all.append(impf_tc)

# Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
haz.check()

# Exposures
exp = Exposures.from_hdf5(EXP_DEMO_H5)
exp.check()

# impact before
imp = Impact()
imp.calc(exp, impf_all, haz)
```

(continues on next page)

(continued from previous page)

```
imp.calc_freq_curve().plot()

# impact after. risk_transf will be added to the cost of the measure
imp_new, risk_transf = meas.calc_impact(exp, impf_all, haz)
imp_new.calc_freq_curve().plot()
print('risk_transfer {:.3}'.format(risk_transf.aai_agg))
```

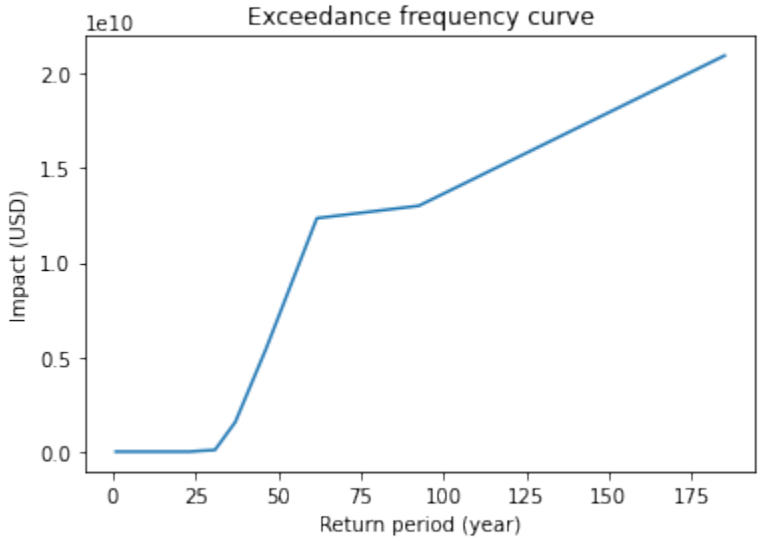
```
2021-04-23 15:54:19,137 - climada.hazard.base - INFO - Reading /Users/zeliestalhanske/
↪climada/demo/data/tc_fl_1990_2004.h5
2021-04-23 15:54:19,169 - climada.entity.exposures.base - INFO - meta set to default␣
↪value {}
2021-04-23 15:54:19,169 - climada.entity.exposures.base - INFO - tag set to default␣
↪value  File:
 Description:
2021-04-23 15:54:19,170 - climada.entity.exposures.base - INFO - ref_year set to default␣
↪value 2018
2021-04-23 15:54:19,170 - climada.entity.exposures.base - INFO - value_unit set to␣
↪default value USD
2021-04-23 15:54:19,171 - climada.entity.exposures.base - INFO - crs set to default␣
↪value: EPSG:4326
2021-04-23 15:54:19,183 - climada.entity.exposures.base - INFO - Reading /Users/
↪zeliestalhanske/climada/demo/data/exp_demo_today.h5
2021-04-23 15:54:19,201 - climada.entity.exposures.base - INFO - meta set to default␣
↪value {}
2021-04-23 15:54:19,202 - climada.entity.exposures.base - INFO - tag set to default␣
↪value  File:
 Description:
2021-04-23 15:54:19,202 - climada.entity.exposures.base - INFO - ref_year set to default␣
↪value 2018
2021-04-23 15:54:19,203 - climada.entity.exposures.base - INFO - value_unit set to␣
↪default value USD
2021-04-23 15:54:19,209 - climada.entity.exposures.base - INFO - crs set to default␣
↪value: EPSG:4326
2021-04-23 15:54:19,221 - climada.entity.exposures.base - INFO - centr_ not set.
2021-04-23 15:54:19,223 - climada.entity.exposures.base - INFO - Matching 50 exposures␣
↪with 2500 centroids.
2021-04-23 15:54:19,231 - climada.engine.impact - INFO - Calculating damage for 50␣
↪assets (>0) and 216 events.
2021-04-23 15:54:19,255 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-23 15:54:19,258 - climada.engine.impact - INFO - Calculating damage for 50␣
↪assets (>0) and 216 events.
risk_transfer 2.7e+07
```

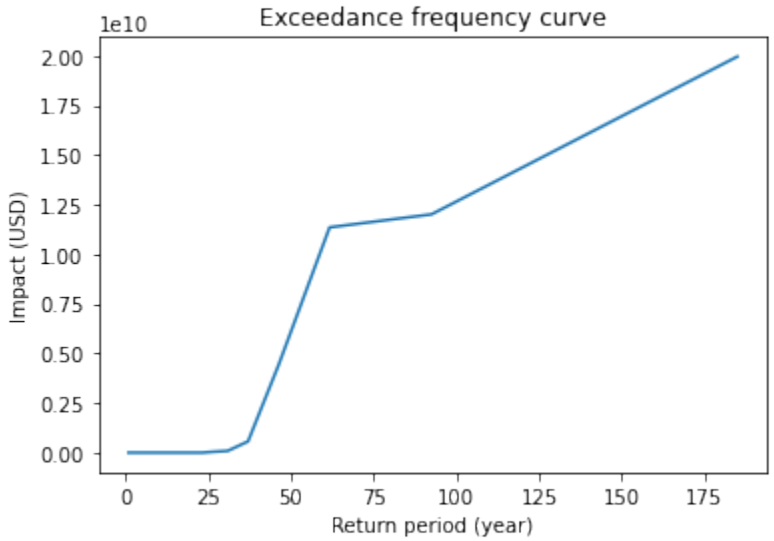
```
/Users/zeliestalhanske/miniconda3/envs/climada_env/lib/python3.8/site-packages/pyproj/
↪crs/crs.py:53: FutureWarning: '+init=<authority>:<code>' syntax is deprecated. '
↪<authority>:<code>' is the preferred initialization method. When making the change, be␣
↪mindful of axis order changes: https://pyproj4.github.io/pyproj/stable/gotchas.html
↪#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
/Users/zeliestalhanske/python_projects/climada_python/climada/entity/exposures/base.py:
↪221: FutureWarning: Assigning CRS to a GeoDataFrame without a geometry column is now␣
↪deprecated and will not be supported in the future.
```

(continues on next page)

(continued from previous page)

```
self.gdf.crs = self.meta['crs']
```





5.7.2 MeasureSet class

Similarly to the `ImpactFuncSet`, `MeasureSet` is a container which handles `Measure` instances through the methods `append()`, `extend()`, `remove_measure()`and `get_measure()`. Use the `check()` method to make sure all the measures have been propoerly set.

`MeasureSet` contains the attribute `tag` of type `Tag`, which stores information about the data: the original file name and a description.

[5]:
```
from climada.entity import MeasureSet
help(MeasureSet)
```

```
Help on class MeasureSet in module climada.entity.measures.measure_set:
```

(continues on next page)

(continued from previous page)

```
class MeasureSet(builtins.object)
 |  Contains measures of type Measure. Loads from
 |  files with format defined in FILE_EXT.
 |
 |  Attributes
 |  ----------
 |  tag : Tag
 |      information about the source data
 |  _data : dict
 |      cotains Measure classes. It's not suppossed to be
 |      directly accessed. Use the class methods instead.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Empty initialization.
 |
 |      Examples
 |      --------
 |      Fill MeasureSet with values and check consistency data:
 |
 |      >>> act_1 = Measure()
 |      >>> act_1.name = 'Seawall'
 |      >>> act_1.color_rgb = np.array([0.1529, 0.2510, 0.5451])
 |      >>> act_1.hazard_intensity = (1, 0)
 |      >>> act_1.mdd_impact = (1, 0)
 |      >>> act_1.paa_impact = (1, 0)
 |      >>> meas = MeasureSet()
 |      >>> meas.append(act_1)
 |      >>> meas.tag.description = "my dummy MeasureSet."
 |      >>> meas.check()
 |
 |      Read measures from file and checks consistency data:
 |
 |      >>> meas = MeasureSet.from_excel(ENT_TEMPLATE_XLS)
 |
 |  append(self, meas)
 |      Append an Measure. Override if same name and haz_type.
 |
 |      Parameters
 |      ----------
 |      meas : Measure
 |          Measure instance
 |
 |      Raises
 |      ------
 |      ValueError
 |
 |  check(self)
 |      Check instance attributes.
 |
 |      Raises
```

(continues on next page)

(continued from previous page)

```
|      ------
|      ValueError
|
|  clear(self)
|      Reinitialize attributes.
|
|  extend(self, meas_set)
|      Extend measures of input MeasureSet to current
|      MeasureSet. Overwrite Measure if same name and haz_type.
|
|      Parameters
|      ----------
|      impact_funcs : MeasureSet
|          ImpactFuncSet instance to extend
|
|      Raises
|      ------
|      ValueError
|
|  get_hazard_types(self, meas=None)
|      Get measures hazard types contained for the name provided.
|      Return all hazard types if no input name.
|
|      Parameters
|      ----------
|      name : str, optional
|          measure name
|
|      Returns
|      -------
|      list(str)
|
|  get_measure(self, haz_type=None, name=None)
|      Get ImpactFunc(s) of input hazard type and/or id.
|      If no input provided, all impact functions are returned.
|
|      Parameters
|      ----------
|      haz_type : str, optional
|          hazard type
|      name : str, optional
|          measure name
|
|      Returns
|      -------
|      Measure (if haz_type and name),
|      list(Measure) (if haz_type or name),
|      {Measure.haz_type : {Measure.name : Measure}} (if None)
|
|  get_names(self, haz_type=None)
|      Get measures names contained for the hazard type provided.
|      Return all names for each hazard type if no input hazard type.
```

(continues on next page)

(continued from previous page)

```
 |
 |      Parameters
 |      ----------
 |      haz_type : str, optional
 |          hazard type from which to obtain the names
 |
 |      Returns
 |      -------
 |      list(Measure.name) (if haz_type provided),
 |      {Measure.haz_type : list(Measure.name)} (if no haz_type)
 |
 |  read_excel(self, *args, **kwargs)
 |      This function is deprecated, use MeasureSet.from_excel instead.
 |
 |  read_mat(self, *args, **kwargs)
 |      This function is deprecated, use MeasureSet.from_mat instead.
 |
 |  remove_measure(self, haz_type=None, name=None)
 |      Remove impact function(s) with provided hazard type and/or id.
 |      If no input provided, all impact functions are removed.
 |
 |      Parameters
 |      ----------
 |      haz_type : str, optional
 |          all impact functions with this hazard
 |      name : str, optional
 |          measure name
 |
 |  size(self, haz_type=None, name=None)
 |      Get number of measures contained with input hazard type and
 |      /or id. If no input provided, get total number of impact functions.
 |
 |      Parameters
 |      ----------
 |      haz_type : str, optional
 |          hazard type
 |      name : str, optional
 |          measure name
 |
 |      Returns
 |      -------
 |      int
 |
 |  write_excel(self, file_name, var_names={'sheet_name': 'measures', 'col_name': {'name
↪': 'name', 'color': 'color', 'cost': 'cost', 'haz_int_a': 'hazard intensity impact a',
↪'haz_int_b': 'hazard intensity impact b', 'haz_frq': 'hazard high frequency cutoff',
↪'haz_set': 'hazard event set', 'mdd_a': 'MDD impact a', 'mdd_b': 'MDD impact b', 'paa_a
↪': 'PAA impact a', 'paa_b': 'PAA impact b', 'fun_map': 'damagefunctions map', 'exp_set
↪': 'assets file', 'exp_reg': 'Region_ID', 'risk_att': 'risk transfer attachement',
↪'risk_cov': 'risk transfer cover', 'risk_fact': 'risk transfer cost factor', 'haz':
↪'peril_ID'}})
 |      Write excel file following template.
```

(continues on next page)

(continued from previous page)

```
 |
 |      Parameters
 |      ----------
 |      file_name : str
 |          absolute file name to write
 |      var_names : dict, optional
 |          name of the variables in the file
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_excel(file_name, description='', var_names={'sheet_name': 'measures', 'col_name
↪': {'name': 'name', 'color': 'color', 'cost': 'cost', 'haz_int_a': 'hazard intensity␣
↪impact a', 'haz_int_b': 'hazard intensity impact b', 'haz_frq': 'hazard high frequency␣
↪cutoff', 'haz_set': 'hazard event set', 'mdd_a': 'MDD impact a', 'mdd_b': 'MDD impact b
↪', 'paa_a': 'PAA impact a', 'paa_b': 'PAA impact b', 'fun_map': 'damagefunctions map',
↪'exp_set': 'assets file', 'exp_reg': 'Region_ID', 'risk_att': 'risk transfer␣
↪attachement', 'risk_cov': 'risk transfer cover', 'risk_fact': 'risk transfer cost␣
↪factor', 'haz': 'peril_ID'}}) from builtins.type
 |      Read excel file following template and store variables.
 |
 |      Parameters
 |      ----------
 |      file_name : str
 |          absolute file name
 |      description : str, optional
 |          description of the data
 |      var_names : dict, optional
 |          name of the variables in the file
 |
 |      Returns
 |      -------
 |      meas_set : climada.entity.MeasureSet
 |          Measures set from Excel
 |
 |  from_mat(file_name, description='', var_names={'sup_field_name': 'entity', 'field_
↪name': 'measures', 'var_name': {'name': 'name', 'color': 'color', 'cost': 'cost', 'haz_
↪int_a': 'hazard_intensity_impact_a', 'haz_int_b': 'hazard_intensity_impact_b', 'haz_frq
↪': 'hazard_high_frequency_cutoff', 'haz_set': 'hazard_event_set', 'mdd_a': 'MDD_impact_
↪a', 'mdd_b': 'MDD_impact_b', 'paa_a': 'PAA_impact_a', 'paa_b': 'PAA_impact_b', 'fun_map
↪': 'damagefunctions_map', 'exp_set': 'assets_file', 'exp_reg': 'Region_ID', 'risk_att':
↪ 'risk_transfer_attachement', 'risk_cov': 'risk_transfer_cover', 'haz': 'peril_ID'}})␣
↪from builtins.type
 |      Read MATLAB file generated with previous MATLAB CLIMADA version.
 |
 |      Parameters
 |      ----------
 |      file_name : str
 |          absolute file name
 |      description : str, optional
 |          description of the data
 |      var_names : dict, optional
```

(continues on next page)

(continued from previous page)

```
|          name of the variables in the file
|
|      Returns
|      -------
|      meas_set: climada.entity.MeasureSet()
|          Measure Set from matlab file
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

[6]:
```
# build measures
import numpy as np
import matplotlib.pyplot as plt
from climada.entity.measures import Measure, MeasureSet

meas_1 = Measure()
meas_1.haz_type = 'TC'
meas_1.name = 'Mangrove'
meas_1.color_rgb = np.array([1, 1, 1])
meas_1.cost = 500000000
meas_1.mdd_impact = (1, 2)
meas_1.paa_impact = (1, 2)
meas_1.hazard_inten_imp = (1, 2)
meas_1.risk_transf_cover = 500

meas_2 = Measure()
meas_2.haz_type = 'TC'
meas_2.name = 'Sandbags'
meas_2.color_rgb = np.array([1, 1, 1])
meas_2.cost = 22000000
meas_2.mdd_impact = (1, 2)
meas_2.paa_impact = (1, 3)
meas_2.hazard_inten_imp = (1, 2)
meas_2.exp_region_id = 2

# gather all measures
meas_set = MeasureSet()
meas_set.append(meas_1)
meas_set.append(meas_2)
meas_set.check()

# select one measure
meas_sel = meas_set.get_measure(name='Sandbags')
print(meas_sel[0].name, meas_sel[0].cost)
```

```
Sandbags 22000000
```

5.7.3 Read measures of an Excel file

Measures defined in an excel file following the template provided in sheet `measures` of `climada_python/data/system/entity_template.xlsx` can be ingested directly using the method `from_excel()`.

[7]:
```
from climada.entity.measures import MeasureSet
from climada.util import ENT_TEMPLATE_XLS

# Fill DataFrame from Excel file
file_name = ENT_TEMPLATE_XLS # provide absolute path of the excel file
meas_set = MeasureSet.from_excel(file_name)
print('Read file:', meas_set.tag.file_name)
```

```
Read file: $CLIMADA_DIR/data/entity_template.xlsx
```

5.7.4 Write measures

Measures can be writen in Excel format using `write_excel()` method.

[8]:
```
from climada.entity.measures import MeasureSet
from climada.util import ENT_TEMPLATE_XLS

# Fill DataFrame from Excel file
file_name = ENT_TEMPLATE_XLS # provide absolute path of the excel file
meas_set = MeasureSet.from_excel(file_name)

# write file
meas_set.write_excel('results/tutorial_meas_set.xlsx')
```

Pickle can always be used as well:

[9]:
```
from climada.util.save import save
# this generates a results folder in the current path and stores the output there
save('tutorial_meas_set.p', meas_set)
```

5.8 Hazard class

5.8.1 What is a hazard?

A hazard describes weather events such as storms, floods, droughts, or heat waves both in terms of probability of occurrence as well as physical intensity.

5.8.2 How are hazards embedded in the CLIMADA architecture?

Hazards are defined by the base class `Hazard` which gathers the required attributes that enable the impact computation (such as centroids, frequency per event, and intensity per event and centroid) and common methods such as readers and visualization functions. Each hazard class collects historical data or model simulations and transforms them, if necessary, in order to construct a coherent event database. Stochastic events can be generated taking into account the frequency and main intensity characteristics (such as local water depth for floods or gust speed for storms) of historical events, producing an ensemble of probabilistic events for each historical event. CLIMADA provides therefore an event-based probabilistic approach which does not depend on a hypothesis of a priori general probability distribution choices. The source of the historical data (e.g. inventories or satellite images) or model simulations (e.g. synthetic tropical cyclone tracks) and the methodologies used to compute the hazard attributes and its stochastic events depend on each hazard type and are defined in its corresponding Hazard-derived class (e.g. `TropCylcone` for tropical cyclones, explained in the tutorial *TropCyclone*). This procedure provides a solid and homogeneous methodology to compute impacts worldwide. In the case where the risk analysis comprises a specific region where good quality data or models describing the hazard intensity and frequency are available, these can be directly ingested by the platform through the reader functions, skipping the hazard modelling part (in total or partially), and allowing us to easily and seamlessly combine CLIMADA with external sources. Hence the impact model can be used for a wide variety of applications, e.g. deterministically to assess the impact of a single (past or future) event or to quantify risk based on a (large) set of probabilistic events. Note that since the `Hazard` class is not an abstract class, any hazard that is not defined in CLIMADA can still be used by providing the `Hazard` attributes.

5.8.3 What do hazards look like in CLIMADA?

A `Hazard` contains events of some hazard type defined at `centroids`. There are certain variables in a `Hazard` instance that *are needed* to compute the impact, while others are *descriptive* and can therefore be set with default values. The full list of looks like this:

Mandatory variables	Data Type	Description
tag	`TagHazard()`	information about the source
units	(str)	units of the intensity
centroids	`Centroids()`	centroids of the events
event_id	(np.array)	id (>0) of each event
frequency	(np.array)	frequency of each event in years
intensity	(sparse.csr_matrix)	intensity of the events at centroids
fraction	(sparse.csr_matrix)	fraction of affected exposures for each event at each centroid

Descriptive variables	Data Type	Description
date	(np.array)	integer date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1 (ordinal format of datetime library)
orig	(np.array)	flags indicating historical events (True) or probabilistic (False)
event_name	(list(str))	name of each event (default: event_id)

Note that `intensity` and `fraction` are scipy.sparse matrices of size num_events x num_centroids. The `Centroids` class contains the geographical coordinates where the hazard is defined. A `Centroids` instance provides the coordinates either as points or raster data together with their Coordinate Reference System (CRS). The default CRS used in climada is the usual EPSG:4326. `Centroids` provides moreover methods to compute centroids areas, on land mask, country iso mask or distance to coast.

How is this tutorial structured?

Part 1: Read hazards from raster data

Part 2: Read hazards from other data

Part 3: Define hazards manually

Part 4: Analyse hazards

Part 5: Visualize hazards

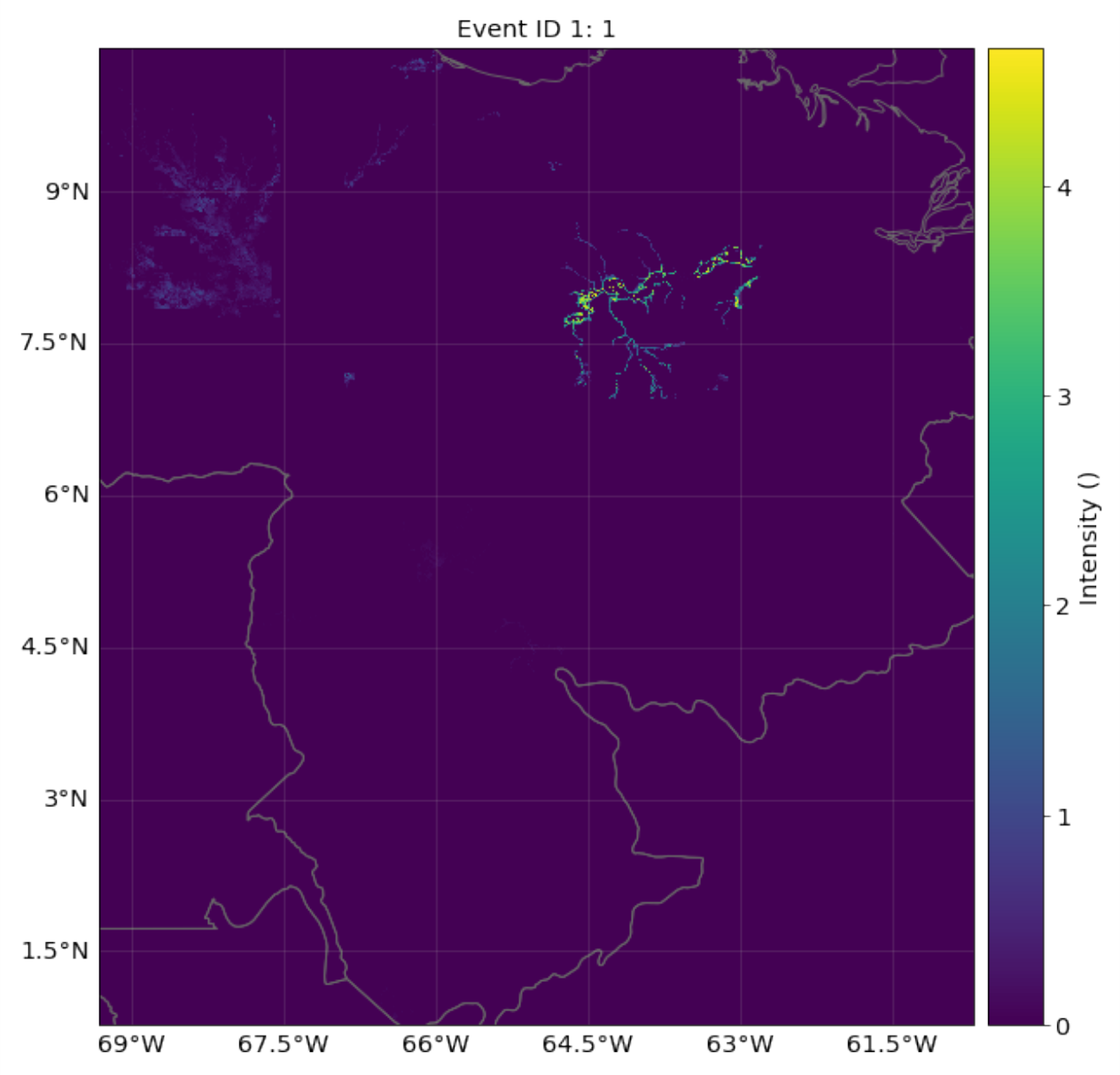
Part 6: Write (=save) hazards

Part 1: Read hazards from raster data

Raster data can be read in any format accepted by rasterio using `Hazard`'s `from_raster()` method. The raster information might refer to the `intensity` or `fraction`of the hazard. Different configuration options such as transforming the coordinates, changing the CRS and reading only a selected area or band are available through the `from_raster()` arguments as follows:

[1]:
```
%matplotlib inline
import numpy as np
from climada.hazard import Hazard
from climada.util.constants import HAZ_DEMO_FL

# read intensity from raster file HAZ_DEMO_FL and set frequency for the contained event
haz_ven = Hazard.from_raster([HAZ_DEMO_FL], attrs={'frequency':np.ones(1)/2}, haz_type=
↪'FL')
haz_ven.check()

# The masked values of the raster are set to 0
# Sometimes the raster file does not contain all the information, as in this case the␣
↪mask value -9999
# We mask it manuall and plot it using plot_intensity()
haz_ven.intensity[haz_ven.intensity==-9999] = 0
haz_ven.plot_intensity(1, smooth=False) # if smooth=True (default value) is used, the␣
↪computation time might increase

# per default the following attributes have been set
print('event_id: ', haz_ven.event_id)
print('event_name: ', haz_ven.event_name)
print('date: ', haz_ven.date)
print('frequency: ', haz_ven.frequency)
print('orig: ', haz_ven.orig)
print('min, max fraction: ', haz_ven.fraction.min(), haz_ven.fraction.max())
```

```
event_id:  [1]
event_name:  ['1']
date:  [1.]
frequency:  [0.5]
orig:  [ True]
min, max fraction:  0.0 1.0
```



EXERCISE:

1. Read raster data in EPSG 2201 Coordinate Reference System (CRS)
2. Read raster data in its given CRS and transform it to the affine transformation Affine(0.009000000000000341, 0.0, -69.33714959699981, 0.0, -0.009000000000000341, 10.42822096697894), height=500, width=501)
3. Read raster data in window Window(10, 10, 20, 30)

[2]:
```
# Put your code here
```

[3]:
```
# Solution:

# 1. The CRS can be reprojected using dst_crs option
haz = Hazard.from_raster([HAZ_DEMO_FL], dst_crs='epsg:2201', haz_type='FL')
haz.check()
print('\n Solution 1:')
print('centroids CRS:', haz.centroids.crs)
print('raster info:', haz.centroids.meta)

# 2. Transformations of the coordinates can be set using the transform option and Affine
from rasterio import Affine
haz = Hazard.from_raster([HAZ_DEMO_FL], haz_type='FL',
                         transform=Affine(0.009000000000000341, 0.0, -69.33714959699981,␣
↪\
                                          0.0, -0.009000000000000341, 10.42822096697894),
                         height=500, width=501)
haz.check()
print('\n Solution 2:')
print('raster info:', haz.centroids.meta)
print('intensity size:', haz.intensity.shape)

# 3. A partial part of the raster can be loaded using the window or geometry
from rasterio.windows import Window
haz = Hazard.from_raster([HAZ_DEMO_FL], haz_type='FL', window=Window(10, 10, 20, 30))
haz.check()
print('\n Solution 3:')
print('raster info:', haz.centroids.meta)
print('intensity size:', haz.intensity.shape)
```

```
 Solution 1:
centroids CRS: 'epsg:2201'
raster info: {'driver': 'GSBG', 'dtype': 'float32', 'nodata': 1.7014100091878 28e+38,
↪'width': 978, 'height': 1091, 'count': 1, 'crs': 'epsg:2201', 'transform': Affine(1011.
↪5372910988809, 0.0, 1120744.5486664253,
       0.0, -1011.5372910988809, 1189133.7652687666)}

 Solution 2:
raster info: {'driver': 'GSBG', 'dtype': 'float32', 'nodata': 1.7014100091878 28e+38,
↪'width': 501, 'height': 500, 'count': 1, 'crs': CRS.from_epsg(4326), 'transform':␣
↪Affine(0.009000000000000341, 0.0, -69.33714959699981,
       0.0, -0.009000000000000341, 10.42822096697894)}
intensity size: (1, 250500)

 Solution 3:
raster info: {'driver': 'GSBG', 'dtype': 'float32', 'nodata': 1.7014100091878 28e+38,
↪'width': 20, 'height': 30, 'count': 1, 'crs': CRS.from_epsg(4326), 'transform':␣
↪Affine(0.009000000000000341, 0.0, -69.2471495969998,
       0.0, -0.009000000000000341, 10.338220966978936)}
intensity size: (1, 600)
```

Part 2: Read hazards from other data

- excel: Hazards can be read from Excel files following the template in `climada_python/data/system/`

hazard_template.xlsx using the from_excel() method.

- MATLAB: Hazards generated with CLIMADA's MATLAB version (.mat format) can be read using from_mat().
- vector data: Use Hazard's from_vector-constructor to read shape data (all formats supported by fiona).
- hdf5: Hazards generated with the CLIMADA in Python (.h5 format) can be read using from_hdf5().

[4]:
```
from climada.hazard import Hazard, Centroids
from climada.util import HAZ_DEMO_H5 # CLIMADA's Python file
# Hazard needs to know the acronym of the hazard type to be constructed!!! Use 'NA' if␣
↪not known.
haz_tc_fl = Hazard.from_hdf5(HAZ_DEMO_H5)  # Historic tropical cyclones in Florida from␣
↪1990 to 2004
haz_tc_fl.check() # Use always the check() method to see if the hazard has been loaded␣
↪correctly
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

Part 3: Define hazards manually A Hazard can be defined by filling its values one by one, as follows:

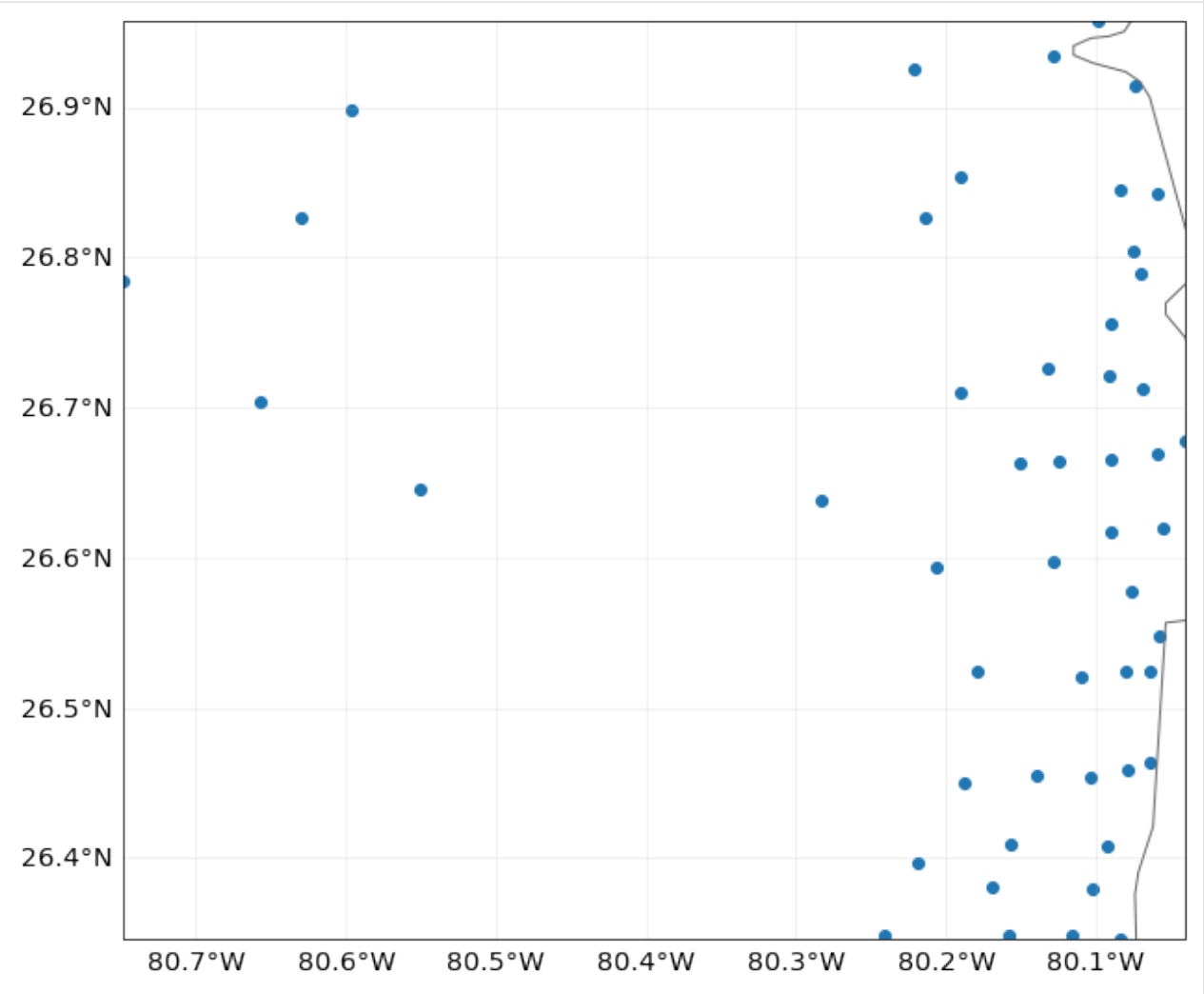
[5]:
```
# setting points
import numpy as np
from scipy import sparse

lat = np.array([26.933899, 26.957203, 26.783846, 26.645524, 26.897796, 26.925359, \
       26.914768, 26.853491, 26.845099, 26.82651 , 26.842772, 26.825905, \
       26.80465 , 26.788649, 26.704277, 26.71005 , 26.755412, 26.678449, \
       26.725649, 26.720599, 26.71255 , 26.6649  , 26.664699, 26.663149, \
       26.66875 , 26.638517, 26.59309 , 26.617449, 26.620079, 26.596795, \
       26.577049, 26.524585, 26.524158, 26.523737, 26.520284, 26.547349, \
       26.463399, 26.45905 , 26.45558 , 26.453699, 26.449999, 26.397299, \
       26.4084  , 26.40875 , 26.379113, 26.3809  , 26.349068, 26.346349, \
       26.348015, 26.347957])

lon = np.array([-80.128799, -80.098284, -80.748947, -80.550704, -80.596929, \
       -80.220966, -80.07466 , -80.190281, -80.083904, -80.213493, \
       -80.0591  , -80.630096, -80.075301, -80.069885, -80.656841, \
       -80.190085, -80.08955 , -80.041179, -80.1324  , -80.091746, \
       -80.068579, -80.090698, -80.1254  , -80.151401, -80.058749, \
       -80.283371, -80.206901, -80.090649, -80.055001, -80.128711, \
       -80.076435, -80.080105, -80.06398 , -80.178973, -80.110519, \
       -80.057701, -80.064251, -80.07875 , -80.139247, -80.104316, \
       -80.188545, -80.21902 , -80.092391, -80.1575  , -80.102028, \
       -80.16885 , -80.116401, -80.08385 , -80.241305, -80.158855])

n_cen = lon.size # number of centroids
n_ev = 10 # number of events

haz = Hazard('TC')
haz.centroids = Centroids.from_lat_lon(lat, lon) # default crs used
```

(continues on next page)

(continued from previous page)

```
haz.intensity = sparse.csr_matrix(np.random.random((n_ev, n_cen)))
haz.units = 'm'
haz.event_id = np.arange(n_ev, dtype=int)
haz.event_name = ['ev_12', 'ev_21', 'Maria', 'ev_35', 'Irma', 'ev_16', 'ev_15', 'Edgar',
↪'ev_1', 'ev_9']
haz.date = [721166, 734447, 734447, 734447, 721167, 721166, 721167, 721200, 721166,␣
↪721166]
haz.orig = np.zeros(n_ev, bool)
haz.frequency = np.ones(n_ev)/n_ev
haz.fraction = haz.intensity.copy()
haz.fraction.data.fill(1)
haz.check()
haz.centroids.plot()
```

[5]:
```
<GeoAxesSubplot:>
```



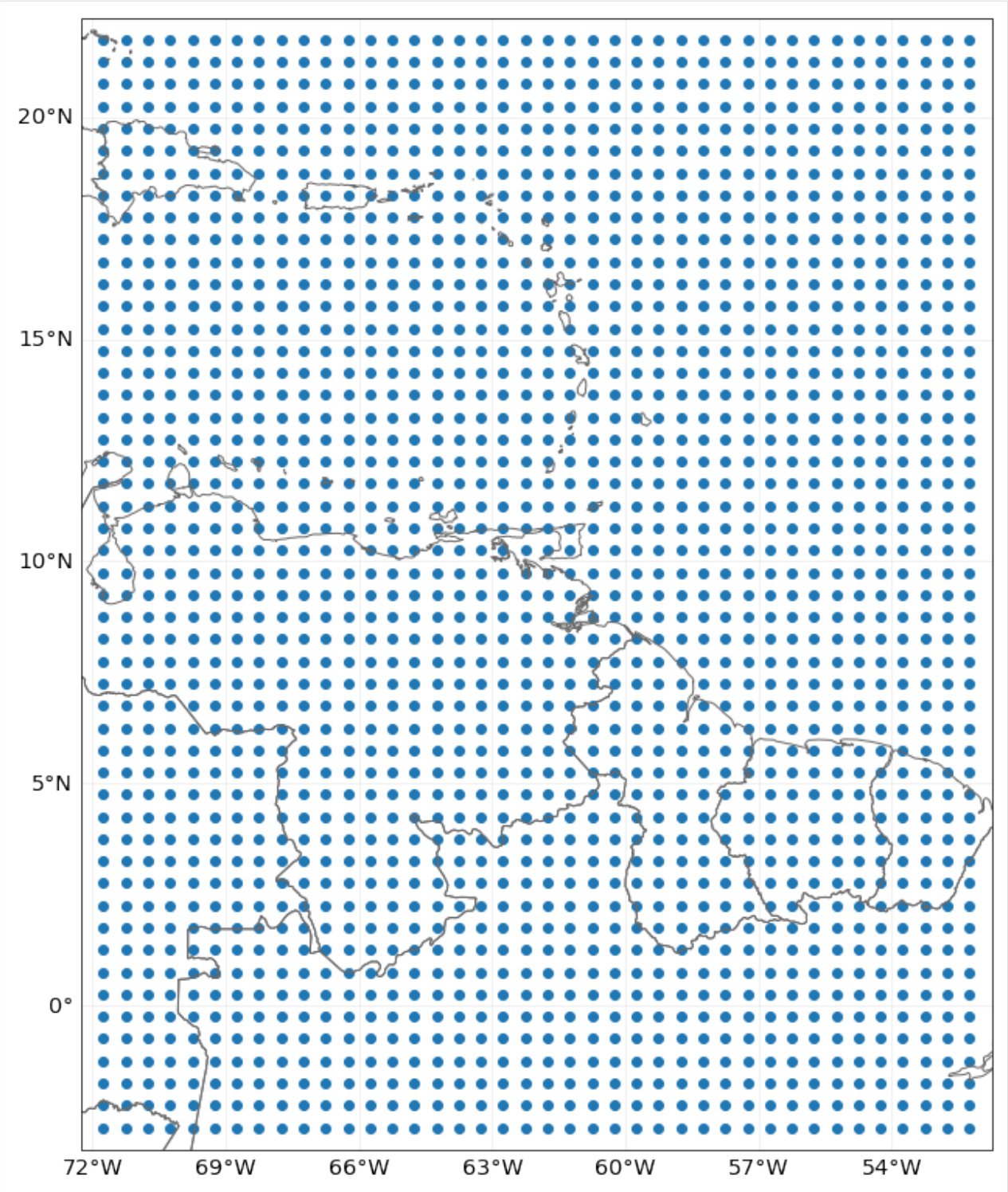
[6]:
```
# setting raster
import numpy as np
from scipy import sparse
```

(continues on next page)

(continued from previous page)

```
# raster info:
# border upper left corner (of the pixel, not of the center of the pixel)
xf_lat = 22
xo_lon = -72
# resolution in lat and lon
d_lat = -0.5 # negative because starting in upper corner
d_lon = 0.5 # same step as d_lat
# number of points
n_lat = 50
n_lon = 40

n_ev = 10 # number of events

haz = Hazard('TC')
haz.centroids = Centroids.from_pix_bounds(xf_lat, xo_lon, d_lat, d_lon, n_lat, n_lon) #␣
↪default crs used
haz.intensity = sparse.csr_matrix(np.random.random((n_ev, haz.centroids.size)))
haz.units = 'm'
haz.event_id = np.arange(n_ev, dtype=int)
haz.event_name = ['ev_12', 'ev_21', 'Maria', 'ev_35', 'Irma', 'ev_16', 'ev_15', 'Edgar',
↪'ev_1', 'ev_9']
haz.date = [721166, 734447, 734447, 734447, 721167, 721166, 721167, 721200, 721166,␣
↪721166]
haz.orig = np.zeros(n_ev, bool)
haz.frequency = np.ones(n_ev)/n_ev
haz.fraction = haz.intensity.copy()
haz.fraction.data.fill(1)
haz.check()
print('Check centroids borders:', haz.centroids.total_bounds)
haz.centroids.plot()

# using from_pnt_bounds, the bounds refer to the bounds of the center of the pixel
left, bottom, right, top = xo_lon, -3.0, -52.0, xf_lat
haz.centroids = Centroids.from_pnt_bounds((left, bottom, right, top), 0.5) # default crs␣
↪used
print('Check centroids borders:', haz.centroids.total_bounds)
```

```
Check centroids borders: (-72.0, -3.0, -52.0, 22.0)
Check centroids borders: (-72.25, -3.25, -51.75, 22.25)
```



Part 4: Analyse Hazards

The following methods can be used to analyse the data in `Hazard`:

- `calc_year_set()` method returns a dictionary with all the historical (not synthetic) event ids that happened at each year.

- `get_event_date()` returns strings of dates in ISO format.
- To obtain the relation between event ids and event names, two methods can be used `get_event_name()` and `get_event_id()`.

Other methods to handle one or several Hazards are: - the property `size` returns the number of events contained. - `append()` is used to expand events with data from another `Hazard` (and same centroids). - `select()` returns a new hazard with the selected region, date and/or synthetic or historical filter. - `remove_duplicates()` removes events with same name and date. - `local_exceedance_inten()` returns a matrix with the exceedence frequency at every frequency and provided return periods. This is the one used in `plot_rp_intensity()`. - `reproject_raster()`, `reproject_vector()`, `raster_to_vector()`, `vector_to_raster()` are methods to change centroids' CRS and between raster and vector data.

`Centroids` methods: - centroids properties such as area per pixel, distance to coast, country ISO code, on land mask or elevation are available through different `set_XX()`methods. - `set_lat_lon_to_meta()` computes the raster `meta` dictionary from present lat and lon. `set_meta_to_lat_lon()` computes lat and lon of the center of the pixels described in attribute `meta`. The raster `meta` information contains at least: `width`, `height`, `crs` and `transform` data (use help(Centroids) for more info). Using raster centroids can increase computing performance for several computations. - when using lats and lons (vector data) the `geopandas.GeoSeries` `geometry` attribute contains the CRS information and can be filled with point shapes to perform different computation. The geometry points can be then released using `empty_geometry_points()`.

EXERCISE:

Using the previous hazard `haz_tc_fl` answer these questions: 1. How many synthetic events are contained? 2. Generate a hazard with historical hurricanes ocurring between 1995 and 2001. 3. How many historical hurricanes occured in 1999? Which was the year with most hurricanes between 1995 and 2001? 4. What is the number of centroids with distance to coast smaller than 1km?

[7]:
```
# Put your code here:
```

[8]:
```
#help(hist_tc.centroids)
```

[9]:
```
# SOLUTION:

# 1.How many synthetic events are contained?
print('Number of total events:', haz_tc_fl.size)
print('Number of synthetic events:', np.logical_not(haz_tc_fl.orig).astype(int).sum())

# 2. Generate a hazard with historical hurricanes ocurring between 1995 and 2001.
hist_tc = haz_tc_fl.select(date=('1995-01-01', '2001-12-31'), orig=True)
print('Number of historical events between 1995 and 2001:', hist_tc.size)

# 3. How many historical hurricanes occured in 1999? Which was the year with most␣
↪hurricanes between 1995 and 2001?
ev_per_year = hist_tc.calc_year_set() # events ids per year
print('Number of events in 1999:', ev_per_year[1999].size)
max_year = 1995
max_ev = ev_per_year[1995].size
for year, ev in ev_per_year.items():
    if ev.size > max_ev:
        max_year = year
```

(continues on next page)

(continued from previous page)

```
print('Year with most hurricanes between 1995 and 2001:', max_year)

# 4. What is the number of centroids with distance to coast smaller than 1km?
hist_tc.centroids.set_dist_coast()
num_cen_coast = np.argwhere(hist_tc.centroids.dist_coast < 1000).size
print('Number of centroids close to coast: ', num_cen_coast)
```

```
Number of total events: 216
Number of synthetic events: 0
Number of historical events between 1995 and 2001: 109
Number of events in 1999: 16
Year with most hurricanes between 1995 and 2001: 1995
```

```
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
$CONDA_PREFIX/lib/python3.8/site-packages/pyproj/crs/crs.py:68: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

```
Number of centroids close to coast:  41
```

Part 5: Visualize Hazards

There are three different plot functions: `plot_intensity()`, `plot_fraction()`and `plot_rp_intensity()`. Depending on the inputs, different properties can be visualized. Check the documentation of the functions:

[10]:
```
help(haz_tc_fl.plot_intensity)
help(haz_tc_fl.plot_rp_intensity)
```

```
Help on method plot_intensity in module climada.hazard.base:

plot_intensity(event=None, centr=None, smooth=True, axis=None, adapt_fontsize=True,␣
↪**kwargs) method of climada.hazard.base.Hazard instance
    Plot intensity values for a selected event or centroid.

    Parameters
    ----------
    event: int or str, optional
        If event > 0, plot intensities of
        event with id = event. If event = 0, plot maximum intensity in
```

(continues on next page)

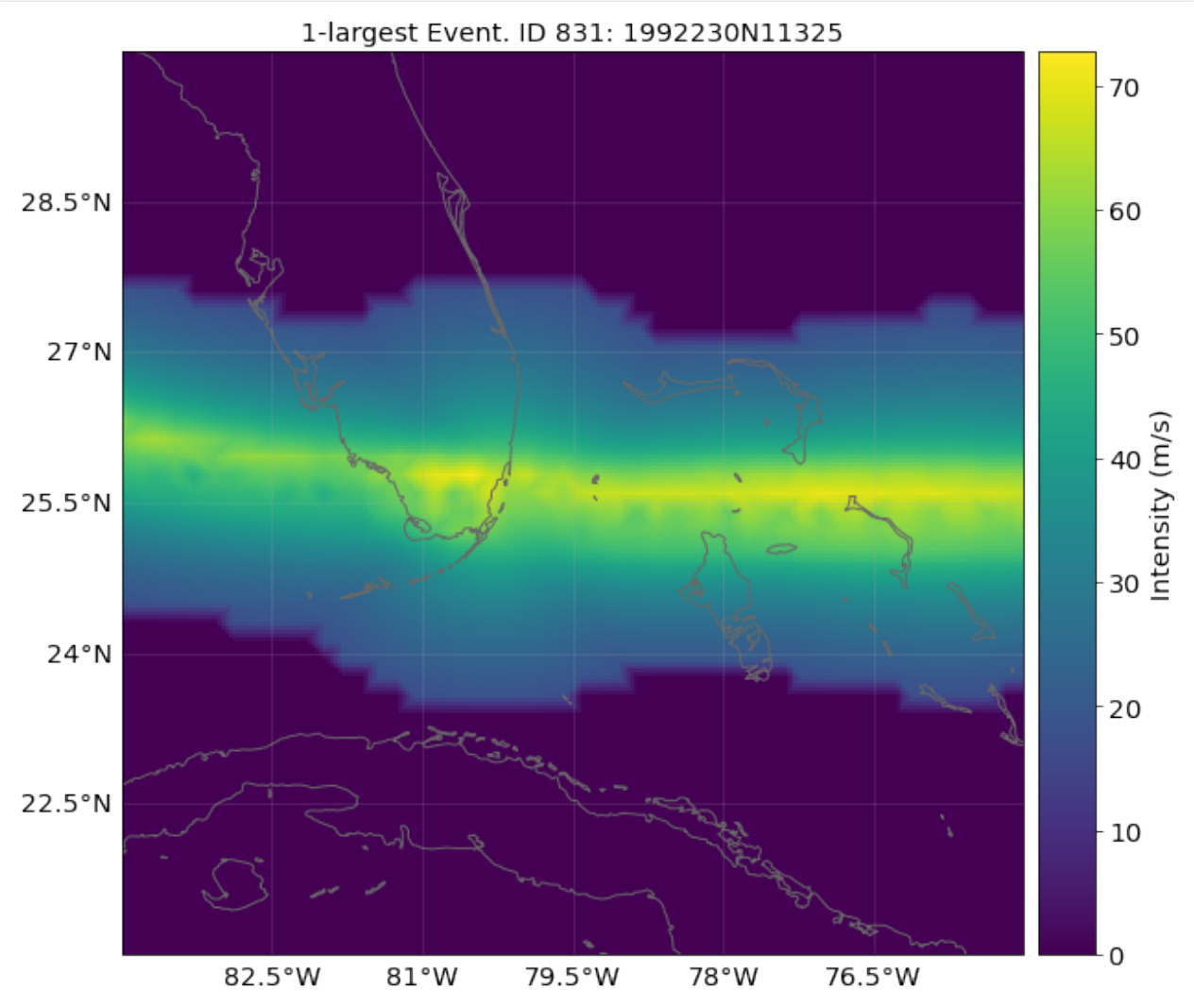
(continued from previous page)

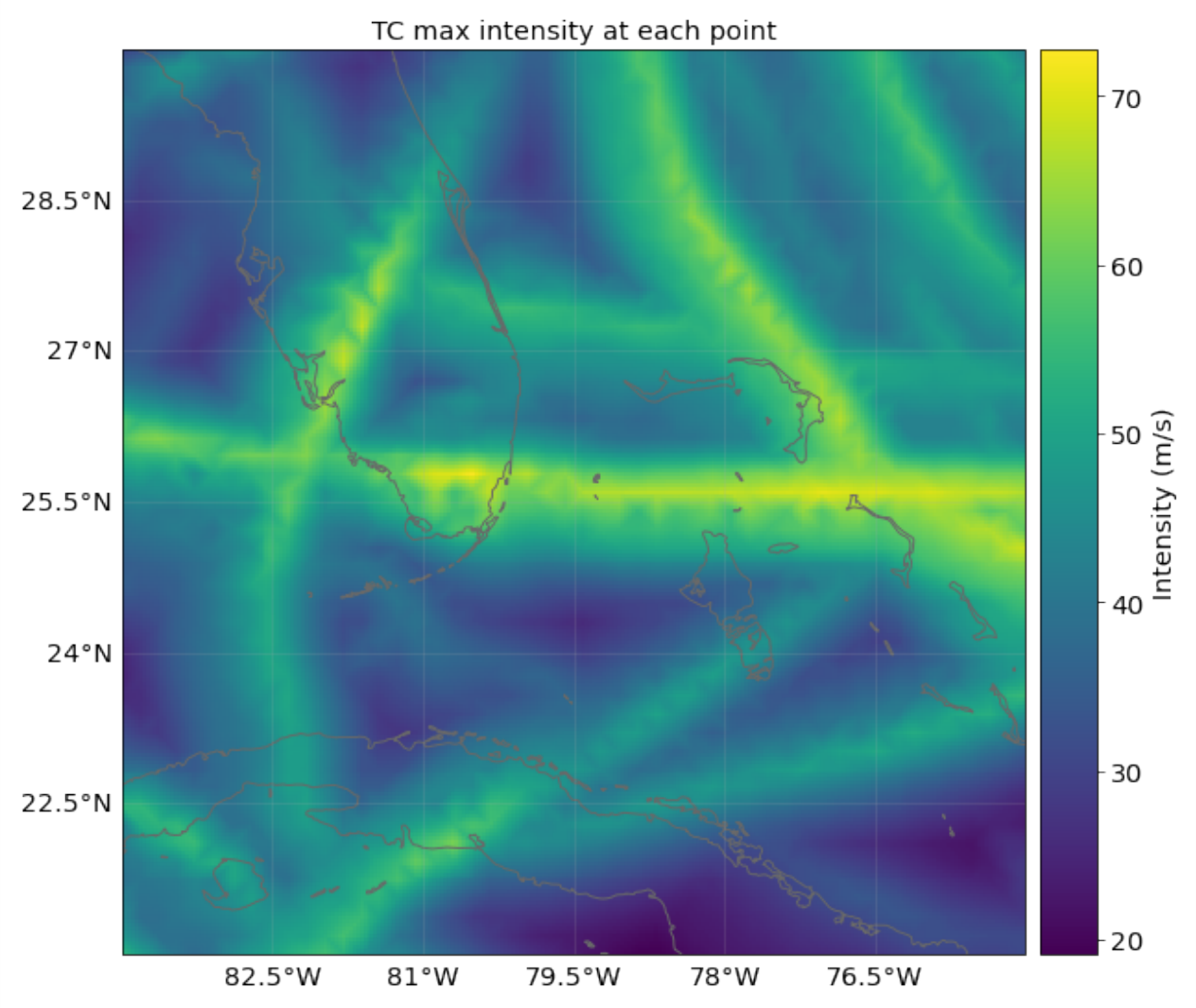
```
        each centroid. If event < 0, plot abs(event)-largest event. If
        event is string, plot events with that name.
    centr: int or tuple, optional
        If centr > 0, plot intensity
        of all events at centroid with id = centr. If centr = 0,
        plot maximum intensity of each event. If centr < 0,
        plot abs(centr)-largest centroid where higher intensities
        are reached. If tuple with (lat, lon) plot intensity of nearest
        centroid.
    smooth: bool, optional
        Rescale data to RESOLUTIONxRESOLUTION pixels (see constant
        in module `climada.util.plot`)
    axis: matplotlib.axes._subplots.AxesSubplot, optional
        axis to use
    kwargs: optional
        arguments for pcolormesh matplotlib function
        used in event plots or for plot function used in centroids plots

    Returns
    -------
        matplotlib.axes._subplots.AxesSubplot

    Raises
    ------
        ValueError

Help on method plot_rp_intensity in module climada.hazard.base:

plot_rp_intensity(return_periods=(25, 50, 100, 250), smooth=True, axis=None, figsize=(9,␣
↪13), adapt_fontsize=True, **kwargs) method of climada.hazard.base.Hazard instance
    Compute and plot hazard exceedance intensity maps for different
    return periods. Calls local_exceedance_inten.

    Parameters
    ----------
    return_periods: tuple(int), optional
        return periods to consider
    smooth: bool, optional
        smooth plot to plot.RESOLUTIONxplot.RESOLUTION
    axis: matplotlib.axes._subplots.AxesSubplot, optional
        axis to use
    figsize: tuple, optional
        figure size for plt.subplots
    kwargs: optional
        arguments for pcolormesh matplotlib function used in event plots

    Returns
    -------
    axis, inten_stats:  matplotlib.axes._subplots.AxesSubplot, np.ndarray
        intenstats is return_periods.size x num_centroids
```

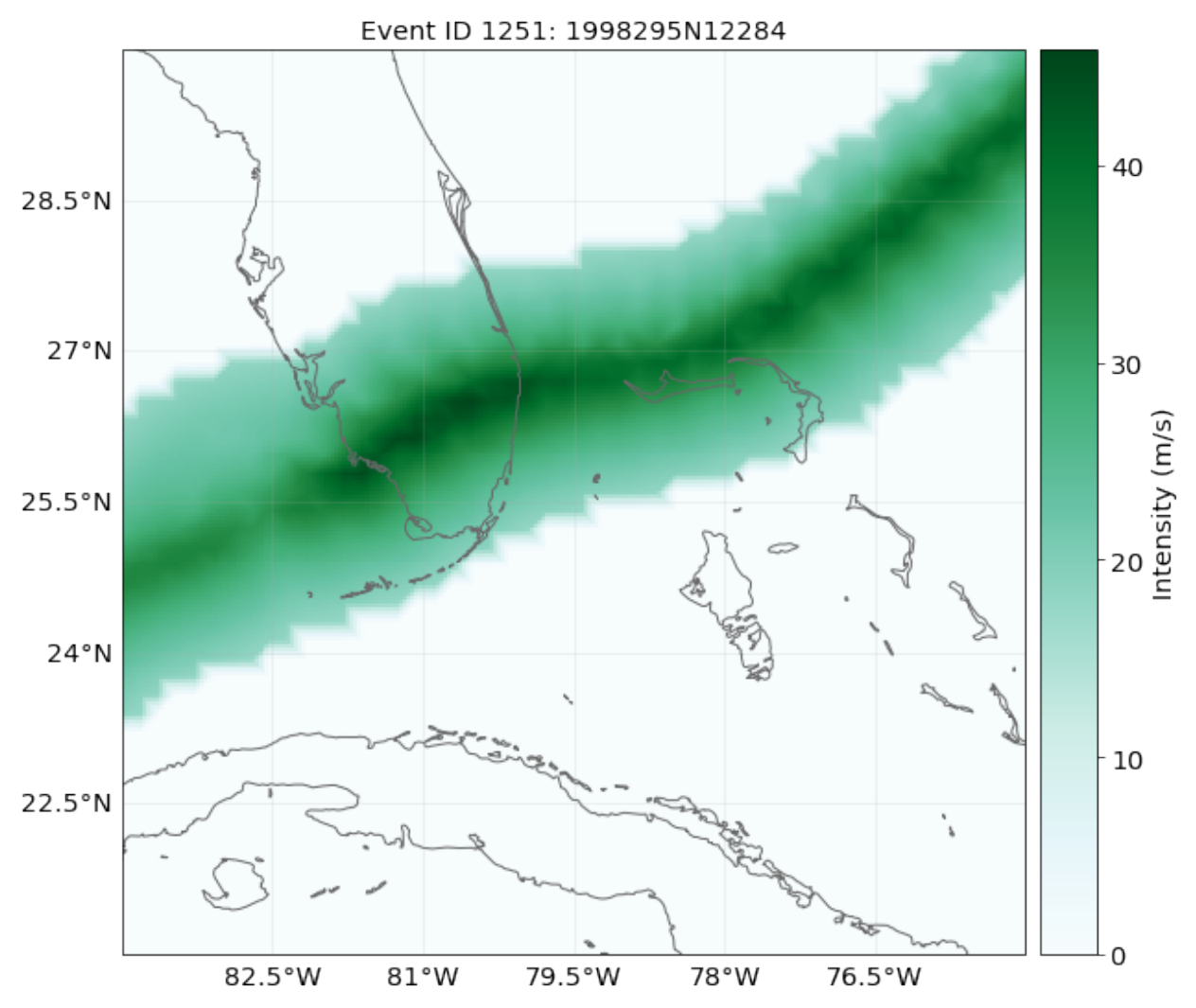
[11]:
```
# 1. intensities of the largest event (defined as greater sum of intensities):
# all events:
haz_tc_fl.plot_intensity(event=-1) # largest historical event: 1992230N11325 hurricane␣
↪ANDREW

# 2. maximum intensities at each centroid:
haz_tc_fl.plot_intensity(event=0)

# 3. intensities of hurricane 1998295N12284:
haz_tc_fl.plot_intensity(event='1998295N12284', cmap='BuGn') # setting color map

# 4. tropical cyclone intensities maps for the return periods [10, 50, 75, 100]
_, res = haz_tc_fl.plot_rp_intensity([10, 50, 75, 100])

# 5. intensities of all the events in centroid with id 50
haz_tc_fl.plot_intensity(centr=50)

# 6. intensities of all the events in centroid closest to lat, lon = (26.5, -81)
haz_tc_fl.plot_intensity(centr=(26.5, -81));
```

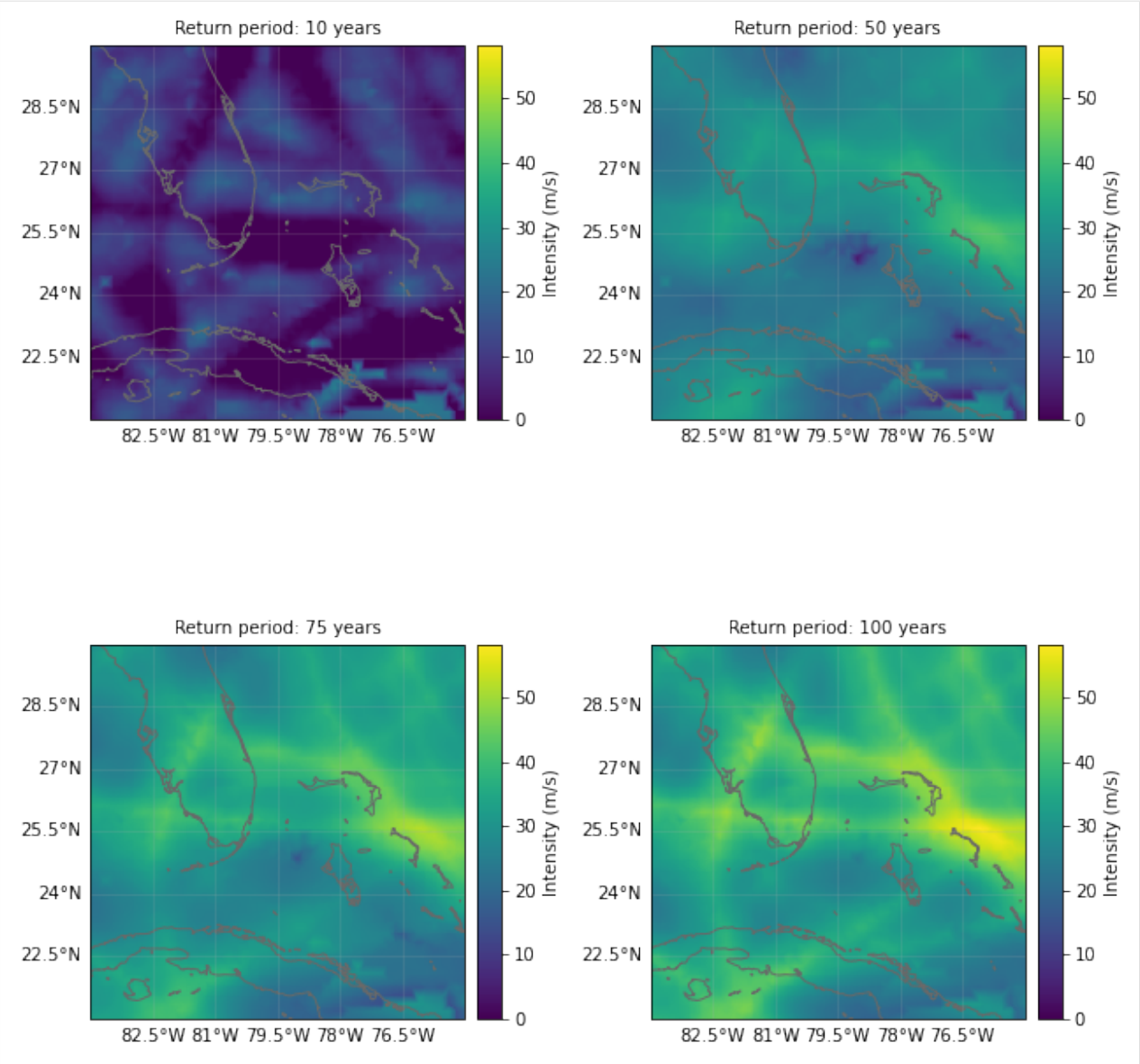
```
2021-06-04 17:07:56,644 - climada.hazard.base - INFO - Computing exceedance intenstiy␣
↪map for return periods: [ 10  50  75 100]
2021-06-04 17:07:57,198 - climada.hazard.base - WARNING - Exceedance intenstiy values␣
↪below 0 are set to 0.                    Reason: no negative intensity values were␣
↪found in hazard.
```

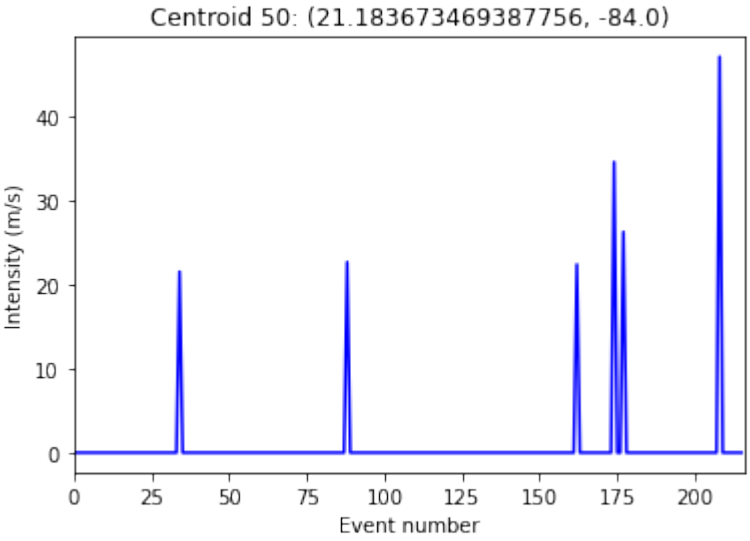
```
/Users/zeliestalhanske/python_projects/climada_python/climada/hazard/centroids/centr.py:
↪571: UserWarning: Geometry is in a geographic CRS. Results from 'distance' are likely␣
↪incorrect. Use 'GeoSeries.to_crs()' to re-project geometries to a projected CRS before␣
↪this operation.

  close_idx = self.geometry.distance(Point(x_lon, y_lat)).values.argmin()
```

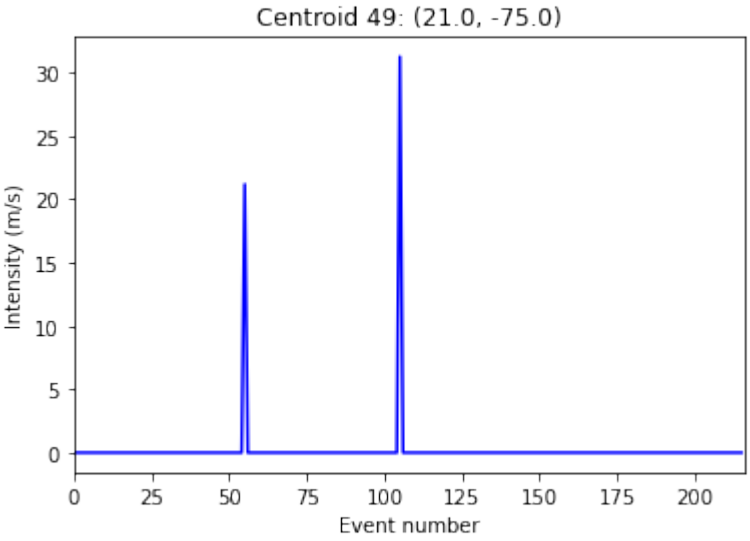

1-largest Event. ID 831: 1992230N11325
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
70
60
50
40
30
20
10
0
Intensity (m/s)


TC max intensity at each point
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
70
60
50
40
30
20
Intensity (m/s)


Event ID 1251: 1998295N12284
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W
81°W
79.5°W
78°W
76.5°W
40
30
20
10
0
Intensity (m/s)


Return period: 10 years
Return period: 50 years
Return period: 75 years
Return period: 100 years
28.5°N
27°N
25.5°N
24°N
22.5°N
82.5°W 81°W 79.5°W 78°W 76.5°W
Intensity (m/s)
0
10
20
30
40
50





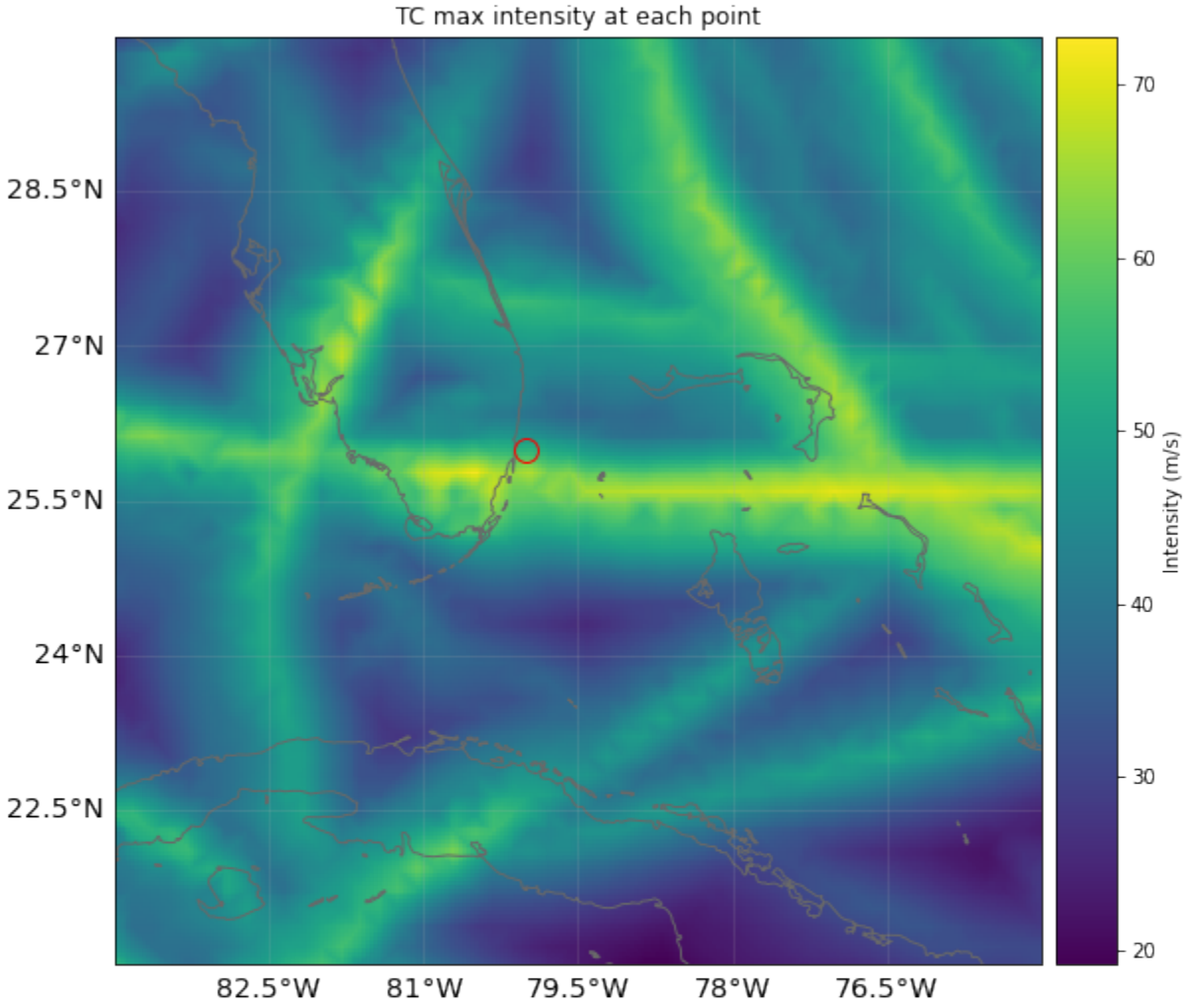
[12]:
```
# 7. one figure with two plots: maximum intensities and selected centroid with all␣
↪intensities:
from climada.util.plot import make_map
import matplotlib.pyplot as plt
plt.ioff()
fig, ax1, fontsize = make_map(1)  # map
ax2 = fig.add_subplot(2, 1, 2) # add regular axes
haz_tc_fl.plot_intensity(axis=ax1, event=0) # plot original resolution
ax1.plot(-80, 26, 'or', mfc='none', markersize=12)
haz_tc_fl.plot_intensity(axis=ax2, centr=(26, -80))
fig.subplots_adjust(hspace=6.5)
```

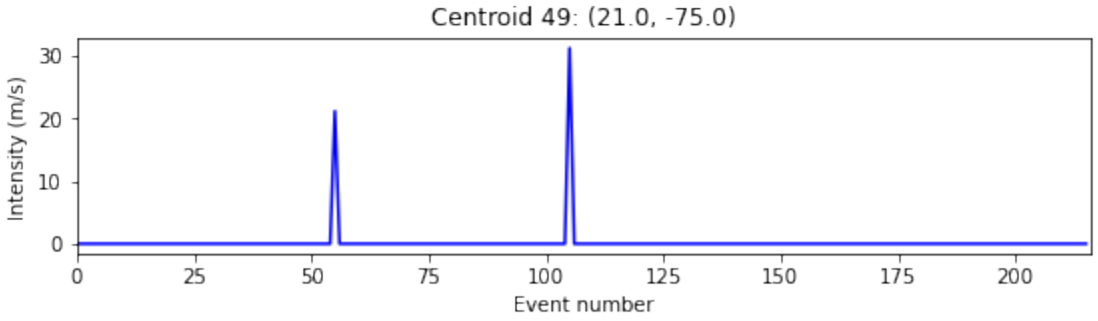
```
/Users/zeliestalhanske/python_projects/climada_python/climada/hazard/centroids/centr.py:
↪571: UserWarning: Geometry is in a geographic CRS. Results from 'distance' are likely␣
↪incorrect. Use 'GeoSeries.to_crs()' to re-project geometries to a projected CRS before␣
↪this operation.
```

(continues on next page)

(continued from previous page)

```
close_idx = self.geometry.distance(Point(x_lon, y_lat)).values.argmin()
```





Part 6: Write (=save) hazards

Hazards can be written and read in hdf5 format as follows:

[13]:
```
haz_tc_fl.write_hdf5('results/haz_tc_fl.h5')

haz = Hazard.from_hdf5('results/haz_tc_fl.h5')
haz.check()
```

GeoTiff data is generated using `write_raster()`:

[14]:
```
haz_ven.write_raster('results/haz_ven.tif') # each event is a band of the tif file
```

Pickle will work as well:

[15]:
```
from climada.util.save import save
# this generates a results folder in the current path and stores the output there
save('tutorial_haz_tc_fl.p', haz_tc_fl)
```

5.9 Hazard: Tropical cyclones

Tropical cyclones tracks are gathered in the class `TCTracks` and then provided to the hazard `TropCyclone` which computes the wind gusts at each centroid. `TropCyclone` inherits from `Hazard` and has an associated hazard type `TC`.

5.9.1 What do tropical cyclones look like in CLIMADA?

`TCTracks` reads and handles historical tropical cyclone tracks of the IBTrACS repository or synthetic tropical cyclone tracks simulated using fully statistical or coupled statistical-dynamical modeling approaches. It also generates synthetic tracks from the historical ones using Wiener processes.

The tracks are stored in the attribute `data`, which is a list of `xarray`'s `Dataset` (see xarray.Dataset). Each `Dataset` contains the following variables:

Coordinates

time latitude longitude ==========

Descriptive variables

time_step radius_max_wind max_sustained_wind central_pressure environmental_pressure
==========

Attributes

max_sustained_wind_unit central_pressure_unit sid name orig_event_flag data_provider basin id_no category
==========

How is this tutorial structured?

Part 1: Load TC tracks

a) Load TC tracks from historical records

b) Generate probabilistic events

c) ECMWF Forecast Tracks

d) Load TC tracks from other sources

Part 2: `TropCyclone()` class <#Part2>`__

a) Default hazard generation for tropical cyclones

b) Implementing climate change

c) Multiprocessing - improving performance for big computations

d) Making videos

Part 1: Load TC tracks

Records of historical TCs are very limited and therefore the database to study this natural hazard remains sparse. Only a small fraction of the TCs make landfall every year and reliable documentation of past TC landfalling events has just started in the 1950s (1980s - satellite era). The generation of synthetic storm tracks is an important tool to overcome this spatial and temporal limitation. Synthetic dataset are much larger and thus allow to estimate the risk of much rarer events. Here we show the most prominent tools in CLIMADA to load TC tracks from historical records *a)*, generate a probabilistic dataset thereof *b)*, and work with model simulations *c)*.

a) Load TC tracks from historical records

The best-track historical data from the International Best Track Archive for Climate Stewardship (IBTrACS) can easily be loaded into CLIMADA to study the historical records of TC events. The constructor `from_ibtracs_netcdf()` generates the `Datasets` for tracks selected by IBTrACS id, or by basin and year range. To achieve this, it downloads the first time the IBTrACS data v4 in netcdf format and stores it in `climada_python/data/system`. The tracks can be accessed later either using the attribute `data` or using `get_track()`, which allows to select tracks by its name or id. Use the method `append()` to extend the `data` list.

If you get an error downloading the IBTrACS data, try to manually access ftp://eclipse.ncdc.noaa.gov/pub/ibtracs//v04r00/provisional/netcdf/, connect as a *Guest* and copy the file `IBTrACS.ALL.v04r00.nc` to `climada_python/data/system`.

To visualize the tracks use `plot()`.

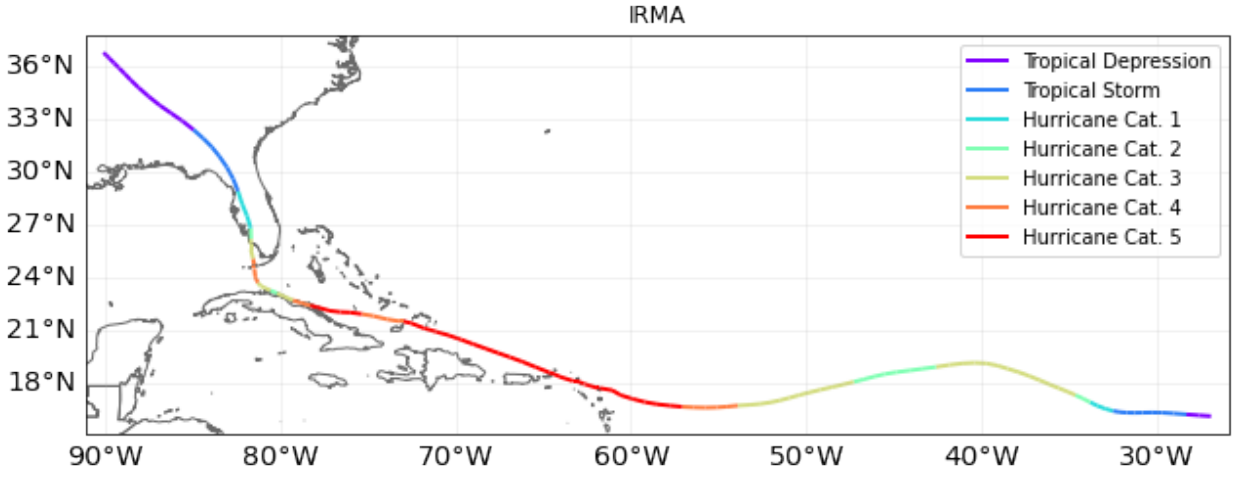
[1]:
```
%matplotlib inline
from climada.hazard import TCTracks

tr_irma = TCTracks.from_ibtracs_netcdf(provider='usa', storm_id='2017242N16333') # IRMA␣
↪2017
ax = tr_irma.plot()
ax.set_title('IRMA') # set title

# other ibtracs selection options
from climada.hazard import TCTracks
# years 1993 and 1994 in basin EP.
# correct_pres ignores tracks with not enough data. For statistics (frequency of events),
↪ these should be considered as well
sel_ibtracs = TCTracks.from_ibtracs_netcdf(provider='usa', year_range=(1993, 1994),␣
↪basin='EP', correct_pres=False)
```

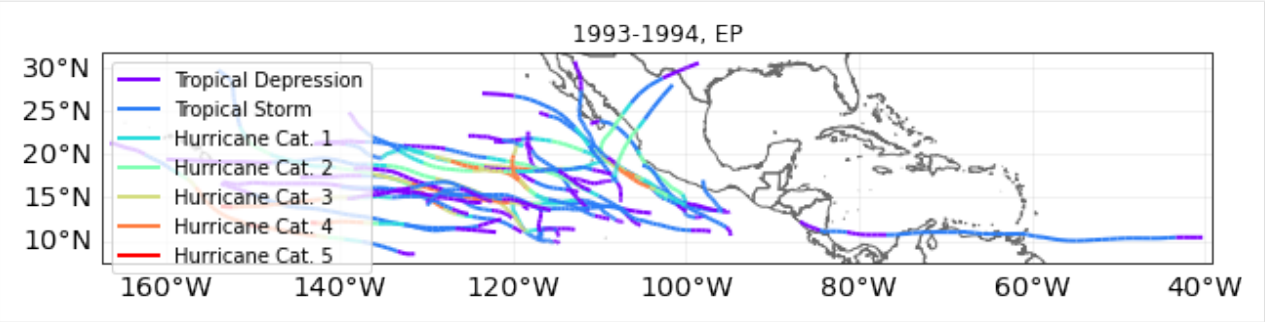
(continues on next page)

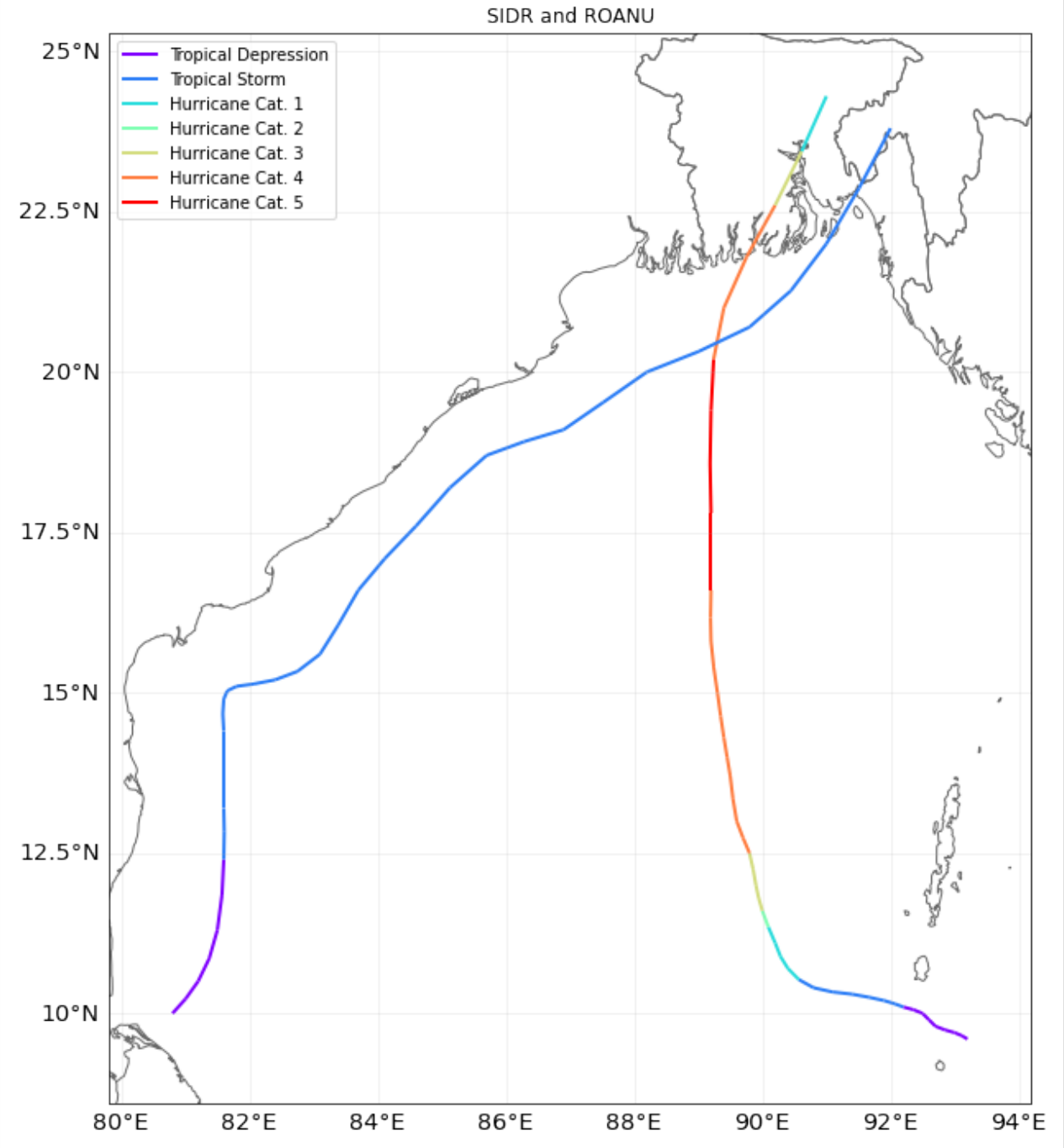
(continued from previous page)

```
print('Number of tracks:', sel_ibtracs.size)
ax = sel_ibtracs.plot()
ax.get_legend()._loc = 2 # correct legend location
ax.set_title('1993-1994, EP') # set title

track1 = TCTracks.from_ibtracs_netcdf(provider='usa', storm_id='2007314N10093') # SIDR␣
↪2007
track2 = TCTracks.from_ibtracs_netcdf(provider='usa', storm_id='2016138N10081') # ROANU␣
↪2016
track1.append(track2.data) # put both tracks together
ax = track1.plot()
ax.get_legend()._loc = 2 # correct legend location
ax.set_title('SIDR and ROANU'); # set title
```

```
2021-06-04 17:07:33,515 - climada.hazard.tc_tracks - INFO - Progress: 100%
2021-06-04 17:07:35,833 - climada.hazard.tc_tracks - WARNING - 19 storm events are␣
↪discarded because no valid wind/pressure values have been found: 1993178N14265,␣
↪1993221N12216, 1993223N07185, 1993246N16129, 1993263N11168, ...
2021-06-04 17:07:35,940 - climada.hazard.tc_tracks - INFO - Progress: 11%
2021-06-04 17:07:36,028 - climada.hazard.tc_tracks - INFO - Progress: 23%
2021-06-04 17:07:36,119 - climada.hazard.tc_tracks - INFO - Progress: 35%
2021-06-04 17:07:36,218 - climada.hazard.tc_tracks - INFO - Progress: 47%
2021-06-04 17:07:36,312 - climada.hazard.tc_tracks - INFO - Progress: 58%
2021-06-04 17:07:36,399 - climada.hazard.tc_tracks - INFO - Progress: 70%
2021-06-04 17:07:36,493 - climada.hazard.tc_tracks - INFO - Progress: 82%
2021-06-04 17:07:36,585 - climada.hazard.tc_tracks - INFO - Progress: 94%
2021-06-04 17:07:36,612 - climada.hazard.tc_tracks - INFO - Progress: 100%
Number of tracks: 33
2021-06-04 17:07:38,825 - climada.hazard.tc_tracks - INFO - Progress: 100%
2021-06-04 17:07:39,974 - climada.hazard.tc_tracks - INFO - Progress: 100%
```




1993-1994, EP
Tropical Depression
Tropical Storm
Hurricane Cat. 1
Hurricane Cat. 2
Hurricane Cat. 3
Hurricane Cat. 4
Hurricane Cat. 5
30°N
25°N
20°N
15°N
10°N
160°W
140°W
120°W
100°W
80°W
60°W
40°W


SIDR and ROANU
Tropical Depression
Tropical Storm
Hurricane Cat. 1
Hurricane Cat. 2
Hurricane Cat. 3
Hurricane Cat. 4
Hurricane Cat. 5
25°N
22.5°N
20°N
17.5°N
15°N
12.5°N
10°N
80°E
82°E
84°E
86°E
88°E
90°E
92°E
94°E

[2]:
```
tr_irma.get_track('2017242N16333')
```

[2]:
```
<xarray.Dataset>
Dimensions:                 (time: 123)
Coordinates:
  * time                    (time) datetime64[ns] 2017-08-30 ... 2017-09-13T1...
    lat                     (time) float32 16.1 16.15 16.2 ... 36.2 36.5 36.8
    lon                     (time) float32 -26.9 -27.59 -28.3 ... -89.79 -90.1
Data variables:
    time_step               (time) float64 3.0 3.0 3.0 3.0 ... 3.0 3.0 3.0 3.0
    radius_max_wind         (time) float32 60.0 60.0 60.0 ... 60.0 60.0 60.0
    radius_oci              (time) float32 180.0 180.0 180.0 ... 350.0 350.0
    max_sustained_wind      (time) float32 30.0 32.0 35.0 ... 15.0 15.0 15.0
    central_pressure        (time) float32 1.008e+03 1.007e+03 ... 1.005e+03
    environmental_pressure  (time) float64 1.012e+03 1.012e+03 ... 1.008e+03
    basin                   (time) <U2 'NA' 'NA' 'NA' 'NA' ... 'NA' 'NA' 'NA'
Attributes:
    max_sustained_wind_unit:  kn
    central_pressure_unit:    mb
    name:                     IRMA
    sid:                      2017242N16333
    orig_event_flag:          True
    data_provider:            ibtracs_usa
    id_no:                    2017242016333.0
    category:                 5
```

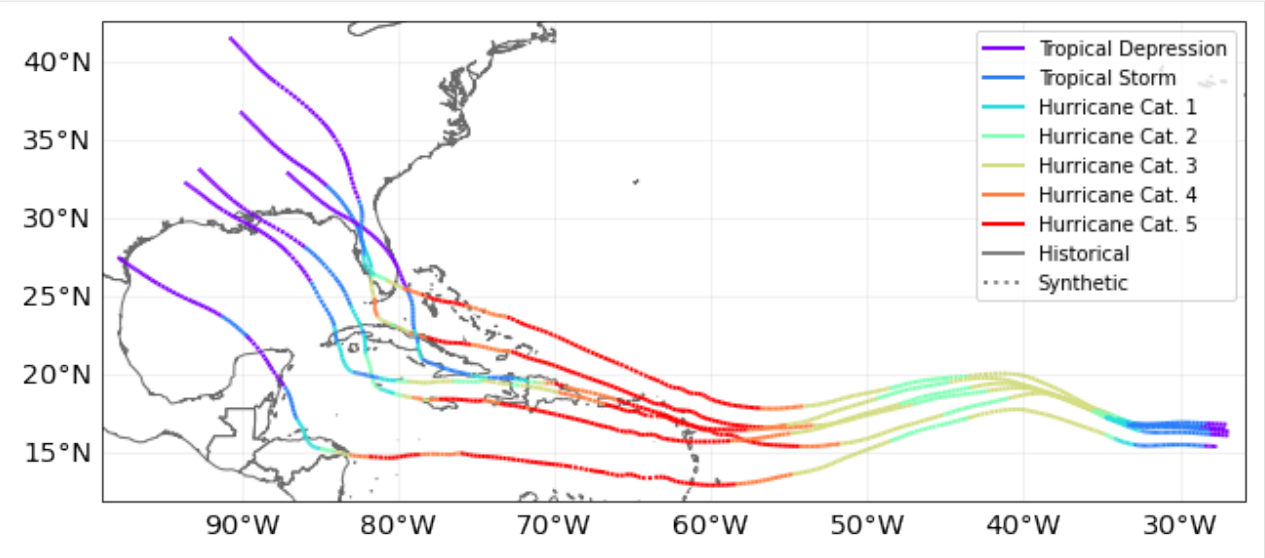
b) Generate probabilistic events

Once tracks are present in `TCTracks`, one can generate synthetic tracks for each present track based on directed random walk. Note that the tracks should be interpolated to use the same timestep **before** generation of probabilistic events.

`calc_perturbed_trajectories()` generates an ensemble of "nb_synth_tracks" numbers of synthetic tracks is computed for every track. The methodology perturbs the tracks locations, and if decay is `True` it additionally includes decay of wind speed and central pressure drop after landfall. No other track parameter is perturbed.

[3]:
```
# here we use tr_irma retrieved from IBTrACS with the function above
# select number of synthetic tracks (nb_synth_tracks) to generate per present tracks.
tr_irma.equal_timestep()
tr_irma.calc_perturbed_trajectories(nb_synth_tracks=5)
tr_irma.plot()
# see more configutration options (e.g. amplitude of max random starting point shift in␣
↪decimal degree; max_shift_ini)
```

[3]:
```
<GeoAxesSubplot:>
```



[4]:
```
tr_irma.data[-1] # last synthetic track. notice the value of orig_event_flag and name
```

[4]:
```
<xarray.Dataset>
Dimensions:                  (time: 349)
Coordinates:
  * time                     (time) datetime64[ns] 2017-08-30 ... 2017-09-13T1...
    lon                      (time) float64 -27.64 -27.8 -27.96 ... -97.81 -97.93
    lat                      (time) float64 15.39 15.41 15.42 ... 27.41 27.49
Data variables:
    time_step                (time) float64 1.0 1.0 1.0 1.0 ... 1.0 1.0 1.0 1.0
    radius_max_wind          (time) float64 60.0 60.0 60.0 ... 60.0 60.0 60.0
    radius_oci               (time) float64 180.0 180.0 180.0 ... 350.0 350.0
    max_sustained_wind       (time) float64 30.0 30.67 31.33 ... 15.0 14.99 14.96
    central_pressure         (time) float64 1.008e+03 1.008e+03 ... 1.005e+03
    environmental_pressure   (time) float64 1.012e+03 1.012e+03 ... 1.008e+03
    basin                    (time) <U2 'NA' 'NA' 'NA' 'NA' ... 'NA' 'NA' 'NA'
    on_land                  (time) bool False False False ... False True True
    dist_since_lf            (time) float64 nan nan nan nan ... nan 7.605 22.71
Attributes:
    max_sustained_wind_unit:  kn
    central_pressure_unit:    mb
    name:                     IRMA_gen5
    sid:                      2017242N16333_gen5
    orig_event_flag:          False
    data_provider:            ibtracs_usa
    id_no:                    2017242016333.05
    category:                 5
```

EXERCISE

Using the first synthetic track generated,

1. Which is the time frequency of the data?
2. Compute the maximum sustained wind for each day.

[5]:
```
# Put your code here
```

[6]:
```
# SOLUTION:
import numpy as np
# select the track
tc_syn = tr_irma.get_track('2017242N16333_gen1')

# 1. Which is the time frequency of the data?
# The values of a DataArray are numpy.arrays.
# The nummpy.ediff1d computes the different between elements in an array
diff_time_ns = np.ediff1d(tc_syn.time)
diff_time_h = diff_time_ns.astype(int)/1000/1000/1000/60/60
print('Mean time frequency in hours:', diff_time_h.mean())
print('Std time frequency in hours:', diff_time_h.std())
print()

# 2. Compute the maximum sustained wind for each day.
print('Daily max sustained wind:', tc_syn.max_sustained_wind.groupby('time.day').max())
```

```
Mean time frequency in hours: 1.0
Std time frequency in hours: 0.0

Daily max sustained wind: <xarray.DataArray 'max_sustained_wind' (day: 15)>
array([100.        , 100.        , 100.        , 123.33333333,
       155.        , 155.        , 150.        , 138.        ,
        51.85384486,  58.03963987,  29.03963987,   3.57342356,
         3.35512013,  54.        ,  99.        ])
Coordinates:
  * day      (day) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 30 31
```

c) ECMWF Forecast Tracks

ECMWF publishes tropical cyclone forecast tracks free of charge as part of the WMO essentials. These tracks are detected automatically in the ENS and HRES models. The non-supervised nature of the model may lead to artefacts.

The `tc_fcast` trackset below inherits from TCTracks, but contains some additional metadata that follows ECMWF's definitions. Try plotting these tracks and compare them to the official cones of uncertainty! The example track at `tc_fcast.data[0]` shows the data structure.

[7]:
```
# This functionality is part of climada_petals, uncomment to execute
# from climada_petals.hazard import TCForecast
#
# tc_fcast = TCForecast()
```

(continues on next page)

(continued from previous page)

```
# tc_fcast.fetch_ecmwf()
#
# print(tc_fcast.data[0])
```

d) Load TC tracks from other sources

In addition to the *historical records of TCs (IBTrACS)*, the *probabilistic extension* of these tracks, and the *ECMWF Forecast tracks*, CLIMADA also features functions to read in synthetic TC tracks from other sources. These include synthetic storm tracks from Kerry Emanuel's coupled statistical-dynamical model (Emanuel et al., 2006 as used in Geiger et al., 2016), synthetic storm tracks from a second coupled statistical-dynamical model (CHAZ) (as described in Lee et al., 2018), and synthetic storm tracks from a fully statistical model (STORM) Bloemendaal et al., 2020). However, these functions are partly under development and/or targeted at advanced users of CLIMADA in the context of very specific use cases. They are thus not covered in this tutorial.

Part 2: TropCyclone() class

The `TropCyclone` class is a derived class of *Hazard*. As such, it contains all the attributes and methods of a `Hazard`. Additionally, it comes with the constructor method `from_tracks` to model tropical cyclones from tracks contained in a `TCTracks` instance.

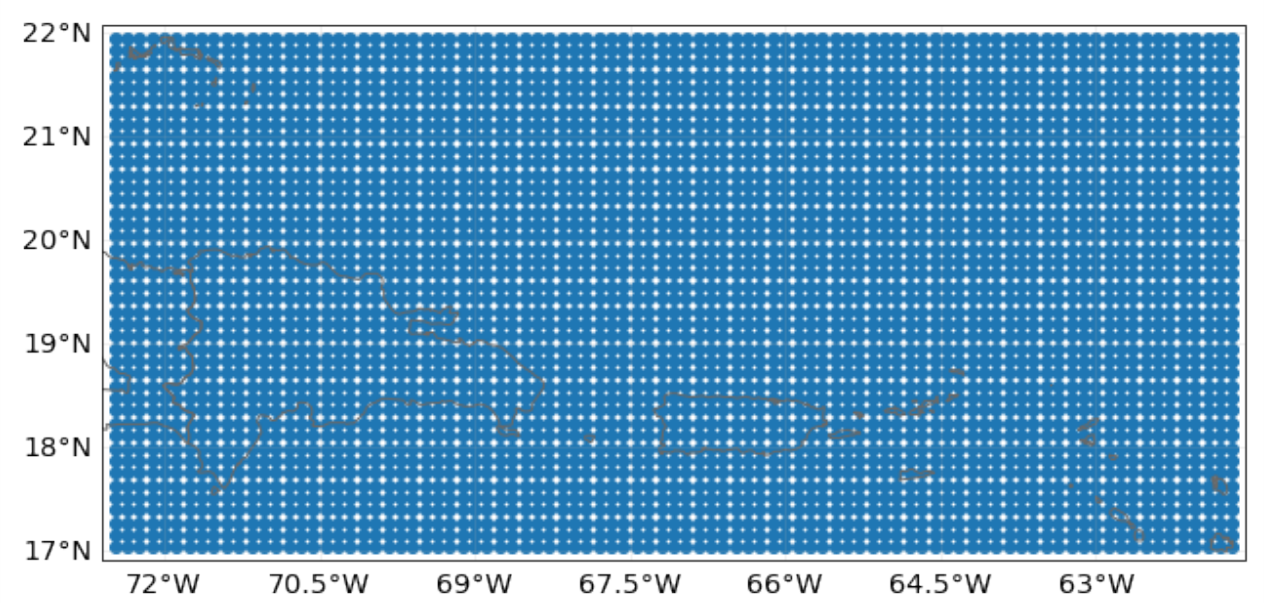
When setting tropical cyclones from tracks, the centroids where to map the wind gusts (the hazard intensity) can be provided. If no centroids are provided, the global centroids `GLB_NatID_grid_0360as_adv_2.mat` are used.

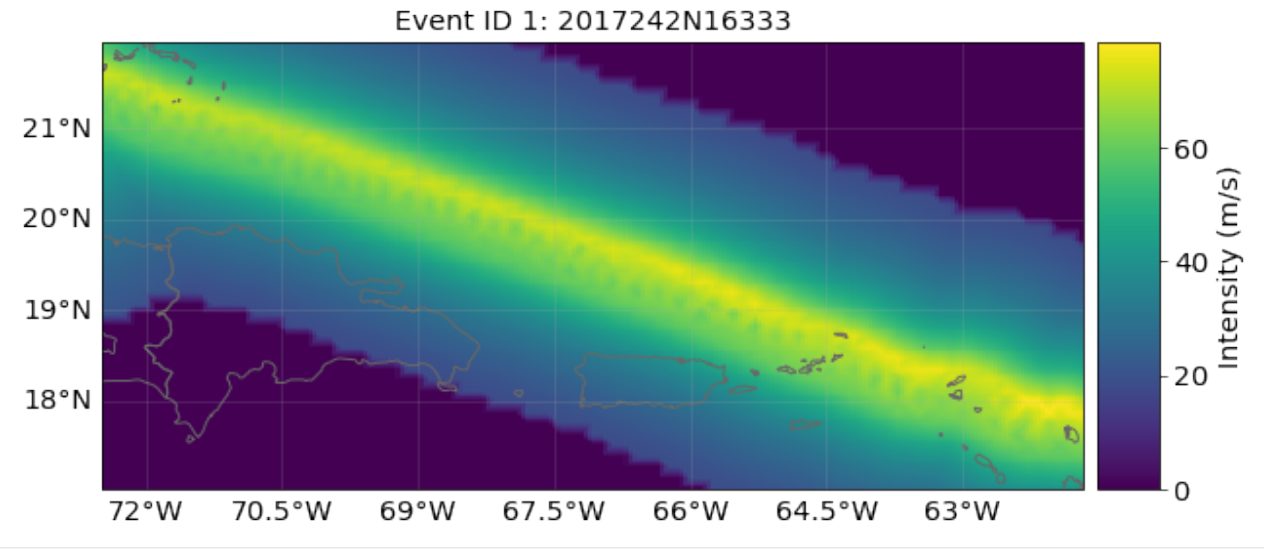
From the track properties the 1 min sustained peak gusts are computed in each centroid as the sum of a circular wind field (following Holland, 2008) and the translational wind speed that arises from the storm movement. We incorporate the decline of the translational component from the cyclone centre by multiplying it by an attenuation factor. See CLIMADA v1 and references therein for more information.

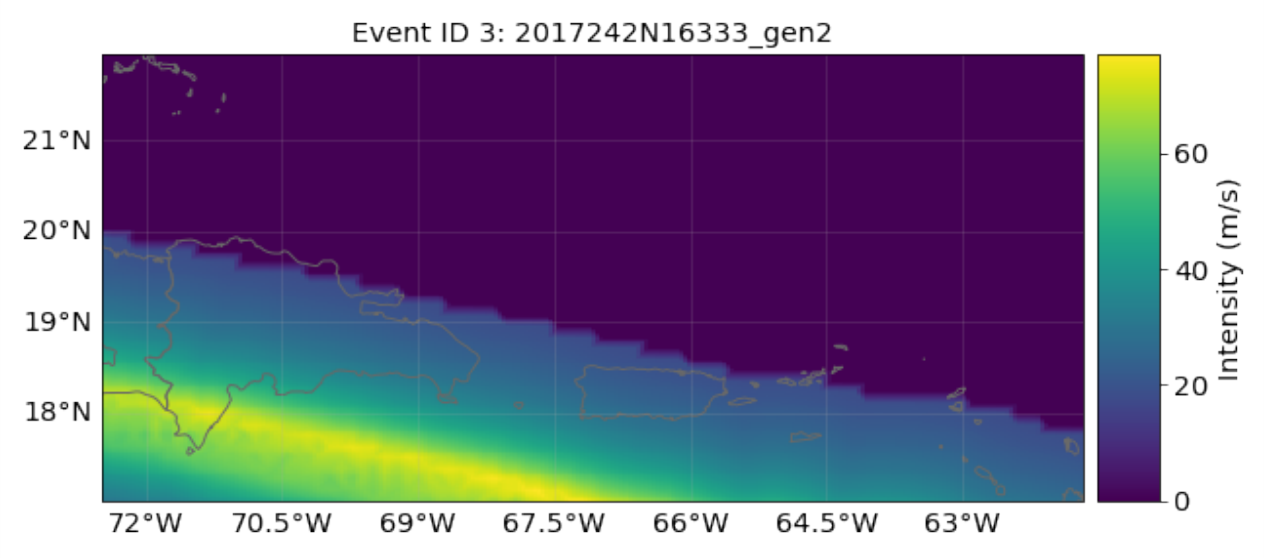
a) Default hazard generation for tropical cyclones

[8]:
```
from climada.hazard import Centroids, TropCyclone

# construct centroids
min_lat, max_lat, min_lon, max_lon = 16.99375, 21.95625, -72.48125, -61.66875
cent = Centroids.from_pnt_bounds((min_lon, min_lat, max_lon, max_lat), res=0.12)
cent.check()
cent.plot()

# construct tropical cyclones
tc_irma = TropCyclone.from_tracks(tr_irma, centroids=cent)
# tc_irma = TropCyclone.from_tracks(tr_irma)  # try without given centroids
tc_irma.check()
tc_irma.plot_intensity('2017242N16333')  # IRMA
tc_irma.plot_intensity('2017242N16333_gen2')  # IRMA's synthetic track 2
```

[8]:
```
<GeoAxesSubplot:title={'center':'Event ID 3: 2017242N16333_gen2'}>
```


22°N
21°N
20°N
19°N
18°N
17°N
72°W
70.5°W
69°W
67.5°W
66°W
64.5°W
63°W


Event ID 1: 2017242N16333
21°N
20°N
19°N
18°N
72°W
70.5°W
69°W
67.5°W
66°W
64.5°W
63°W
60
40
20
0
Intensity (m/s)

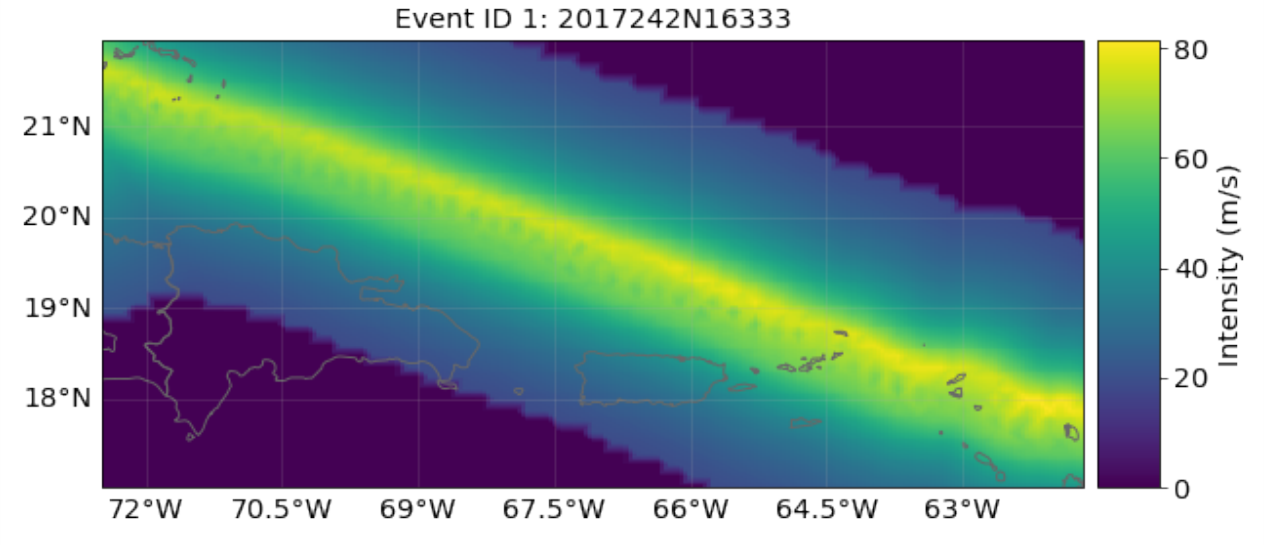


b) Implementing climate change

`apply_climate_scenario_knu` implements the changes on intensity and frequency due to climate change described in *Global projections of intense tropical cyclone activity for the late twenty-first century from dynamical downscaling of CMIP5/RCP4.5 scenarios* of Knutson et al 2015. Other RCP scenarios are approximated from the RCP 4.5 values by interpolating them according to their relative radiative forcing.

[9]:
```
# an Irma event-like in 2055 under RCP 4.5:
tc_irma = TropCyclone.from_tracks(tr_irma, centroids=cent)
tc_irma_cc = tc_irma.apply_climate_scenario_knu(ref_year=2055, rcp_scenario=45)
tc_irma_cc.plot_intensity('2017242N16333')
```

[9]:
```
<GeoAxesSubplot:title={'center':'Event ID 1: 2017242N16333'}>
```



Note: this method to implement climate change is simplified and does only take into account changes in TC frequency and intensity. However, how hurricane damage changes with climate remains challenging to assess. Records of hurricane damage exhibit widely fluctuating values because they depend on rare, landfalling events which are substantially more volatile than the underlying basin-wide TC characteristics. For more accurate future projections of how a warming climate might shape TC characteristics, there is a two-step process needed. First, the understanding of how climate

change affects critical environmental factors (like SST, humidity, etc.) that shape TCs is required. Second, the means of simulating how these changes impact TC characteristics (such as intensity, frequency, etc.) are necessary. Statistical-dynamical models (Emanuel et al., 2006 and Lee et al., 2018) are physics-based and allow for such climate change studies. However, this goes beyond the scope of this tutorial.

c) Multiprocessing - improving performance for big computations

WARNING: Uncomment and execute these lines in a console, outside Jupyter Notebook. Multiprocessing is implemented in the tropical cyclones. Simply provide pool in the constructor. When dealing with a big amount of data, you might consider using it as follows:

[10]:
```
#from climada.hazard import TCTracks, Centroids, TropCyclone
#from pathos.pools import ProcessPool as Pool

#pool = Pool() # start a pathos pool

#tc_track = TCTracks.from_ibtracs_netcdf(provider='usa', year_range=(1992, 1994), basin='EP
↪')
#tc_track.calc_perturbed_trajectories(pool=pool) # OPTIONAL: if you want to generate a␣
↪probabilistic set of TC tracks.
#tc_track.equal_timestep(pool=pool)

#lon_min, lat_min, lon_max, lat_max = -160, 10, -100, 36
#centr = Centroids.from_pnt_bounds((lon_min, lat_min, lon_max, lat_max), 0.1)

#tc_haz = TropCyclone.from_tracks(tc_track, centroids=centr, pool=pool)  # provide the␣
↪pool in the constructor
#tc_haz.check()

#pool.close()
#pool.join()
```

d) Making videos **WARNING**: Uncomment and execute these lines in a console, outside Jupyter Notebook.

Videos of a tropical cyclone hitting specific centroids are done automatically using the method `video_intensity()`.

[11]:
```
#lon_min, lat_min, lon_max, lat_max = -83.5, 24.4, -79.8, 29.6
#centr_video = Centroids.from_pnt_bounds((lon_min, lat_min, lon_max, lat_max), 0.04)
#centr_video.check()

#track_name = '2017242N16333' # '2016273N13300' # '1992230N11325'
#tc_video = TropCyclone()
# use file_name='' to not to write the video
#tc_list, tr_coord = tc_video.video_intensity(track_name, tr_irma, centr_video, file_
↪name='./results/irma_tc_fl.gif')
# tc_list contains a list with TropCyclone instances plotted at each time step
# tr_coord contains a list with the track path coordinates plotted at each time step

# mp4 occupies much less space! To use it:
# conda install ffmpeg
# in code:
# plt.rcParams['animation.ffmpeg_path']='path/to/climada_env/bin/ffmpeg'
# writer=animation.FFMpegWriter(bitrate=500)
# tc_list, tr_coord = tc_video.video_intensity(track_name, tr_irma, centr_video, file_
↪name='./results/irma_tc_fl.gif', writer=writer)
```

REFERENCES:

- Bloemendaal, N., Haigh, I. D., de Moel, H., Muis, S., Haarsma, R. J., & Aerts, J. C. J. H. (2020). Generation of a global synthetic tropical cyclone hazard dataset using STORM. Scientific Data, 7(1). https://doi.org/10.1038/s41597-020-0381-2
- Emanuel, K., S. Ravela, E. Vivant, and C. Risi, 2006: A Statistical Deterministic Approach to Hurricane Risk Assessment. Bull. Amer. Meteor. Soc., 87, 299–314, https://doi.org/10.1175/BAMS-87-3-299.
- Geiger, T., Frieler, K., & Levermann, A. (2016). High-income does not protect against hurricane losses. Environmental Research Letters, 11(8). https://doi.org/10.1088/1748-9326/11/8/084012
- Knutson, T. R., Sirutis, J. J., Zhao, M., Tuleya, R. E., Bender, M., Vecchi, G. A., … Chavas, D. (2015). Global projections of intense tropical cyclone activity for the late twenty-first century from dynamical downscaling of CMIP5/RCP4.5 scenarios. Journal of Climate, 28(18), 7203–7224. https://doi.org/10.1175/JCLI-D-15-0129.1
- Lee, C. Y., Tippett, M. K., Sobel, A. H., & Camargo, S. J. (2018). An environmentally forced tropical cyclone hazard model. Journal of Advances in Modeling Earth Systems, 10(1), 223–241. https://doi.org/10.1002/2017MS001186

5.10 Hazard: winter windstorms / extratropical cyclones in Europe

5.10.1 Or: The StormEurope hazard subclass of CLIMADA

Auth: Jan Hartman & Thomas Röösli
Date: 2018-04-26 & 2020-03-03

This notebook will give a quick tour of the capabilities of the StormEurope hazard class. This includes functionalities to apply probabilistic alterations to historical storms.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [15, 10]
```

5.10.2 Reading Data

StormEurope was written under the presumption that you'd start out with WISC storm footprint data in netCDF format. This notebook works with a demo dataset. If you would like to work with the real data: (1) Please follow the link and download the file C3S_WISC_FOOTPRINT_NETCDF_0100.tgz from the Copernicus Windstorm Information Service, (2) unzip it (3) uncomment the last two lines in the following codeblock and (4) adjust the variable "WISC_files".

We first construct an instance and then point the reader at a directory containing compatible `.nc` files. Since there are other files in there, we must be explicit and use a globbing pattern; supplying incompatible files will make the reader fail.

The reader actually calls `climada.util.files_handler.get_file_names`, so it's also possible to hand it an explicit list of filenames, or a dirname, or even a list of glob patterns or directories.

[2]:
```
from climada.hazard import StormEurope
from climada.util.constants import WS_DEMO_NC

storm_instance = StormEurope.from_footprints(WS_DEMO_NC)
```

(continues on next page)

(continued from previous page)

```
# WISC_files = '/path/to/folder/C3S_WISC_FOOTPRINT_NETCDF_0100/fp_era[!er5]*_0.nc'
# storm_instance = StormEurope.from_footprints(WISC_files)
```

```
$CLIMADA_SRC/climada/hazard/centroids/centr.py:822: UserWarning: Geometry is in a␣
↪geographic CRS. Results from 'buffer' are likely incorrect. Use 'GeoSeries.to_crs()'␣
↪to re-project geometries to a projected CRS before this operation.

  xy_pixels = self.geometry.buffer(res / 2).envelope
```

5.10.3 Introspection

Let's quickly see what attributes this class brings with it:

[3]:
```
storm_instance?
```

```
Type:            StormEurope
String form:     <climada.hazard.storm_europe.StormEurope object at 0x7f2a986b4c70>
File:            ~/code/climada_python/climada/hazard/storm_europe.py
Docstring:
A hazard set containing european winter storm events. Historic storm
events can be downloaded at http://wisc.climate.copernicus.eu/ and read
with `from_footprints`. Weather forecasts can be automatically downloaded from
https://opendata.dwd.de/ and read with from_icon_grib(). Weather forecast
from the COSMO-Consortium http://www.cosmo-model.org/ can be read with
from_cosmoe_file().

Attributes
----------
ssi_wisc : np.array, float
    Storm Severity Index (SSI) as recorded in
    the footprint files; apparently not reproducible from the footprint
    values only.
ssi : np.array, float
    SSI as set by set_ssi; uses the Dawkins
    definition by default.
Init docstring: Calls the Hazard init dunder. Sets unit to 'm/s'.
```

You could also try listing all permissible methods with `dir(storm_instance)`, but since that would include the methods from the `Hazard` base class, you wouldn't know what's special. The best way is to read the source: uncomment the following statement to read more.

[4]:
```
# StormEurope??
```

5.10.4 Into the Storm Severity Index (SSI)

The SSI, according to Dawkins et al. 2016 or Lamb and Frydendahl, 1991, can be set using `set_ssi`. For demonstration purposes, I show the default arguments. (Check also the defaults using `storm_instance.calc_ssi?`, the method for which set_ssi is a wrapper.)

We won't be using the `plot_ssi` functionality just yet, because we only have two events; the graph really isn't informative. After this, we'll generate some more storms to make that plot more aesthetically pleasing.

[5]:
```
storm_instance.set_ssi(
    method = 'wind_gust',
    intensity = storm_instance.intensity,
    # the above is just a more explicit way of passing the default
    on_land = True,
    threshold = 25,
    sel_cen = None
    # None is default. sel_cen could be used to subset centroids
)
```

5.10.5 Probabilistic Storms

This class allows generating probabilistic storms from historical ones according to a method outlined in Schwierz et al. 2010. This means that per historical event, we generate 29 new ones with altered intensities. Since it's just a bunch of vector operations, this is pretty fast.

However, we should not return the entire probabilistic dataset in-memory: in trials, this used up 60 GB of RAM, thus requiring a great amount of swap space. Instead, we must select a country by setting the `reg_id` parameter to an ISO_N3 country code used in the Natural Earth dataset. It is also possible to supply a list of ISO codes. If your machine is up for the job of handling the whole dataset, set the `reg_id` parameter to `None`.

Since assigning each centroid a country ID is a rather inefficient affair, you may need to wait a minute or two for the entire WISC dataset to be processed. For the small demo dataset, it runs pretty quickly.

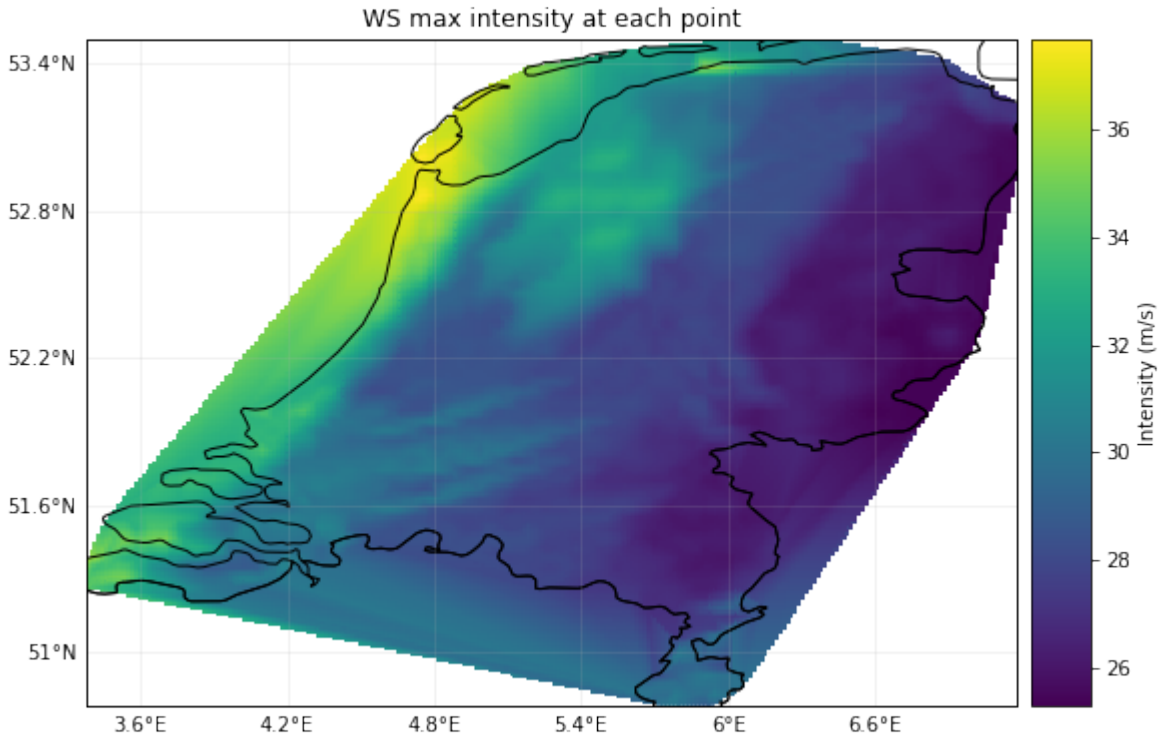
[7]:
```
%%time
storm_prob = storm_instance.generate_prob_storms(reg_id=528)
storm_prob.plot_intensity(0)
```

```
2020-03-05 10:29:31,845 - climada.hazard.centroids.centr - INFO - Setting geometry␣
↪points.
2020-03-05 10:29:32,248 - climada.hazard.centroids.centr - DEBUG - Setting region_id␣
↪9944 points.
2020-03-05 10:29:32,466 - climada.util.coordinates - DEBUG - Setting region_id 9944␣
↪points.
2020-03-05 10:29:33,506 - climada.hazard.storm_europe - INFO - Commencing probabilistic␣
↪calculations
2020-03-05 10:29:33,620 - climada.hazard.storm_europe - INFO - Generating new␣
↪StormEurope instance
2020-03-05 10:29:33,663 - climada.util.checker - DEBUG - Hazard.ssi not set.
2020-03-05 10:29:33,664 - climada.util.checker - DEBUG - Hazard.ssi_wisc not set.
2020-03-05 10:29:33,665 - climada.util.checker - DEBUG - Hazard.event_name not set.␣
↪Default values set.
```

```
C:\shortpaths\GitHub\climada_python\climada\util\plot.py:311: UserWarning: Tight layout
↪not applied. The left and right margins cannot be made large enough to accommodate all
↪axes decorations.
  fig.tight_layout()
```

```
Wall time: 2.24 s
```

[7]:
```
<cartopy.mpl.geoaxes.GeoAxesSubplot at 0x1dafba69940>
```

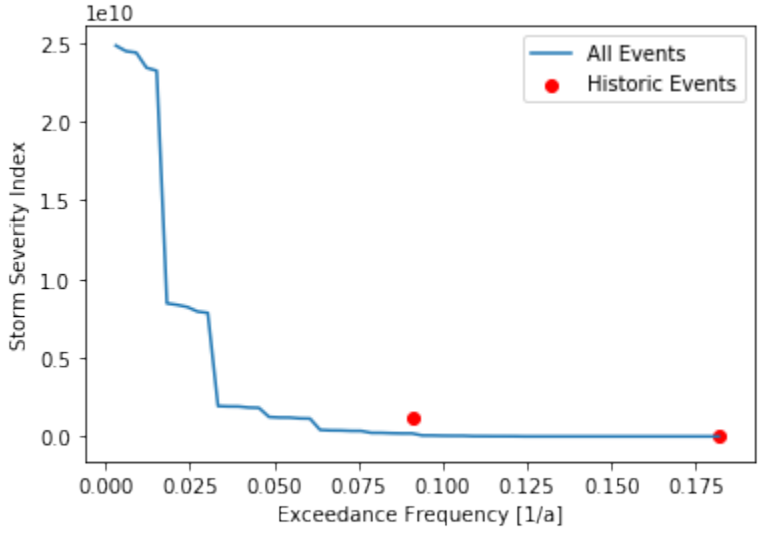


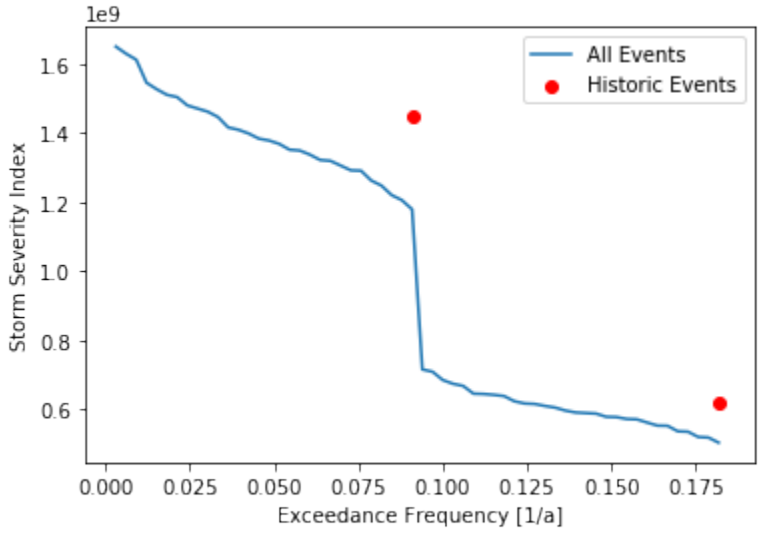
We can get much more fancy in our calls to `generate_prob_storms`; the keyword arguments after `ssi_args` are passed on to `_hist2prob`, allowing us to tweak the probabilistic permutations.

[7]:
```
ssi_args = {
    'on_land': True,
    'threshold': 25,
}

storm_prob_xtreme = storm_instance.generate_prob_storms(
    reg_id=[56, 528], # BEL and NLD
    spatial_shift=2,
    ssi_args=ssi_args,
    power=1.5,
    scale=0.3
)
```

We can now check out the SSI plots of both these calculations. The comparison between the historic and probabilistic ssi values, only makes sence for the full dataset.

[10]:
```
storm_prob_xtreme.plot_ssi(full_area=True)
storm_prob.plot_ssi(full_area=True)
```





[10]:
```
(<Figure size 1080x720 with 1 Axes>,
 <AxesSubplot:xlabel='Exceedance Frequency [1/a]', ylabel='Storm Severity Index'>)
```

5.11 END-TO-END IMPACT CALCULATION

5.11.1 Goal of this tutorial

The goal of this tutorial is to show a full end-to-end impact computation. Note that this tutorial examplifies the work flow, but does not explore all possible features.

5.11.2 What is an Impact?

The impact is the combined effect of hazard events on a set of exposures mediated by a set of impact functions. By computing the impact for each event (historical and synthetic) and for each exposure value at each geographical location, the Impact class provides different risk measures, such as the expected annual impact per exposure, the probable maximum impact for different return periods, and the total average annual impact.

5.11.3 Impact class data structure

The impact class does not require any attributes to be defined by the user. All attributes are set by the method `impact.calc()`. This method requires three attributes: an `Exposure`, a `Hazard`, and an `ImpactFuncSet`. After calling `impact.calc(Exposure, ImpactFuncSet, Hazard, save_mat=False)`, the Impact object has the following attributes:

Attributes from input	Data Type	Description
tag	(dict)	dictionary storing the tags of the inputs (Exposure.tag, ImpactFuncSet.tag Hazard.tag)
even_id	list(int)	id (>0) of each hazard event (Hazard.event_id)
event_name	(list(str))	name of each event (Hazard.event_name)
date	np.array	date of events (Hazard.date)
coord_exp	np.array	exposures coordinates [lat, lon] (in degrees) (Exposure.gdf.latidues, Exposure.gdf.longitude)
frequency	np.array	annual frequency of events (Hazard.frequency)
unit	str	value unit used (Exposure.value_unit)
csr	str	unit system for Exposure and Hazard geographical data (Exposure.csr)

Computed attributes	Data Type	Description
at_event	np.array	impact for each hazard event summed over all locations
eai_exp	np.array	expected annual impact for each locations, summed over all events weigthed by frequency
aai_agg	float	total annual average aggregated impact value (summed over events and locations)
impt_mat	sparse.csr_matrix	matrix (num_events X num_exp) with impact values (only filled if save_mat is True).
tot_value	float	total exposure value affected (sum of value all exposures locations affected by at least one hazard event)

All other methods compute values from the attributes set by `Impact.cacl()`. For example, one can compute the frequency exceedance curve, plot impact data, or compute traditional risk transfer over impact.

5.11.4 How do I compute an impact in CLIMADA?

In CLIMADA, impacts are computed using the `Impact` class. To computation of the impact requires an `Exposure` , an `ImpactFuncSet`, and a `Hazard` object. For details about how to define *Exposures* , *Hazard*, *Impact Functions* see the respective tutorials.

The steps of an impact caculations are typically:

- Set exposure
- Set hazard and hazard centroids
- Set impact functions in impact function set
- Compute impact
- Visualize, save, use impact output

Hints: Before computing the impact of a given `Exposure` and `Hazard`, it is important to correctly match the `Exposures'` coordinates with the `Hazard Centroids`. Try to have similar resolutions in `Exposures` and `Hazard`. By the impact calculation the nearest neighbor for each `Exposure` to the `Hazard's Centroids` is searched.

Hint: Set first the `Exposures` and use its coordinates information to set a matching `Hazard`.

Hint: The configurable parameter `max_matrix_size` defined in the configuration file (located at `/climada/conf/defaults.conf`) controls the maximum matrix size contained in a chunk. You can decrease its value if you are having memory issues when using the `Impact.calc()` method. A high value will make the computation fast, but increase the memory use.

5.11.5 Structure of the tutorial

We begin with one very detailled example, and later present in quick and dirty examples.

Part1: Detailed impact calculation with Litpop and TropCyclone

Part2: Quick examples: raster and point exposures/hazards

Part3: Visualization methods

Detailed Impact calculation - LitPop + TropCyclone

We present a detailed example for the hazard *Tropical Cyclones* and the exposures from *LitPop* .

Define the exposure

Reminder: The exposures must be defined according to your problem either using CLIMADA exposures such as BlackMarble, *LitPop*,OSM, extracted from external sources (imported via csv, excel, api, ...) or directly user defined. As a reminder, exposures are geopandas dataframes with at least columns 'latitude', 'longitude' and 'value' of exposures. For impact calculations, for each exposure value the corresponding impact function to use (defined by the column **impf_**) and the associated hazard centroids must be defined. This is done after defining the impact function(s) and the hazard(s). See tutorials on *Exposures* , *Hazard*, *ImpactFuncSet* for more details.

Exposures are either defined as a series of (latitude/longitude) points or as a raster of (latitude/longitude) points. Fundamentally, this changes nothing for the impact computations. Note that for larger number of points, consider using a raster which might be more efficient (computationally). For a low number of points, avoid using a raster if this adds a lot of exposures values equal to 0.

We shall here use a raster example.

[1]:
```
# Exposure from the module Litpop
# Note that the file gpw_v4_population_count_rev11_2015_30_sec.tif must be downloaded␣
↪(do not forget to unzip) if
# you want to execute this cell on your computer.

%matplotlib inline
import numpy as np
from climada.entity import LitPop

# Cuba with resolution 10km and financial_mode = income group.
exp_lp = LitPop.from_countries(countries=['CUB'], res_arcsec=300, fin_mode='income_group
↪')
exp_lp.check()
```

```
2021-10-19 16:49:06,420 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

[2]:
```
exp_lp.gdf.head()
```

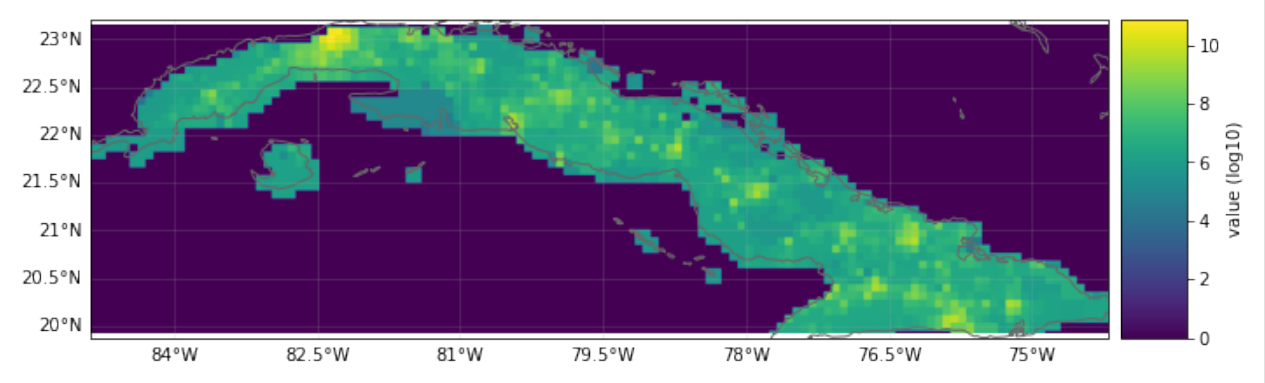
[2]:

	value	geometry	latitude	longitude	region_id	\
0	4.730991e+10	POINT (-82.37500 23.12500)	23.125	-82.375000	192	
1	2.990876e+10	POINT (-82.29167 23.12500)	23.125	-82.291667	192	
2	6.839380e+09	POINT (-82.20833 23.12500)	23.125	-82.208333	192	
3	1.925246e+09	POINT (-82.12500 23.12500)	23.125	-82.125000	192	
4	2.422598e+08	POINT (-82.04167 23.12500)	23.125	-82.041667	192	

	impf_
0	1
1	1
2	1
3	1
4	1

[3]:
```
# not needed for impact calculations
# visualize the define exposure
exp_lp.plot_raster()
print('\n Raster properties exposures:', exp_lp.meta)
```

```
2021-04-30 13:11:13,034 - climada.util.coordinates - INFO - Raster from resolution 0.
↪0833333333333286 to 0.0833333333333286.

 Raster properties exposures: {'width': 129, 'height': 41, 'crs': 'EPSG:4326', 'transform
↪': Affine(0.0833333333333286, 0.0, -84.91666666666669,
       0.0, 0.0833333333333286, 19.833333333333336)}
```



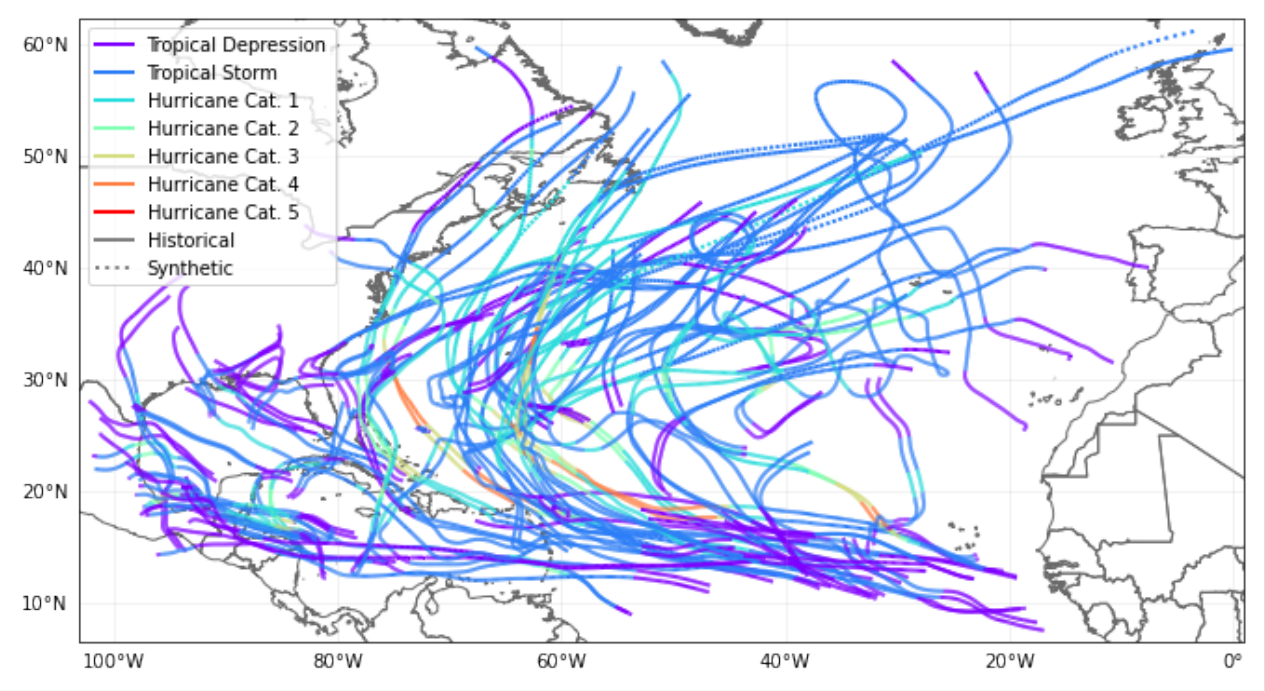
Define the hazard

Let us define a tropical cyclone hazard using the TropCyclone and TCTracks modules.

[4]:
```
from climada.hazard import TCTracks, TropCyclone, Centroids

# Load histrocial tropical cyclone tracks from ibtracs over the North Atlantic basin␣
↪between 2010-2012
ibtracks_na = TCTracks.from_ibtracs_netcdf(provider='usa', basin='NA', year_range=(2010,␣
↪2012), correct_pres=True)
print('num tracks hist:', ibtracks_na.size)

ibtracks_na.equal_timestep(0.5)  # Interpolation to make the track smooth and to allow␣
↪applying calc_perturbed_trajectories
# Add randomly generated tracks using the calc_perturbed_trajectories method (1 per␣
↪historical track)
ibtracks_na.calc_perturbed_trajectories(nb_synth_tracks=1)
print('num tracks hist+syn:', ibtracks_na.size)
```

```
2021-10-19 16:49:12,745 - climada.hazard.tc_tracks - WARNING - `correct_pres` is␣
↪deprecated. Use `estimate_missing` instead.
num tracks hist: 60
num tracks hist+syn: 120
```

[5]:
```
# not needed for calculations
# visualize tracks
ax = ibtracks_na.plot()
ax.get_legend()._loc = 2
```



From the tracks, we generate the hazards (the tracks are only the coordinates of the center of the cyclones, the full cyclones however affects a region around the tracks).

First thing we define the set of centroids which are geographical points where the hazard has a defined value. In our case, we want to define windspeeds from the tracks.

Remember: In the impact computations, for each exposure geographical point, one must assign a centroid from the hazard. By default, each exposure is assigned to the closest centroid from the hazard. But one can also define manually which centroid is assigned to which exposure point.

Examples: - Define the exposures from a given source (e.g., raster of asset values from LitPop). Define the hazard centroids from the exposures' geolocations (e.g. compute Tropical Cyclone windspeed at each raster point and assign centroid to each raster point). - Define the exposures from a given source (e.g. houses position and value). Define the hazard from a given source (e.g. where lanslides occur). Use a metric to assign to each exposures point a hazard centroid (all houses in a radius of 5km around the lanslide are assigned to this centroid, if a house is within 5km of two landslides, choose the closest one). - Define a geographical raster. Define the exposures value on this raster. Define the hazard centroids on the geographical raster.

We shall pursue with the first case (Litpop + TropicalCyclone)

Hint: computing the wind speeds in many locations for many tc tracks is a computationally costly operation. Thus, we should define centroids only where we also have an exposure.

[6]:
```
# Define the centroids from the exposures position
lat = exp_lp.gdf['latitude'].values
lon = exp_lp.gdf['longitude'].values
centrs = Centroids.from_lat_lon(lat, lon)
centrs.check()
```

[7]:
```
# Using the tracks, compute the windspeed at the location of the centroids
tc = TropCyclone.from_tracks(ibtracks_na, centroids=centrs)
tc.check()
```

Hint: The operation of computing the windspeed in different location is in general computationally expensive. Hence,

if you have a lot of tropical cyclone tracks, you should first make sure that all your tropical cyclones actually affect your exposure (remove those that don't). Then, be careful when defining the centroids. For a large country like China, there is no need for centroids 500km inlans (no tropical cyclones gets so far).

Impact function

For Tropical Cyclones, some calibrated default impact functions exist. Here we will use the one from Emanuel (2011).

[8]:
```
from climada.entity import ImpactFuncSet, ImpfTropCyclone
# impact function TC
impf_tc = ImpfTropCyclone.from_emanuel_usa()

# add the impact function to an Impact function set
impf_set = ImpactFuncSet()
impf_set.append(impf_tc)
impf_set.check()
```

```
2021-10-19 16:49:44,667 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
```

Recall that the exposures, hazards and impact functions must be matched in the impact calculations. Here it is simple, since there is a single impact function for all the hazards. We must simply make sure that the exposure is assigned this impact function through renaming the `impf_` column from the hazard type of the impact function in the impact function set and set the values of the column to the id of the impact function.

[9]:
```
# Get the hazard type and hazard id
[haz_type] = impf_set.get_hazard_types()
[haz_id] = impf_set.get_ids()[haz_type]
print(f"hazard type: {haz_type}, hazard id: {haz_id}")
```

```
hazard type: TC, hazard id: 1
```

[10]:
```
# Exposures: rename column and assign id
exp_lp.gdf.rename(columns={"impf_": "impf_" + haz_type}, inplace=True)
exp_lp.gdf['impf_' + haz_type] = haz_id
exp_lp.check()
exp_lp.gdf.head()
```

[10]:

	value	geometry	latitude	longitude	region_id	impf_TC
0	4.730991e+10	POINT (-82.37500 23.12500)	23.125	-82.375000	192	1
1	2.990876e+10	POINT (-82.29167 23.12500)	23.125	-82.291667	192	1
2	6.839380e+09	POINT (-82.20833 23.12500)	23.125	-82.208333	192	1
3	1.925246e+09	POINT (-82.12500 23.12500)	23.125	-82.125000	192	1
4	2.422598e+08	POINT (-82.04167 23.12500)	23.125	-82.041667	192	1

Impact computation

We are finally ready for the impact computation. This is the simplest step. Just give the exposure, impact function and hazard to the Impact.calc() method.

Note: we did not specifically assign centroids to the exposures. Hence, the default is used - each exposure is associated with the closest centroids. Since we defined the centroids from the exposures, this is a one-to-one mapping.

Note: we did not define an `Entity` in this impact calculations. Recall that `Entity` is a container class for *Exposures*, *Impact Functions*, *Discount Rates* and *Measures*. Since we had only one Exposure and one Impact Function, the container would not have added any value, but for more complex projects, the Entity class is very useful.

[11]:
```
# Compute impact
from climada.engine import Impact
imp = Impact()
imp.calc(exp_lp, impf_set, tc, save_mat=False) #Do not save the results geographically␣
↪resolved (only aggregate values)
```

[12]:
```
exp_lp.gdf
```

[12]:
```
             value                    geometry   latitude  longitude  \
0     4.730991e+10  POINT (-82.37500 23.12500)  23.125000 -82.375000
1     2.990876e+10  POINT (-82.29167 23.12500)  23.125000 -82.291667
2     6.839380e+09  POINT (-82.20833 23.12500)  23.125000 -82.208333
3     1.925246e+09  POINT (-82.12500 23.12500)  23.125000 -82.125000
4     2.422598e+08  POINT (-82.04167 23.12500)  23.125000 -82.041667
...            ...                         ...        ...        ...
1383  6.191982e+06  POINT (-78.29167 22.45833)  22.458333 -78.291667
1384  8.882190e+05  POINT (-79.20833 22.62500)  22.625000 -79.208333
1385  7.629772e+05  POINT (-79.62500 22.79167)  22.791667 -79.625000
1386  9.065058e+05  POINT (-79.45833 22.70833)  22.708333 -79.458333
1387  2.411896e+05  POINT (-80.79167 23.20833)  23.208333 -80.791667

      region_id  impf_TC  centr_TC
0           192        1         0
1           192        1         1
2           192        1         2
3           192        1         3
4           192        1         4
...         ...      ...       ...
1383        192        1      1383
1384        192        1      1384
1385        192        1      1385
1386        192        1      1386
1387        192        1      1387

[1388 rows x 7 columns]
```

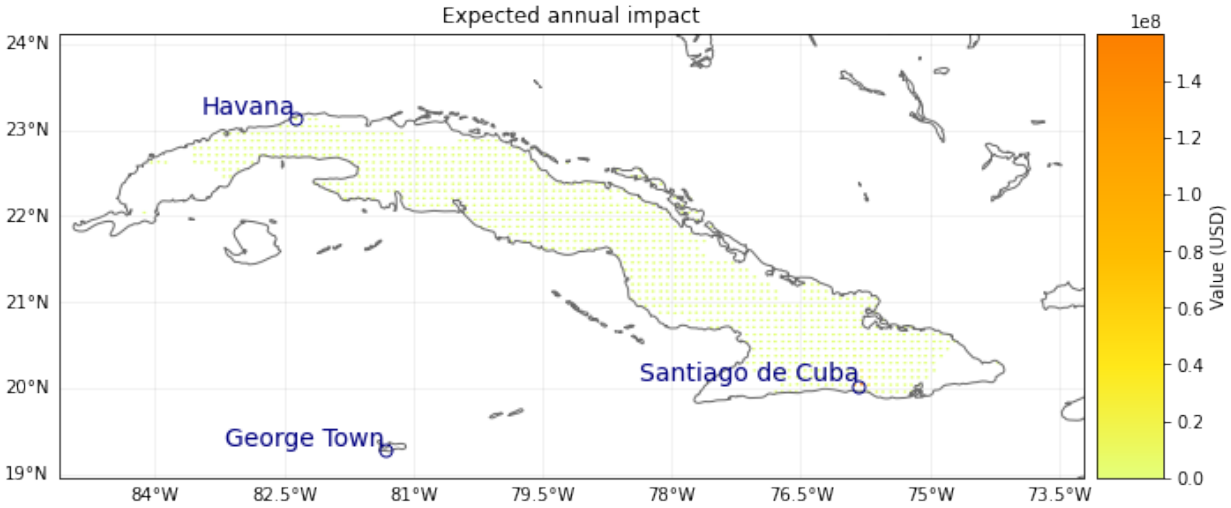
For example we can now obtain the aggregated average annual impact or plot the average annual impact in each exposure location.

[13]:
```
print(f"Aggregated average annual impact: {round(imp.aai_agg,0)} $")
```

```
Aggregated average annual impact: 563366225.0 $
```

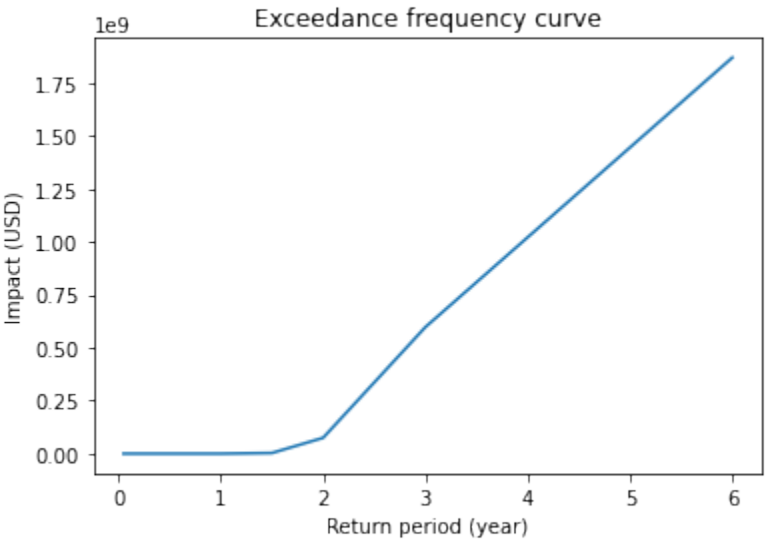
[14]:
```
imp.plot_hexbin_eai_exposure(buffer=1)
```

[14]:
```
<GeoAxesSubplot:title={'center':'Expected annual impact'}>
```



[15]:
```
# Compute exceedance frequency curve
freq_curve = imp.calc_freq_curve()
freq_curve.plot()
```

[15]:
```
<AxesSubplot:title={'center':'Exceedance frequency curve'}, xlabel='Return period (year)
↪', ylabel='Impact (USD)'>
```



Quick examples - points, raster, custom

User defined point exposure and Tropical Cyclone hazard

[16]:
```
%matplotlib inline
# EXAMPLE: POINT EXPOSURES WITH POINT HAZARD
import numpy as np
from climada.entity import Exposures, ImpactFuncSet, IFTropCyclone
from climada.hazard import Centroids, TCTracks, TropCyclone
from climada.engine import Impact

# Set Exposures in points
exp_pnt = Exposures(crs='epsg:4326') #set coordinate system
exp_pnt.gdf['latitude'] = np.array([21.899326, 21.960728, 22.220574, 22.298390, 21.
↪787977, 21.787977, 21.981732])
exp_pnt.gdf['longitude'] = np.array([88.307422, 88.565362, 88.378337, 87.806356, 88.
↪348835, 88.348835, 89.246521])
exp_pnt.gdf['value'] = np.array([1.0e5, 1.2e5, 1.1e5, 1.1e5, 2.0e5, 2.5e5, 0.5e5])
exp_pnt.check()
exp_pnt.plot_scatter(buffer=0.05)

# Set Hazard in Exposures points
# set centroids from exposures coordinates
centr_pnt = Centroids.from_lat_lon(exp_pnt.gdf.latitude.values, exp_pnt.gdf.longitude.
↪values, exp_pnt.crs)
# compute Hazard in that centroids
tr_pnt = TCTracks.from_ibtracs_netcdf(storm_id='2007314N10093')
tc_pnt = TropCyclone.from_tracks(tr_pnt, centroids=centr_pnt)
tc_pnt.check()
ax_pnt = tc_pnt.centroids.plot(c=np.array(tc_pnt.intensity[0,:].todense()).squeeze()) #␣
↪plot intensity per point
ax_pnt.get_figure().colorbar(ax_pnt.collections[0], fraction=0.0175, pad=0.02).set_label(
↪'Intensity (m/s)') # add colorbar

# Set impact function
impf_pnt = ImpactFuncSet()
impf_tc = ImpfTropCyclone.from_emanuel_usa()
impf_pnt.append(impf_tc)
impf_pnt.check()

# Get the hazard type and hazard id
[haz_type] = impf_set.get_hazard_types()
[haz_id] = impf_set.get_ids()[haz_type]
# Exposures: rename column and assign id
exp_lp.gdf.rename(columns={"impf_": "impf_" + haz_type}, inplace=True)
exp_lp.gdf['impf_' + haz_type] = haz_id
exp_lp.gdf.head()

# Compute Impact
imp_pnt = Impact()
imp_pnt.calc(exp_pnt, impf_pnt, tc_pnt)
# nearest neighbor of exposures to centroids gives identity
```

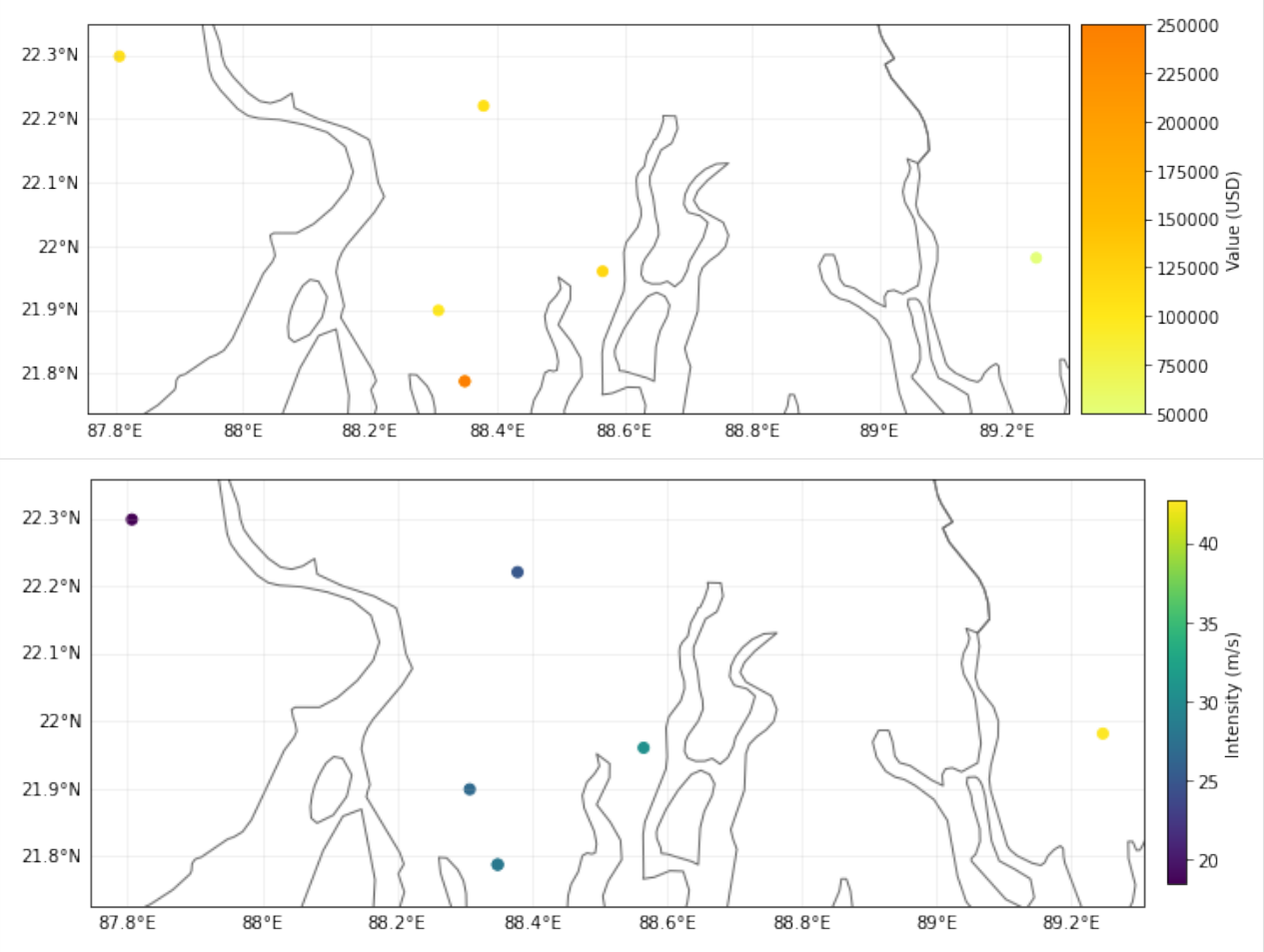
(continues on next page)

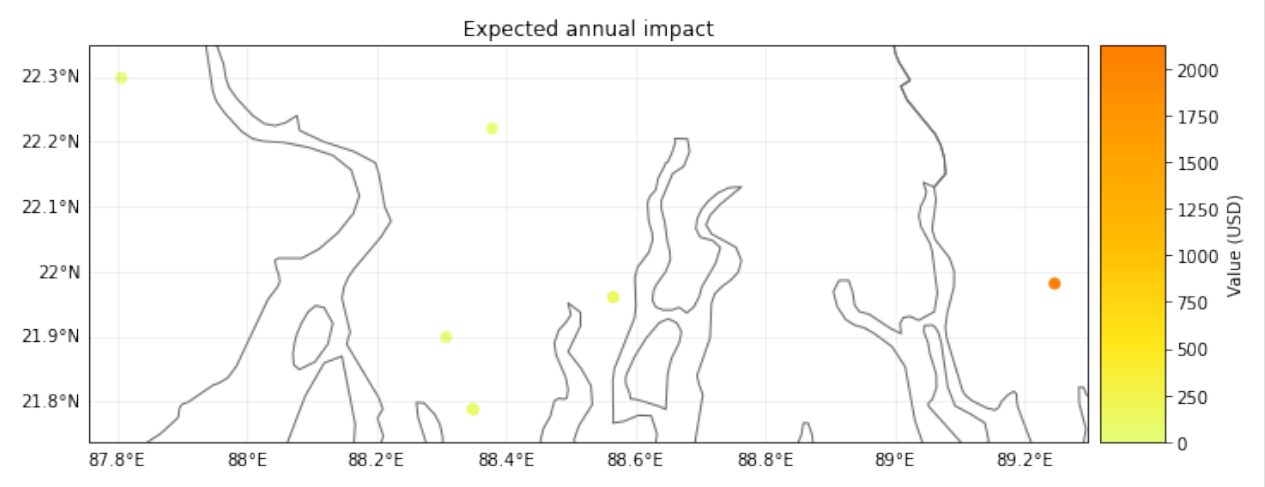
(continued from previous page)

```
print('Nearest neighbor hazard.centroids indexes for each exposure:', exp_pnt.gdf.centr_
↪TC.values)
imp_pnt.plot_scatter_eai_exposure(ignore_zero=False, buffer=0.05)
```

```
2021-10-19 16:49:49,046 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
Nearest neighbor hazard.centroids indexes for each exposure: [0 1 2 3 4 5 6]
```

[16]: <GeoAxesSubplot:title={'center':'Expected annual impact'}>





Raster from file

[17]:
```
# EXAMPLE: RASTER EXPOSURES WITH RASTER HAZARD
from rasterio.warp import Resampling
from climada.entity import LitPop, ImpactFuncSet, ImpactFunc
from climada.hazard import Hazard
from climada.engine import Impact
from climada.util.constants import HAZ_DEMO_FL

# Exposures belonging to a raster (the raser information is contained in the meta␣
↪attribute)
exp_ras = LitPop.from_countries(countries=['VEN'], res_arcsec=300, fin_mode='income_group
↪')
exp_ras.gdf.reset_index()
exp_ras.check()
exp_ras.plot_raster()
print('\n Raster properties exposures:', exp_ras.meta)

# Initialize hazard object with haz_type = 'FL' (for Flood)
hazard_type='FL'
# Load a previously generated (either with CLIMADA or other means) hazard
# from file (HAZ_DEMO_FL) and resample the hazard raster to the exposures' ones
# Hint: check how other resampling methods affect to final impact
haz_ras = Hazard.from_raster([HAZ_DEMO_FL], haz_type=hazard_type, dst_crs=exp_ras.meta[
↪'crs'], transform=exp_ras.meta['transform'],
                             width=exp_ras.meta['width'], height=exp_ras.meta['height'],
                             resampling=Resampling.nearest)
haz_ras.intensity[haz_ras.intensity==-9999] = 0 # correct no data values
haz_ras.check()
haz_ras.plot_intensity(1)
print('Raster properties centroids:', haz_ras.centroids.meta)

# Set dummy impact function
impf_dum = ImpactFunc()
impf_dum.id = haz_id
```

(continues on next page)

(continued from previous page)

```
impf_dum.name = 'dummy'
impf_dum.intensity_unit = 'm'
impf_dum.haz_type = hazard_type
impf_dum.intensity = np.linspace(0, 10, 100)
impf_dum.mdd = np.linspace(0, 10, 100)
impf_dum.paa = np.ones(impf_dum.intensity.size)
# Add the impact function to the impact function set
impf_ras = ImpactFuncSet()
impf_ras.append(impf_dum)
impf_ras.check()

# Exposures: rename column and assign id
exp_lp.gdf.rename(columns={"impf_": "impf_" + hazard_type}, inplace=True)
exp_lp.gdf['impf_' + haz_type] = haz_id
exp_lp.gdf.head()

# Compute impact
imp_ras = Impact()
imp_ras.calc(exp_ras, impf_ras, haz_ras, save_mat=False)
# nearest neighbor of exposures to centroids is not identity because litpop does not␣
↪contain data outside the country polygon
print('\n Nearest neighbor hazard.centroids indexes for each exposure:', exp_ras.gdf.
↪centr_FL.values)
imp_ras.plot_raster_eai_exposure()
```

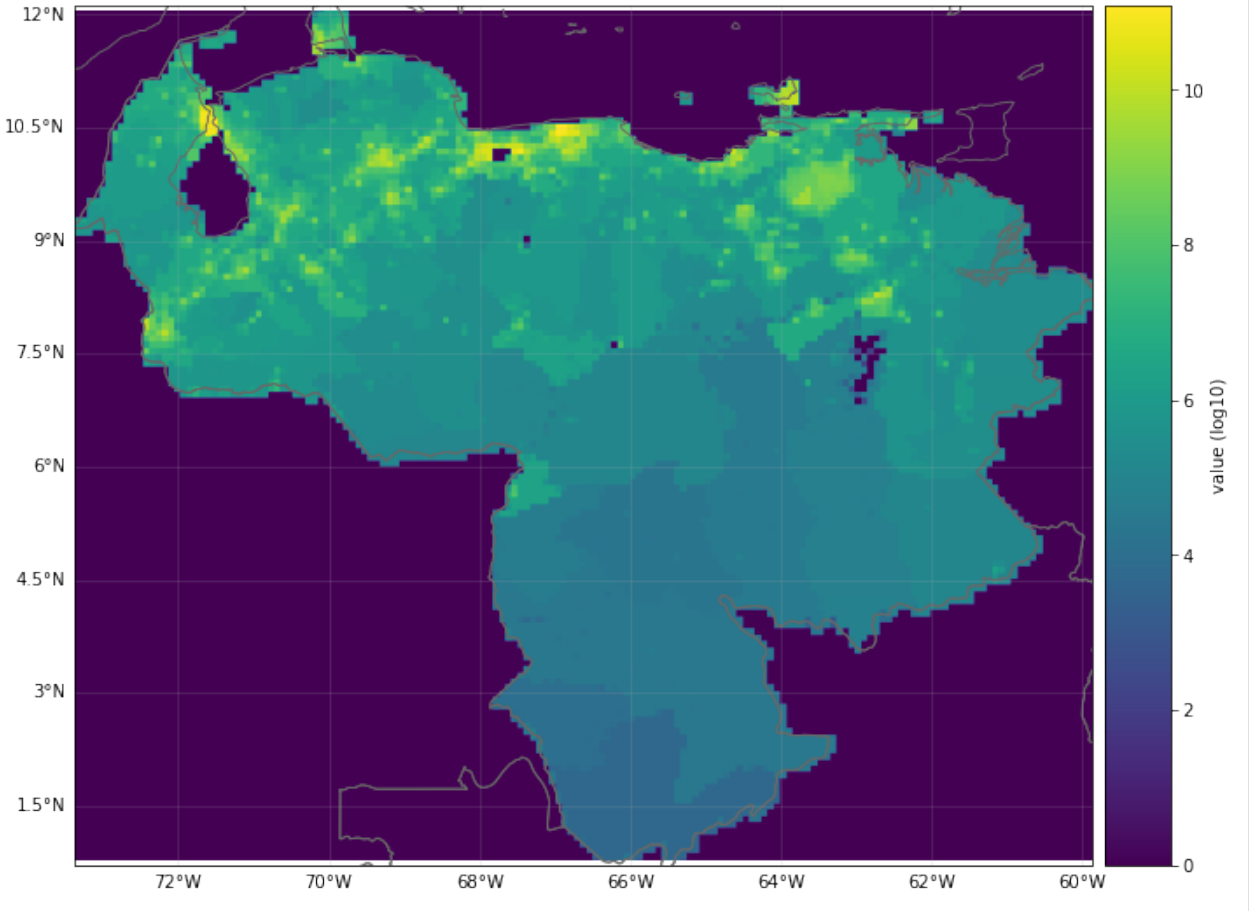
```
2021-10-19 16:49:51,864 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020

 Raster properties exposures: {'width': 163, 'height': 138, 'crs': <Geographic 2D CRS:␣
↪EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- undefined
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
, 'transform': Affine(0.08333333000000209, 0.0, -73.41666666500001,
       0.0, -0.08333332999999987, 12.166666665)}
Raster properties centroids: {'driver': 'GSBG', 'dtype': 'float32', 'nodata': 1.
↪701410009187828e+38, 'width': 163, 'height': 138, 'count': 1, 'crs': <Geographic 2D␣
↪CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- undefined
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
```

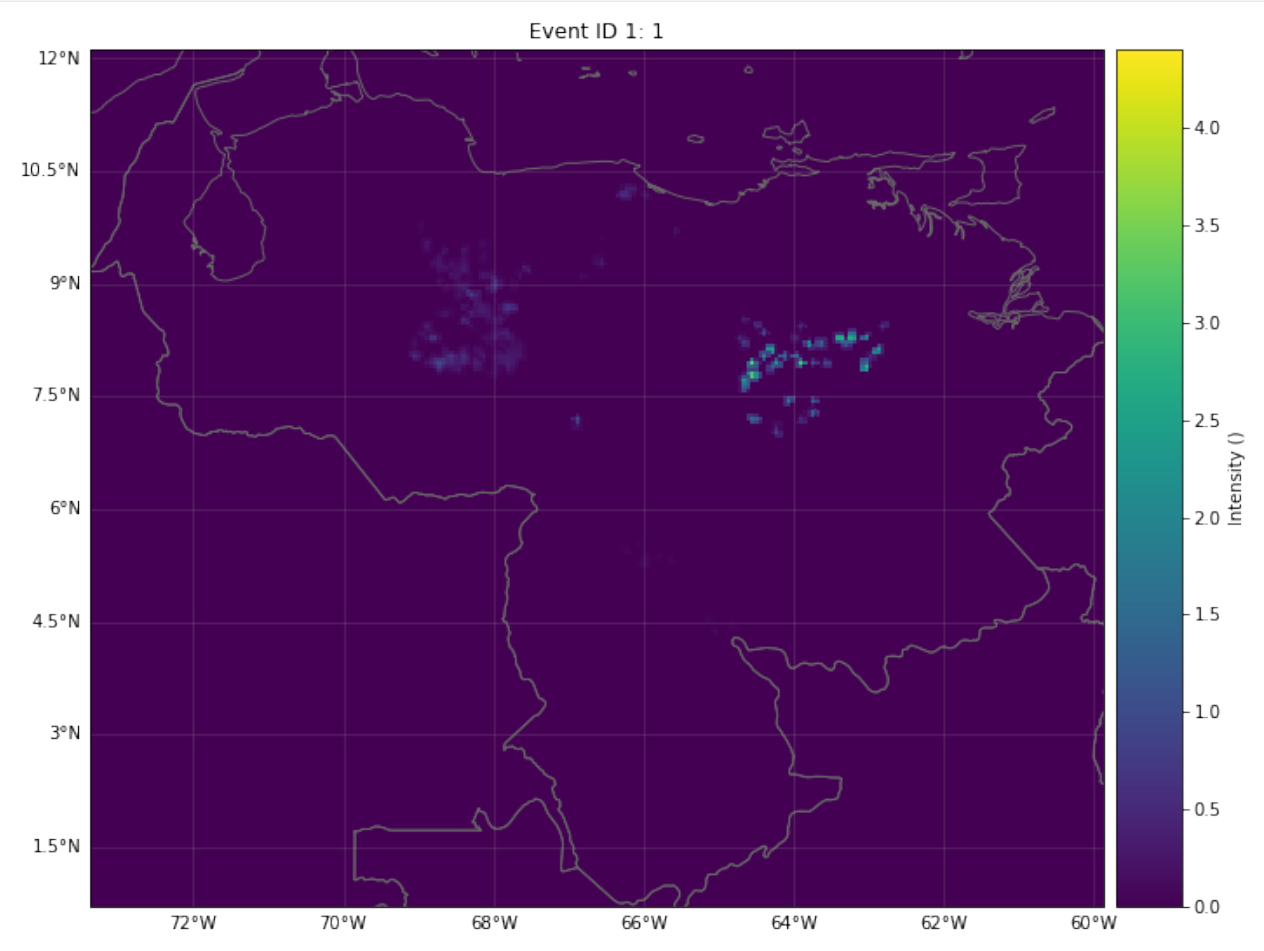
(continues on next page)

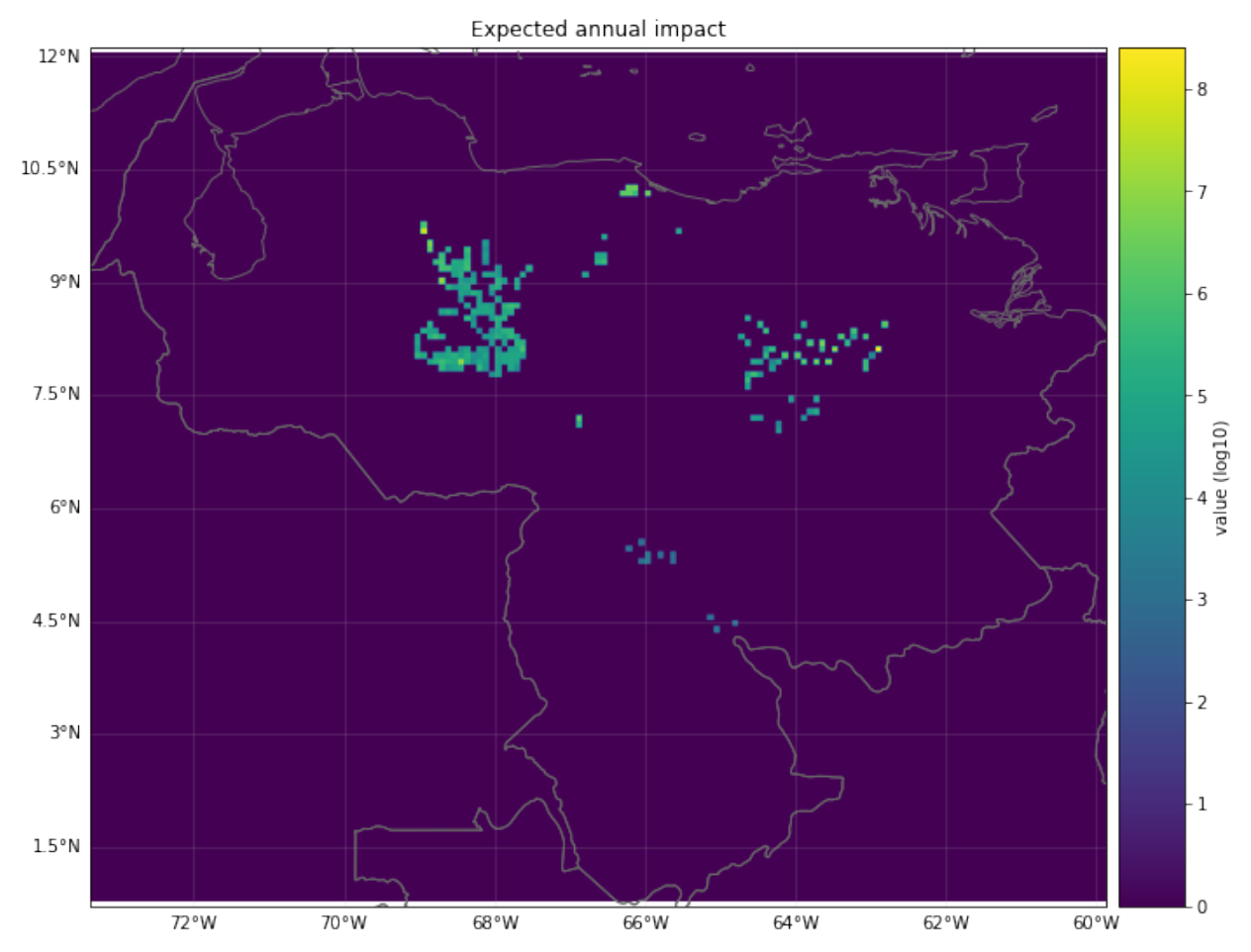
(continued from previous page)

```
- Prime Meridian: Greenwich
, 'transform': Affine(0.08333333000000209, 0.0, -73.41666666500001,
       0.0, -0.08333332999999987, 12.166666665)}
2021-10-19 16:50:02,231 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.

 Nearest neighbor hazard.centroids indexes for each exposure: [  39   40   41 ... 3551␣
↪2721  594]
```

[17]: <GeoAxesSubplot:title={'center':'Expected annual impact'}>




Event ID 1: 1
12°N
10.5°N
9°N
7.5°N
6°N
4.5°N
3°N
1.5°N
72°W
70°W
68°W
66°W
64°W
62°W
60°W
Intensity ()
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0



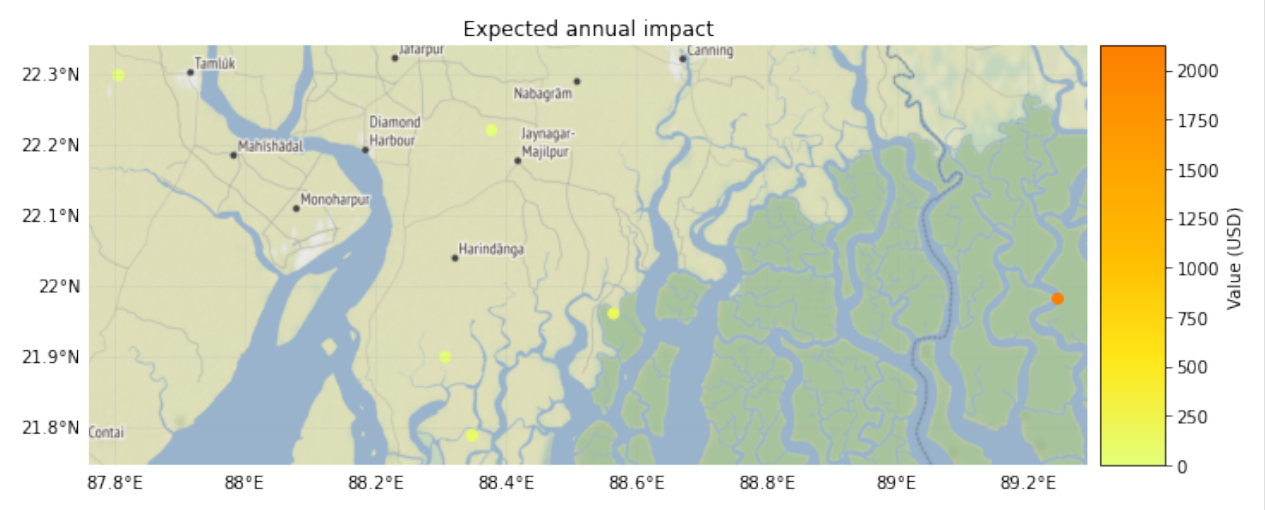
VISUALIZATION

Making plots

The expected annual impact per exposure can be visualized through different methods: `plot_hexbin_eai_exposure()`, `plot_scatter_eai_exposur()`, `plot_raster_eai_exposure()` and `plot_basemap_eai_exposure()` (similarly as with `Exposures`).

[18]:
```
imp_pnt.plot_basemap_eai_exposure(buffer=5000)
```

[18]:
```
<GeoAxesSubplot:title={'center':'Expected annual impact'}>
```



Making videos

Given a fixed exposure and impact functions, a sequence of hazards can be visualized hitting the exposures.

[19]:
```
# exposure
from climada.entity import add_sea
from climada_petals.entity import BlackMarble

exp_video = BlackMarble()
exp_video.set_countries(['Cuba'], 2016, res_km=2.5)
exp_video.check()

# impact function
impf_def = ImpfTropCyclone.from_emanuel_usa()
impfs_video = ImpactFuncSet()
impfs_video.append(impf_def)
impfs_video.check()

# compute sequence of hazards using TropCyclone video_intensity method
exp_sea = add_sea(exp_video, (100, 5))
centr_video = Centroids.from_lat_lon(exp_sea.gdf.latitude.values, exp_sea.gdf.longitude.
↪values)
centr_video.check()

track_name = '2017242N16333'
tr_irma = TCTracks.from_ibtracs_netcdf(provider='usa', storm_id=track_name) # IRMA 2017

tc_video = TropCyclone()
tc_list, _ = tc_video.video_intensity(track_name, tr_irma, centr_video) # empty file␣
↪name to not to write the video

# generate video of impacts
file_name='./results/irma_imp_fl.gif'
imp_video = Impact()
imp_list = imp_video.video_direct_impact(exp_video, impfs_video, tc_list, file_name)
```

(continues on next page)

(continued from previous page)

```
2021-04-30 13:13:09,080 - climada.entity.exposures.base - INFO - meta set to default␣
↪value {}
2021-04-30 13:13:09,086 - climada.entity.exposures.base - INFO - tag set to default␣
↪value  File:
 Description:
2021-04-30 13:13:09,088 - climada.entity.exposures.base - INFO - ref_year set to default␣
↪value 2018
2021-04-30 13:13:09,097 - climada.entity.exposures.base - INFO - value_unit set to␣
↪default value USD
2021-04-30 13:13:09,100 - climada.entity.exposures.base - INFO - crs set to default␣
↪value: EPSG:4326
2021-04-30 13:13:10,128 - climada.util.finance - INFO - GDP CUB 2016: 9.137e+10.
2021-04-30 13:13:10,197 - climada.util.finance - INFO - Income group CUB 2016: 3.
2021-04-30 13:13:10,198 - climada.entity.exposures.black_marble - INFO - Nightlights␣
↪from NASA's earth observatory for year 2016.
2021-04-30 13:13:18,224 - climada.entity.exposures.black_marble - INFO - Processing␣
↪country Cuba.
2021-04-30 13:13:19,316 - climada.entity.exposures.black_marble - INFO - Generating␣
↪resolution of approx 2.5 km.
2021-04-30 13:13:19,478 - climada.entity.exposures.base - INFO - meta set to default␣
↪value {}
2021-04-30 13:13:19,479 - climada.entity.exposures.base - INFO - tag set to default␣
↪value  File:
 Description:
2021-04-30 13:13:19,480 - climada.entity.exposures.base - INFO - ref_year set to default␣
↪value 2018
2021-04-30 13:13:19,481 - climada.entity.exposures.base - INFO - value_unit set to␣
↪default value USD
2021-04-30 13:13:19,485 - climada.entity.exposures.base - INFO - crs set to default␣
↪value: EPSG:4326
2021-04-30 13:13:19,511 - climada.entity.exposures.base - INFO - meta set to default␣
↪value {}
2021-04-30 13:13:19,522 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2021-04-30 13:13:19,525 - climada.entity.exposures.base - INFO - category_id not set.
2021-04-30 13:13:19,528 - climada.entity.exposures.base - INFO - cover not set.
2021-04-30 13:13:19,529 - climada.entity.exposures.base - INFO - deductible not set.
2021-04-30 13:13:19,530 - climada.entity.exposures.base - INFO - geometry not set.
2021-04-30 13:13:19,532 - climada.entity.exposures.base - INFO - centr_ not set.
2021-04-30 13:13:19,534 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
2021-04-30 13:13:19,536 - climada.entity.exposures.base - INFO - Adding sea at 5 km␣
↪resolution and 100 km distance from coast.
2021-04-30 13:13:20,980 - climada.hazard.tc_tracks - INFO - Progress: 100%
2021-04-30 13:13:21,016 - climada.hazard.centroids.centr - INFO - Convert centroids to␣
↪GeoSeries of Point shapes.
2021-04-30 13:13:33,062 - climada.util.coordinates - INFO - dist_to_coast: UTM 32616 (1/
↪3)
2021-04-30 13:13:40,527 - climada.util.coordinates - INFO - dist_to_coast: UTM 32617 (2/
↪3)
```

(continues on next page)

(continued from previous page)

```
2021-04-30 13:14:01,057 - climada.util.coordinates - INFO - dist_to_coast: UTM 32618 (3/
↪3)
2021-04-30 13:14:11,473 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 22776␣
↪coastal centroids.
2021-04-30 13:14:11,509 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,527 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 25701␣
↪coastal centroids.
2021-04-30 13:14:11,563 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,580 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 29239␣
↪coastal centroids.
2021-04-30 13:14:11,633 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,648 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 32187␣
↪coastal centroids.
2021-04-30 13:14:11,691 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,710 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 34921␣
↪coastal centroids.
2021-04-30 13:14:11,763 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,777 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 37244␣
↪coastal centroids.
2021-04-30 13:14:11,831 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,845 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 39418␣
↪coastal centroids.
2021-04-30 13:14:11,897 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,910 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 42155␣
↪coastal centroids.
2021-04-30 13:14:11,966 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:11,981 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 43662␣
↪coastal centroids.
2021-04-30 13:14:12,054 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,067 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 45523␣
↪coastal centroids.
2021-04-30 13:14:12,132 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,151 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 47105␣
↪coastal centroids.
2021-04-30 13:14:12,211 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,227 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 48082␣
↪coastal centroids.
2021-04-30 13:14:12,291 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,304 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 48019␣
↪coastal centroids.
2021-04-30 13:14:12,375 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,389 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 47081␣
↪coastal centroids.
2021-04-30 13:14:12,457 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,469 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 45784␣
↪coastal centroids.
2021-04-30 13:14:12,534 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,547 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 44307␣
↪coastal centroids.
2021-04-30 13:14:12,598 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,613 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 43081␣
↪coastal centroids.
```

(continues on next page)

(continued from previous page)

```
2021-04-30 13:14:12,676 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,691 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 42086␣
↪coastal centroids.
2021-04-30 13:14:12,751 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,768 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 41313␣
↪coastal centroids.
2021-04-30 13:14:12,827 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,844 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 40713␣
↪coastal centroids.
2021-04-30 13:14:12,896 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,911 - climada.hazard.trop_cyclone - INFO - Mapping 1 tracks to 40023␣
↪coastal centroids.
2021-04-30 13:14:12,964 - climada.hazard.trop_cyclone - INFO - Progress: 100%
2021-04-30 13:14:12,978 - climada.entity.exposures.base - INFO - Matching 21923␣
↪exposures with 49817 centroids.
2021-04-30 13:14:13,046 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,047 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,054 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,059 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,060 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,067 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,071 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,073 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,082 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,085 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,089 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,097 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,100 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,101 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,109 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,117 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,123 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,132 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,135 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
```

(continues on next page)

(continued from previous page)

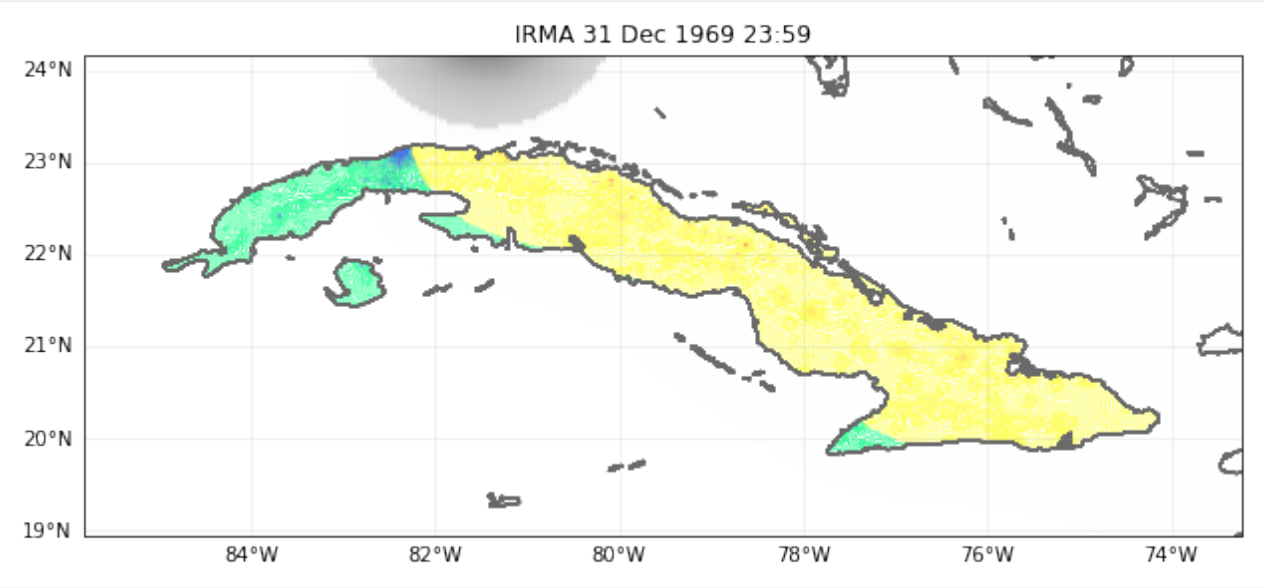
```
2021-04-30 13:14:13,139 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,147 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,151 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,153 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,163 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,168 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,171 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,180 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,184 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,187 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,198 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,202 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,205 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,214 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,218 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,219 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,228 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,233 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,234 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,247 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,251 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,253 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,262 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,266 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,267 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,276 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
```

(continues on next page)

(continued from previous page)

```
2021-04-30 13:14:13,279 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,281 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,288 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,291 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,293 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,300 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,304 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,305 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,314 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,320 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,321 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,328 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,332 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,333 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,342 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2021-04-30 13:14:13,345 - climada.engine.impact - INFO - Calculating damage for 14654␣
↪assets (>0) and 1 events.
2021-04-30 13:14:13,346 - climada.entity.exposures.base - INFO - No specific impact␣
↪function column found for hazard TC. Using the anonymous 'impf_' column.
2021-04-30 13:14:13,355 - climada.engine.impact - INFO - Generating video ./results/irma_
↪imp_fl.gif
```

```
22it [09:40, 26.39s/it]
```



5.12 Impact Data functionalities

Import data from EM-DAT CSV file and populate Impact()-object with the data.

The core functionality of the module is to read disaster impact data as downloaded from the International Disaster Database EM-DAT (www.emdat.be) and produce a CLIMADA Impact()-instance from it. The purpose is to make impact data easily available for comparison with simulated impact inside CLIMADA, e.g. for calibration purposes.

5.12.1 Data Source

The International Disaster Database EM-DAT www.emdat.be

Download: https://public.emdat.be/ (register for free and download data to continue)

5.12.2 Most important functions

- clean_emdat_df: read CSV from EM-DAT into a DataFrame and clean up.
- emdat_to_impact: create Impact-instance populated with impact data from EM-DAT data (CSV).
- emdat_countries_by_hazard: get list of countries affected by a certain haazrd (disaster (sub-)type) in EM-DAT.
- emdat_impact_yearlysum: create DataFrame with impact from EM-DAT summed per country and year.

5.12.3 Demo data

The demo data used hee (demo_emdat_impact_data_2020.csv) contains entries for the disaster subtype "Tropical cyclone" from 2000 to 2020.

[2]:
```
"""Load required packages and set path to CSV-file from EM-DAT"""

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from climada.util.constants import DEMO_DIR
from climada.engine.impact_data import emdat_countries_by_hazard, \
    emdat_impact_yearlysum, emdat_to_impact, clean_emdat_df

# set path to CSV file downloaded from https://public.emdat.be :
emdat_file_path = DEMO_DIR.joinpath('demo_emdat_impact_data_2020.csv')
```

clean_emdat_df()

read CSV from EM-DAT into a DataFrame and clean up.

Use the parameters countries, hazard, and year_range to filter. These parameters are the same for most functions shown here.

[12]:
```
"""Create DataFrame df with EM-DAT entries of tropical cyclones in Thailand and Viet Nam␣
↪in the years 2005 and 2006"""

df = clean_emdat_df(emdat_file_path, countries=['THA', 'Viet Nam'], hazard=['TC'], \
                            year_range=[2005, 2006])
print(df)
```

```
          Dis No  Year  Seq Disaster Group Disaster Subgroup Disaster Type  \
0  2005-0540-VNM  2005  540        Natural    Meteorological         Storm
1  2005-0540-THA  2005  540        Natural    Meteorological         Storm
2  2005-0536-VNM  2005  536        Natural    Meteorological         Storm
3  2005-0611-VNM  2005  611        Natural    Meteorological         Storm
4  2006-0362-VNM  2006  362        Natural    Meteorological         Storm
5  2006-0648-VNM  2006  648        Natural    Meteorological         Storm
6  2006-0251-VNM  2006  251        Natural    Meteorological         Storm
7  2006-0517-VNM  2006  517        Natural    Meteorological         Storm

   Disaster Subtype  Disaster Subsubtype          Event Name Entry Criteria  \
0  Tropical cyclone                  NaN              Damrey          Kill
1  Tropical cyclone                  NaN              Damrey          Kill
2  Tropical cyclone                  NaN             Vicente          Kill
3  Tropical cyclone                  NaN        Kai Tak (21)          Kill
4  Tropical cyclone                  NaN               Bilis          Kill
5  Tropical cyclone                  NaN     Durian (Reming)          Kill
6  Tropical cyclone                  NaN     Chanchu (Caloy)          Kill
7  Tropical cyclone                  NaN  Xangsane (Milenyo)          Kill

   ... End Day Total Deaths No Injured No Affected No Homeless Total Affected  \
```

(continues on next page)

(continued from previous page)

```
0  ...     30.0          75.0       28.0    337632.0          NaN       337660.0
1  ...     30.0          10.0        NaN      2000.0          NaN         2000.0
2  ...     19.0           8.0        NaN      8500.0          NaN         8500.0
3  ...      4.0          20.0        NaN     15000.0          NaN        15000.0
4  ...     19.0          17.0        NaN         NaN       2000.0         2000.0
5  ...      8.0          95.0     1360.0    975000.0     250000.0      1226360.0
6  ...     17.0         204.0        NaN    600000.0          NaN       600000.0
7  ...      6.0          71.0      525.0   1368720.0      98680.0      1467925.0

  Reconstruction Costs ('000 US$) Insured Damages ('000 US$)  \
0                             NaN                         NaN
1                             NaN                         NaN
2                             NaN                         NaN
3                             NaN                         NaN
4                             NaN                         NaN
5                             NaN                         NaN
6                             NaN                         NaN
7                             NaN                         NaN

  Total Damages ('000 US$)        CPI
0                 219250.0  76.388027
1                  20000.0  76.388027
2                  20000.0  76.388027
3                  11000.0  76.388027
4                      NaN  78.852256
5                 456000.0  78.852256
6                      NaN  78.852256
7                 624000.0  78.852256

[8 rows x 43 columns]
```

emdat_countries_by_hazard()

Pick a hazard and a year range to get a list of countries affected from the EM-DAT data.

[2]:
```
"""emdat_countries_by_hazard: get lists of countries impacted by tropical cyclones from␣
↪2010 to 2019"""

iso3_codes, country_names = emdat_countries_by_hazard(emdat_file_path, hazard='TC', year_
↪range=(2010, 2019))

print(country_names)

print(iso3_codes)
```

```
['China', 'Dominican Republic', 'Antigua and Barbuda', 'Fiji', 'Australia', 'Bangladesh',
↪ 'Belize', 'Barbados', 'Cook Islands', 'Canada', 'Bahamas', 'Guatemala', 'Jamaica',
↪'Saint Lucia', 'Madagascar', 'Mexico', "Korea, Democratic People's Republic of", 'El␣
↪Salvador', 'Myanmar', 'French Polynesia', 'Solomon Islands', 'Taiwan, Province of China
↪', 'India', 'United States of America', 'Honduras', 'Haiti', 'Pakistan', 'Philippines',
↪ 'Hong Kong', 'Korea, Republic of', 'Nicaragua', 'Oman', 'Japan', 'Puerto Rico',
↪'Thailand', 'Martinique', 'Papua New Guinea', 'Tonga', 'Venezuela, Bolivarian Republic␣
↪of', 'Viet Nam', 'Saint Vincent and the Grenadines', 'Vanuatu', 'Dominica', 'Cuba',
↪'Comoros', 'Mozambique', 'Malawi', 'Samoa', 'South Africa', 'Sri Lanka', 'Palau',
↪'Wallis and Futuna', 'Somalia', 'Seychelles', 'Réunion', 'Kiribati', 'Cabo Verde',
↪'Micronesia, Federated States of', 'Panama', 'Costa Rica', 'Yemen', 'Tuvalu',
↪'Northern Mariana Islands', 'Colombia', 'Anguilla', 'Djibouti', 'Cambodia', 'Macao',
↪'Indonesia', 'Guadeloupe', 'Turks and Caicos Islands', 'Saint Kitts and Nevis', "Lao␣
↪People's Democratic Republic", 'Mauritius', 'Marshall Islands', 'Portugal', 'Virgin␣
↪Islands, U.S.', 'Zimbabwe', 'Saint Barthélemy', 'Virgin Islands, British', 'Saint␣
↪Martin (French part)', 'Sint Maarten (Dutch part)', 'Tanzania, United Republic of']
['CHN', 'DOM', 'ATG', 'FJI', 'AUS', 'BGD', 'BLZ', 'BRB', 'COK', 'CAN', 'BHS', 'GTM', 'JAM
↪', 'LCA', 'MDG', 'MEX', 'PRK', 'SLV', 'MMR', 'PYF', 'SLB', 'TWN', 'IND', 'USA', 'HND',
↪'HTI', 'PAK', 'PHL', 'HKG', 'KOR', 'NIC', 'OMN', 'JPN', 'PRI', 'THA', 'MTQ', 'PNG',
↪'TON', 'VEN', 'VNM', 'VCT', 'VUT', 'DMA', 'CUB', 'COM', 'MOZ', 'MWI', 'WSM', 'ZAF',
↪'LKA', 'PLW', 'WLF', 'SOM', 'SYC', 'REU', 'KIR', 'CPV', 'FSM', 'PAN', 'CRI', 'YEM',
↪'TUV', 'MNP', 'COL', 'AIA', 'DJI', 'KHM', 'MAC', 'IDN', 'GLP', 'TCA', 'KNA', 'LAO',
↪'MUS', 'MHL', 'PRT', 'VIR', 'ZWE', 'BLM', 'VGB', 'MAF', 'SXM', 'TZA']
```

emdat_to_impact()

function to load EM-DAT impact data and return impact set with impact per event

Parameters:

- emdat_file_csv (str): Full path to EMDAT-file (CSV)
- hazard_type_climada (str): Hazard type abbreviation used in CLIMADA, e.g. 'TC'

Optional parameters:

- hazard_type_emdat (list or str): List of Disaster (sub-)type accordung EMDAT terminology or CLIMADA hazard type abbreviations. e.g. ['Wildfire', 'Forest fire'] or ['BF']
- year_range (list with 2 integers): start and end year e.g. [1980, 2017]
- countries (list of str): country ISO3-codes or names, e.g. ['JAM', 'CUB']. Set to None or ['all'] for all countries
- reference_year (int): reference year of exposures for normalization. Impact is scaled proportional to GDP to the value of the reference year. No scaling for reference_year=0 (default)
- imp_str (str): Column name of impact metric in EMDAT CSV, e.g. 'Total Affected'; default = "Total Damages"

Returns:

- impact_instance (instance of climada.engine.Impact): Impact() instance (same format as output from CLIMADA impact computations). Values are scaled with GDP to reference_year if reference_year not equal 0. impact_instance.eai_exp holds expected annual impact for each country. impact_instance.coord_exp holds rough central coordinates for each country.
- countries (list): ISO3-codes of countries imn same order as in impact_instance.eai_exp

[3]:
```
"""Global TC damages 2000 to 2009"""

impact_emdat, countries = emdat_to_impact(emdat_file_path, 'TC', year_range=(2000,2009))

print('Number of TC events in EM-DAT 2000 to 2009 globally: %i' %(impact_emdat.event_id.
↪size))
print('Global annual average monetary damage (AAI) from TCs as reported in EM-DAT 2000␣
↪to 2009: USD billion %2.2f' \
      %(impact_emdat.aai_agg/1e9))
```

```
2021-10-19 16:44:58,210 - climada.engine.impact_data - WARNING - ISO3alpha code not␣
↪found in iso_country: SPI
2021-10-19 16:44:59,007 - climada.engine.impact_data - WARNING - Country not found in␣
↪iso_country: SPI
Number of TC events in EM-DAT 2000 to 2009 globally: 533
Global annual average monetary damage (AAI) from TCs as reported in EM-DAT 2000 to 2009:␣
↪USD billion 38.07
```

[31]:
```
"""Total people affected by TCs in the Philippines in 2013:"""

# People affected
impact_emdat_PHL, countries = emdat_to_impact(emdat_file_path, 'TC', countries='PHL', \
                                    year_range=(2013,2013), imp_str="Total Affected")

print('Number of TC events in EM-DAT in the Philipppines, 2013: %i' \
      %(impact_emdat_PHL.event_id.size))
print('\nPeople affected by TC events in the Philippines in 2013 (per event):')
print(impact_emdat_PHL.at_event)
print('\nPeople affected by TC events in the Philippines in 2013 (total):')
print(int(impact_emdat_PHL.aai_agg))

# Comparison to monetary damages:
impact_emdat_PHL_USD, _ = emdat_to_impact(emdat_file_path, 'TC', countries='PHL', \
                                    year_range=(2013,2013))

ax = plt.scatter(impact_emdat_PHL_USD.at_event, impact_emdat_PHL.at_event)
plt.title('Typhoon impacts in the Philippines, 2013')
plt.xlabel('Total Damage [USD]')
plt.ylabel('People Affected')
#plt.xscale('log')
#plt.yscale('log')
```

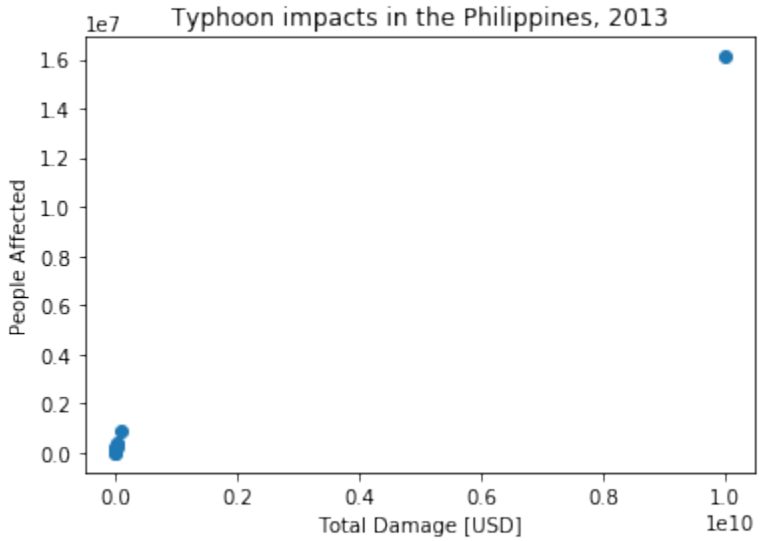
```
Number of TC events in EM-DAT in the Philipppines, 2013: 8
```

(continues on next page)

(continued from previous page)

```
People affected by TC events in the Philippines in 2013 (per event):
[7.269600e+04 1.059700e+04 8.717550e+05 2.204430e+05 1.610687e+07
 3.596000e+03 3.957300e+05 2.628840e+05]

People affected by TC events in the Philippines in 2013 (total):
17944571
```

[31]:
```
Text(0, 0.5, 'People Affected')
```



emdat_impact_yearlysum()

function to load EM-DAT impact data and return DataFrame with impact summed per year and country

Parameters:

- emdat_file_csv (str): Full path to EMDAT-file (CSV)

Optional parameters:

- hazard (list or str): List of Disaster (sub-)type accordung EMDAT terminology or CLIMADA hazard type abbreviations. e.g. [‘Wildfire’, ‘Forest fire’] or [‘BF’]
- year_range (list with 2 integers): start and end year e.g. [1980, 2017]
- countries (list of str): country ISO3-codes or names, e.g. [‘JAM’, ‘CUB’]. Set to None or [‘all’] for all countries
- reference_year (int): reference year of exposures for normalization. Impact is scaled proportional to GDP to the value of the reference year. No scaling for reference_year=0 (default)
- imp_str (str): Column name of impact metric in EMDAT CSV, e.g. ‘Total Affected’; default = “Total Damages”
- version (int): given EM-DAT data format version (i.e. year of download), changes naming of columns/variables (default: 2020)

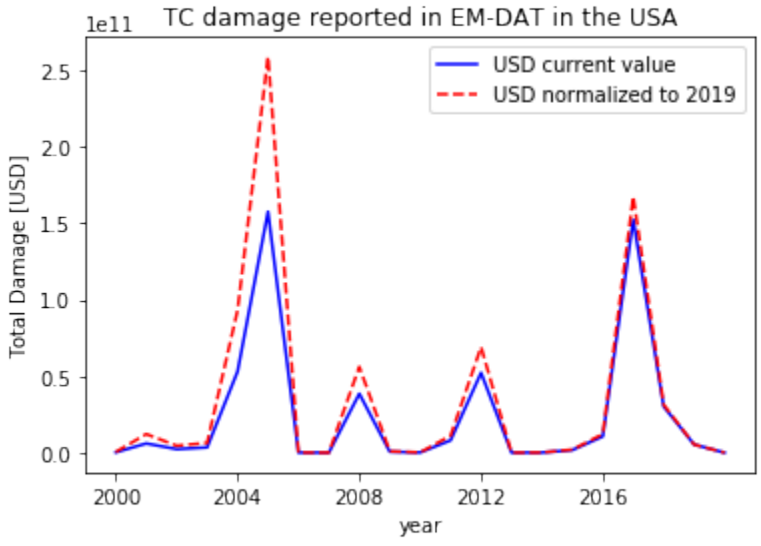
Returns:

- pandas.DataFrame with impact per year and country

[5]:
```
"""Yearly TC damages in the USA, normalized and current"""

yearly_damage_normalized_to_2019 = emdat_impact_yearlysum(emdat_file_path, countries='USA
↪', \
                                                          hazard='Tropical cyclone', year_
↪range=None, \
                                                          reference_year=2019)

yearly_damage_current = emdat_impact_yearlysum(emdat_file_path, countries=['USA'],␣
↪hazard='TC',)

import matplotlib.pyplot as plt

fig, axis = plt.subplots(1, 1)
axis.plot(yearly_damage_current.year, yearly_damage_current.impact, 'b', label='USD␣
↪current value')
axis.plot(yearly_damage_normalized_to_2019.year, yearly_damage_normalized_to_2019.impact_
↪scaled, \
          'r--', label='USD normalized to 2019')
plt.legend()
axis.set_title('TC damage reported in EM-DAT in the USA')
axis.set_xticks([2000, 2004, 2008, 2012, 2016])
axis.set_xlabel('year')
axis.set_ylabel('Total Damage [USD]')
```

```
[2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2014
 2015 2016 2017 2018 2019 2020]
```

[5]:
```
Text(0, 0.5, 'Total Damage [USD]')
```



5.13 Unsequa - a module for uncertainty and sensitivity analysis

This is a tutorial for the unsequa module in CLIMADA. A detailled description can be found in **`Kropf (2021) <>`__**.

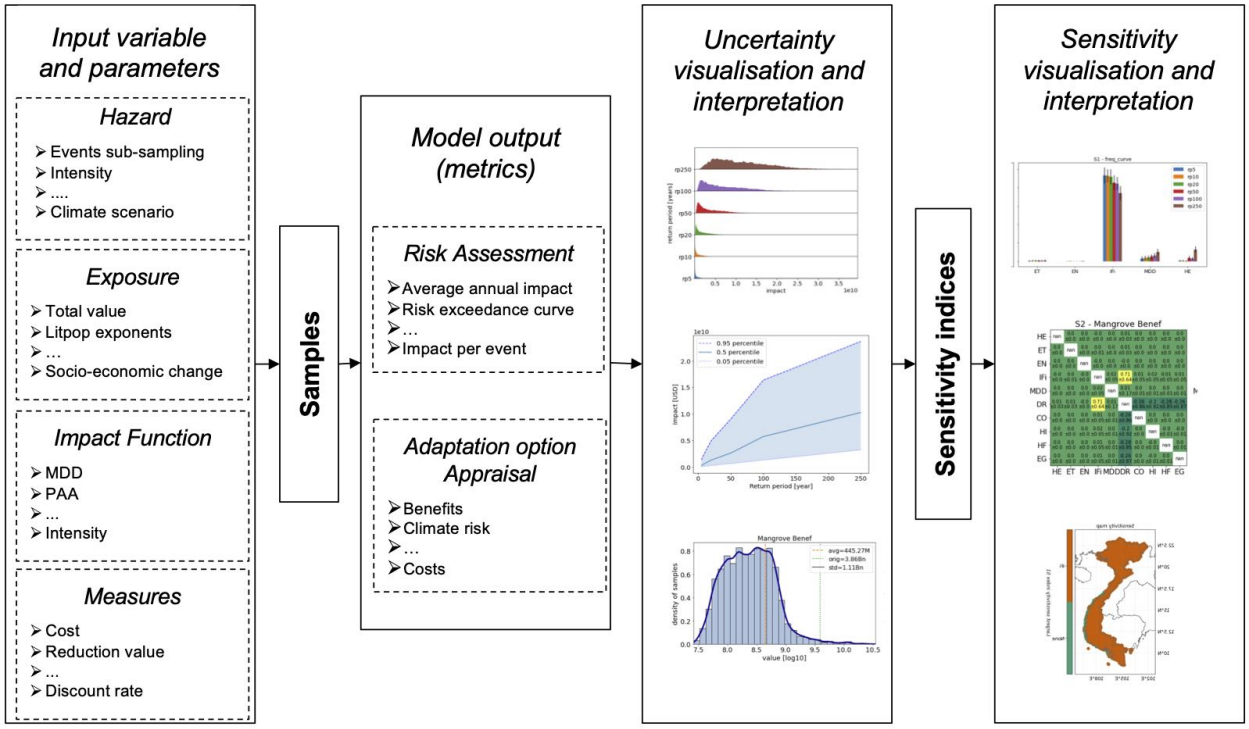
Table of Contents

5.13.1 Uncertainty and sensitivity analysis?

Before doing an uncertainty quantification in CLIMADA, it is imperative that you get first comfortable with the different notions of uncertainty in the modelling world (see e.g.Pianosi (2016) or Douglas-Smith(2020) for a review). In particular, note that the uncertaintity values will only be as good as the input from the user. In addition, not all uncertainties can be numerically quantified, and even worse, some unkonwns are unknown. This means that sometimes, quantifying uncertainty can lead to false confidence in the output!. For a more philosophical discussion about the types of uncertainties in climate research see Knüsel (2020) and **`Otth (2022) <>`__**.

In this module, it is possible to perform global uncertainty analysis, as well as a sensitivity analysis. The word global is meant as opposition to the 'one-factor-at-a-time' (OAT) strategy. The OAT strategy, which consists in analyzing the effect of varying one model input factor at a time while keeping all other fixed, is popular among modellers, but has major shortcomings Saltelli (2010), Saltelli(2019) and should not be used.

A rough schemata of how to perform uncertainty and sensitivity analysis (taken from **`Kropf(2021) <>`__**)



1. **`Kropf, C.M. et al. Uncertainty and sensitivity analysis for global probabilistic weather and climate risk modelling: an implementation in the CLIMADA platform (2021) <>`__**

2. Pianosi, F. et al. Sensitivity analysis of environmental models: A systematic review with practical workflow. Environmental Modelling & Software 79, 214–232 (2016). 3.Douglas-Smith, D., Iwanaga, T., Croke, B. F. W. & Jakeman, A. J. Certain trends in uncertainty and sensitivity analysis: An overview of software tools and techniques. Environmental Modelling & Software 124, 104588 (2020)

3. Knüsel, B. Epistemological Issues in Data-Driven Modeling in Climate Research. (ETH Zurich, 2020)

4. Saltelli, A. et al. Why so many published sensitivity analyses are false: A systematic review of sensitivity analysis practices. Environmental Modelling & Software 114, 29–39 (2019)

5. Saltelli, A. & Annoni, P. How to avoid a perfunctory sensitivity analysis. Environmental Modelling & Software 25, 1508–1517 (2010)

5.13.2 Unsequa Module Structure

The unsequa module contains several key classes.

The model input parameters and their distribution are specified as - `InputVar`: defines input uncertainty variables

The input parameter sampling, Monte-Carlo uncertainty distribution calculation and the sensitivity index computation are done in - `CalcImpact`: compute uncertainties for outputs of `climada.engine.impact.calc` (child class of `Calc`) - `CalcCostBenefit`: compute uncertainties for outputs of `climada.engine.cost_benefit.calc` (child class of `Calc`)

The results are stored in - `UncOutput`: store the uncertainty and sensitivity analysis results. Contains also several plotting methods. This is a class which only stores data. - `UncImpactOutput`: subclass with dataframes specifically for `climada.engine.impact.calc` uncertainty and sensitivity analysis results. - `UncCostBenefitOutput`: subclass with dataframes specifically for `climada.engine.cost_benefit.calc` uncertainty and sensitivity analysis results.

5.13.3 InputVar

The InputVar class is used to define uncertainty variables.

Attribute	Type	Description
func	function	Model variable defined as a function of the uncertainty input parameters
distr_dict	dict	Dictionary of the probability density distributions of the uncertainty input parameters

An **input uncertainty parameter** is a numerical input value that has a certain probability density distribution in your model, such as the total exposure asset value, the slope of the vulnerability function, the exponents of the litpop exposure, the value of the discount rate, the cost of an adaptation measure, ...

The probability densitity distributions (values of `distr_dict`) of the input uncertainty parameters (keyword arguments of the `func` and keys of the `distr_dict`) can be any of the ones defined in scipy.stats.

Several helper methods exist to make generic InputVar for Exposures, ImpactFuncSet, Hazard, Entity (including DiscRates and Measures). These are described in details in the tutorial climada_engine_uncertainty_helper. These are a good bases for your own computations.

Example - custom continuous uncertainty parameter

Suppose we assume that the GDP value used to scale the exposure has a relative error of +-10%.

[1]:
```
import warnings
warnings.filterwarnings('ignore') #Ignore warnings for making the tutorial's pdf.

#Define the base exposure
from climada.util.constants import EXP_DEMO_H5
from climada.entity import Exposures
exp_base = Exposures.from_hdf5(EXP_DEMO_H5)
```

```
2022-01-10 21:09:45,445 - climada.entity.exposures.base - INFO - Reading /Users/ckropf/
↪climada/demo/data/exp_demo_today.h5
```

[2]:
```
# Define the function that returns an exposure with scaled total assed value
# Here x_exp is the input uncertainty parameter and exp_func the inputvar.func.
def exp_func(x_exp, exp_base=exp_base):
    exp = exp_base.copy()
    exp.gdf.value *= x_exp
    return exp
```

[3]:
```
# Define the Uncertainty Variable with +-10% total asset value
# The probability density distribution of the input uncertainty parameter x_exp is sp.
↪stats.uniform(0.9, 0.2)
from climada.engine.unsequa import InputVar
import scipy as sp

exp_distr = {"x_exp": sp.stats.uniform(0.9, 0.2),
             }
exp_iv = InputVar(exp_func, exp_distr)
```

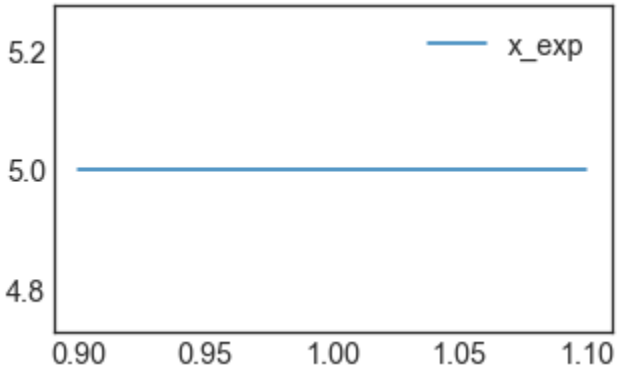
[4]:
```
# Uncertainty parameters
exp_iv.labels
```

[4]:
```
['x_exp']
```

[5]:
```
# Evaluate for a given value of the uncertainty parameters
exp095 = exp_iv.func(x_exp = 0.95)
print(f"Base value is {exp_base.gdf['value'].sum()}, and the value for x_exp=0.95 is
↪{exp095.gdf['value'].sum()}")
```

```
Base value is 657053294559.9105, and the value for x_exp=0.95 is 624200629831.9148
```

[6]:
```
# Defined distribution
exp_iv.plot(figsize=(5, 3));
```



Example - custom categorical uncertainty parameter

Suppose we want to test different exponents (m=1,2 ; n=1,2) for the LitPop exposure for the country Switzerland.

[7]:
```
from climada.entity import LitPop

m_min, m_max = (1, 2)
n_min, n_max = (1, 2)

# Define the function
# Note that this here works, but might be slow because the method LitPop is called␣
↪everytime the the function
# is evaluated, and LitPop is relatively slow.
def litpop_cat(m, n):
    exp = Litpop.from_countries('CHE', res_arcsec=150, exponent=[m, n])
    return exp
```

[9]:
```
# A faster method would be to first create a dictionnary with all the exposures. This␣
↪however
# requires more memory and precomputation time (here ~3-4mins)
exp = LitPop()
litpop_dict = {}
for m in range(m_min, m_max + 1):
    for n in range(n_min, n_max + 1):
        exp_mn = LitPop.from_countries('CHE', res_arcsec=150, exponents=[m, n]);
        litpop_dict[(m, n)] = exp_mn
```

(continues on next page)

(continued from previous page)

```
def litpop_cat(m, n, litpop_dict=litpop_dict):
    return litpop_dict[(m, n)]
```

```
2022-02-11 16:29:34,770 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

```
53.1kKB [00:25, 2.07kKB/s]
```

```
2022-02-11 16:30:05,073 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2022-02-11 16:30:07,529 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
2022-02-11 16:30:10,199 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2018. Using nearest available year for GPW data: 2020
```

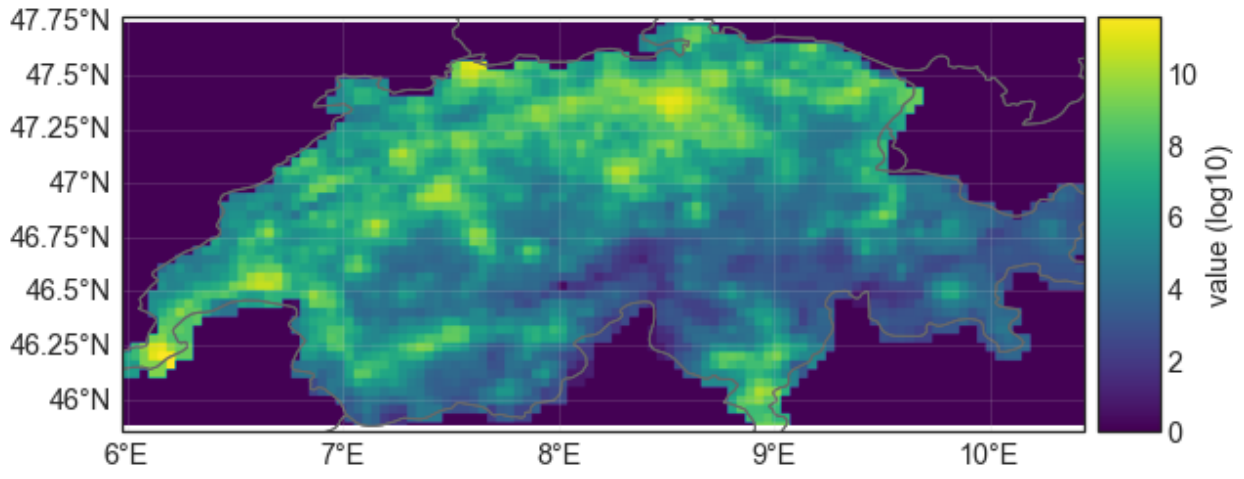
[10]:
```
#Define the distribution dictionnary
import scipy as sp
from climada.engine.unsequa import InputVar

distr_dict = {
    'm': sp.stats.randint(low=m_min, high=m_max+1),
    'n': sp.stats.randint(low=n_min, high=n_max+1)
    }

cat_iv = InputVar(litpop_cat, distr_dict) # One can use either of the above definitions␣
↪of litpop_cat
```

[11]:
```
# Uncertainty parameters
cat_iv.labels
```

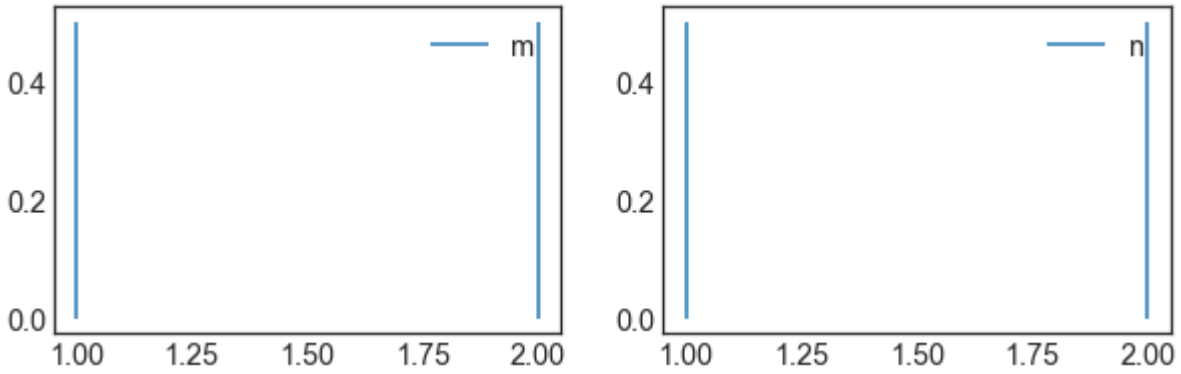
[11]:
```
['m', 'n']
```

[12]:
```
cat_iv.evaluate(m=1, n=2).plot_raster();
```

```
2022-01-10 21:09:52,830 - climada.util.coordinates - INFO - Raster from resolution 0.
↪04166666000000063 to 0.04166666000000063.
```



[13]:
```
cat_iv.plot(figsize=(10, 3));
```



5.13.4 UncOutput

The `UncOutput` class is used to store data from sampling, uncertainty and sensitivity analysis. An UncOutput object can be saved and loaded from .hdf5. The classes `UncImpactOuput` and `UncCostBenefitOutput` are extensions of `UncOutput` specific for `CalcImpact` and `CalcCostBenefit`, respectively.

Data attributes

Attribute	Type	Description
samples_df	pandas.dataframe	Each row represents a sample obtained from the input parameters (one per column) distributions
UncImpactOutput		
aai_agg_unc_df	pandas.dataframe	Uncertainty data for `aai_agg`
tot_value_unc_df	pandas.dataframe	Uncertainty data for `tot_value`.
freq_curve_unc_df	pandas.dataframe	Uncertainty data for `freq_curve`. One return period per column.
eai_exp_unc_df	pandas.dataframe	Uncertainty data for `eai_exp`. One exposure point per column.
at_event_unc_df	pandas.dataframe	Uncertainty data for `at_event`. One event per column.
UncCostBenefitOutput		
imp_meas_present_unc_df	pandas.dataframe	Uncertainty data for `imp_meas_present`. One measure per column.
imp_meas_future_unc_df	pandas.dataframe	Uncertainty data for `imp_meas_present`. One measure per column
tot_climate_risk_unc_df	pandas.dataframe	Uncertainty data for `tot_climate_risk`. One measure per column.
benefit_unc_df	pandas.dataframe	Uncertainty data for `benefit`. One measure per column.
cost_ben_ratio_unc_df	pandas.dataframe	Uncertainty data for `cost_ben_ratio`. One measure per column.
cost_benefit_kwargs	dictionary	Keyword arguments for `climada.engine.cost_benefit.calc`.

Metadata and input data attributes

These attributes are used for book-keeping and characterize the sample, uncertainty and sensitivity data. These attributes are set by the methods from classes `CalcImpact` and `CalcCostBenefit` used to generate sample, uncertainty and sensitivity data.

Attribute	Type	Description
sampling_method	str	The sampling method as defined in SALib. Possible choices: 'saltelli', 'fast_sampler', 'latin', 'morris', 'dgsm', 'ff'
sampling_kwargs	dict	Keyword arguments for the sampling_method.
n_samples	int	Effective number of samples (number of rows of samples_df)
param_labels	list(str)	Name of all the uncertainty input parameters
problem_sa	dict	The description of the uncertainty variables and their distribution as used in SALib.
sensitivity_method	str	Sensitivity analysis method from SALib.analyse Possible choices: 'fast', 'rbd_fact', 'morris', 'sobol', 'delta', 'ff'. Note that in Salib, sampling methods and sensitivity analysis methods should be used in specific pairs.
sensitivity_kwargs	dict	Keyword arguments for sensitivity_method.
unit	str	Unit of the exposures value

Example from file

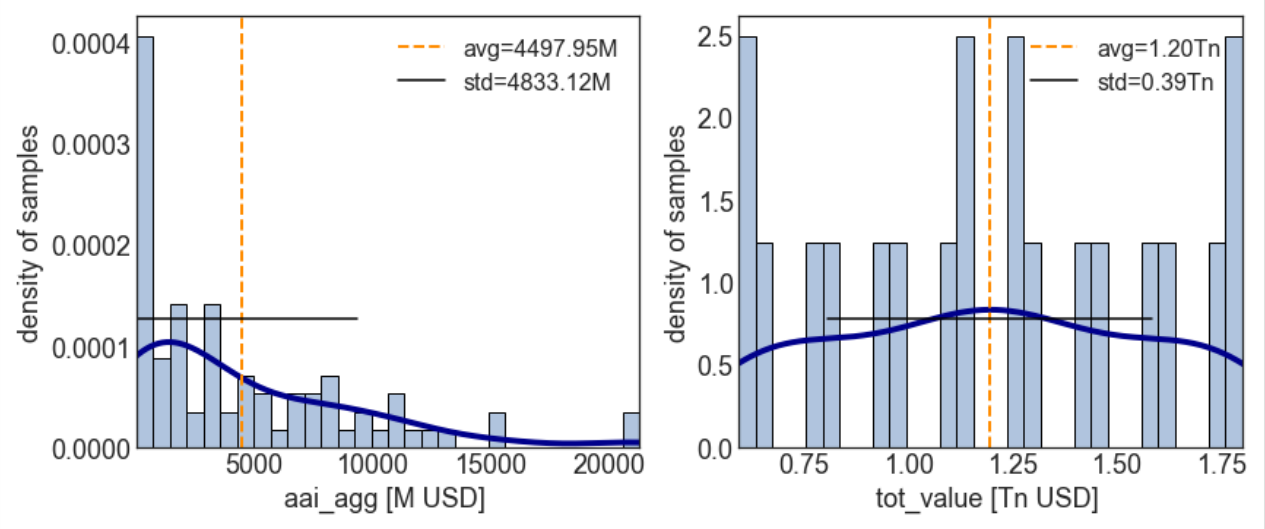
Here we show an example loaded from file. In the sections below this class is extensively used and further examples can be found.

[17]:
```
# Download the test file from the API
# Requires internet connection
from climada.util.constants import TEST_UNC_OUTPUT_IMPACT
from climada.util.api_client import Client
apiclient = Client()
ds = apiclient.get_dataset_info(name=TEST_UNC_OUTPUT_IMPACT, status='test_dataset')
_target_dir, [filename] = apiclient.download_dataset(ds)
```

[18]:
```
# If you produced your own data, you do not need the API. Just replace 'filename' with␣
↪the path to your file.
from climada.engine.unsequa import UncOutput
unc_imp = UncOutput.from_hdf5(filename)
```

```
2022-01-10 21:09:57,236 - climada.engine.unsequa.unc_output - INFO - Reading /Users/
↪ckropf/climada/data/unc_output/unc_output_impact/test_unc_output_impact/v1/test_unc_
↪output_impact.hdf5
```

[19]:
```
unc_imp.plot_uncertainty(metric_list=['aai_agg', 'tot_value'], figsize=(12,5));
```



[21]:
```
# Download the test file from the API
# Requires internet connection
from climada.util.constants import TEST_UNC_OUTPUT_COSTBEN
from climada.util.api_client import Client
apiclient = Client()
ds = apiclient.get_dataset_info(name=TEST_UNC_OUTPUT_COSTBEN, status='test_dataset')
_target_dir, [filename] = apiclient.download_dataset(ds)
```

[22]:
```
# If you produced your own data, you do not need the API. Just replace 'filename' with␣
↪the path to your file.
from climada.engine.unsequa import UncOutput
unc_cb = UncOutput.from_hdf5(filename)
```

```
2022-01-10 21:09:57,901 - climada.engine.unsequa.unc_output - INFO - Reading /Users/
↪ckropf/climada/data/unc_output/unc_output_costben/test_unc_output_costben/v1/test_unc_
↪output_costben.hdf5
```

[23]:
```
unc_cb.get_uncertainty().tail()
```

[23]:

	Mangroves Benef	Beach nourishment Benef	Seawall Benef	\
35	2.375510e+08	1.932608e+08	234557.682554	
36	9.272772e+07	7.643803e+07	9554.257314	
37	1.464219e+08	1.179927e+08	192531.748810	
38	9.376369e+07	7.722882e+07	10681.112247	
39	9.376369e+07	7.722882e+07	10681.112247	

	Building code Benef	Mangroves CostBen	Beach nourishment CostBen	\
35	1.584398e+08	6.347120	10.277239	
36	5.501366e+07	16.260133	25.984286	
37	8.979471e+07	10.297402	16.833137	
38	5.555413e+07	12.965484	20.736269	
39	5.555413e+07	16.080478	25.718218	

	Seawall CostBen	Building code CostBen	no measure - risk - future	\
35	4.350910e+04	66.742129	6.337592e+08	
36	1.068151e+06	192.217876	2.200547e+08	

(continues on next page)

(continued from previous page)

```
37     5.300629e+04             117.764285                3.591788e+08
38     7.703765e+05             153.475031                2.222165e+08
39     9.554617e+05             190.347852                2.222165e+08

    no measure - risk_transf - future  ...  \
35                                0.0  ...
36                                0.0  ...
37                                0.0  ...
38                                0.0  ...
39                                0.0  ...

    Beach nourishment - cost_ins - future  Seawall - risk - future  \
35                                      1             6.335246e+08
36                                      1             2.200451e+08
37                                      1             3.589863e+08
38                                      1             2.222058e+08
39                                      1             2.222058e+08

    Seawall - risk_transf - future  Seawall - cost_meas - future  \
35                               0                  1.020539e+10
36                               0                  1.020539e+10
37                               0                  1.020539e+10
38                               0                  8.228478e+09
39                               0                  1.020539e+10

    Seawall - cost_ins - future  Building code - risk - future  \
35                            1                   4.753194e+08
36                            1                   1.650410e+08
37                            1                   2.693841e+08
38                            1                   1.666624e+08
39                            1                   1.666624e+08

    Building code - risk_transf - future  Building code - cost_meas - future  \
35                                    0                         1.057461e+10
36                                    0                         1.057461e+10
37                                    0                         1.057461e+10
38                                    0                         8.526172e+09
39                                    0                         1.057461e+10

    Building code - cost_ins - future  tot_climate_risk
35                                  1      6.337592e+08
36                                  1      2.200547e+08
37                                  1      3.591788e+08
38                                  1      2.222165e+08
39                                  1      2.222165e+08

[5 rows x 29 columns]
```

5.13.5 CalcImpact

Set the InputVars

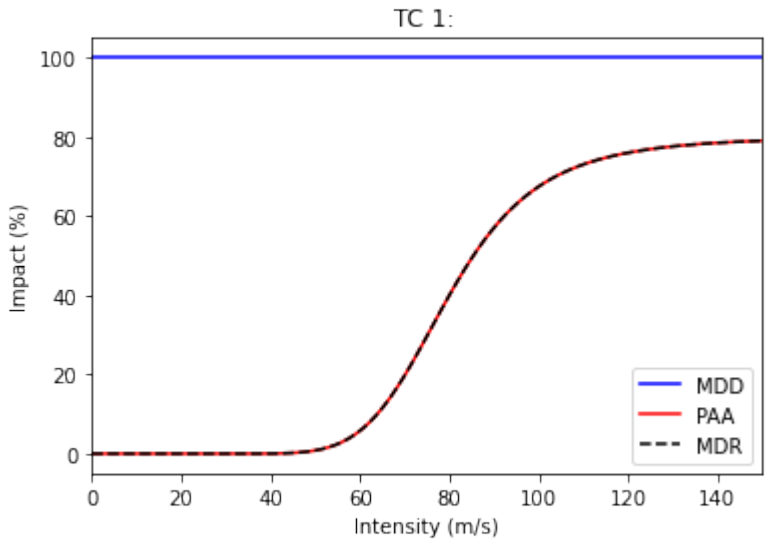
In this example, we model the impact function for tropical cyclones on the parametric function suggested in Emanuel (2015) with 4 parameters. The exposures total value varies between 80% and 120%. For that hazard, we assume to have no good error estimate and thus do not define an InputVar for the hazard.

[24]:
```
#Define the input variable functions
import numpy as np

from climada.entity import ImpactFunc, ImpactFuncSet, Exposures
from climada.util.constants import EXP_DEMO_H5, HAZ_DEMO_H5
from climada.hazard import Hazard

def impf_func(G=1, v_half=84.7, vmin=25.7, k=3, _id=1):

    def xhi(v, v_half, vmin):
        return max([(v - vmin), 0]) / (v_half - vmin)

    def sigmoid_func(v, G, v_half, vmin, k):
        return G * xhi(v, v_half, vmin)**k / (1 + xhi(v, v_half, vmin)**k)

    #In-function imports needed only for parallel computing on Windows
    import numpy as np
    from climada.entity import ImpactFunc, ImpactFuncSet
    imp_fun = ImpactFunc()
    imp_fun.haz_type = 'TC'
    imp_fun.id = _id
    imp_fun.intensity_unit = 'm/s'
    imp_fun.intensity = np.linspace(0, 150, num=100)
    imp_fun.mdd = np.repeat(1, len(imp_fun.intensity))
    imp_fun.paa = np.array([sigmoid_func(v, G, v_half, vmin, k) for v in imp_fun.
↪intensity])
    imp_fun.check()
    impf_set = ImpactFuncSet()
    impf_set.append(imp_fun)
    return impf_set



haz = Hazard.from_hdf5(HAZ_DEMO_H5)
exp_base = Exposures.from_hdf5(EXP_DEMO_H5)
#It is a good idea to assign the centroids to the base exposures in order to avoid␣
↪repeating this
# potentially costly operation for each sample.
exp_base.assign_centroids(haz)
def exp_base_func(x_exp, exp_base):
    exp = exp_base.copy()
    exp.gdf.value *= x_exp
    return exp
from functools import partial
exp_func = partial(exp_base_func, exp_base=exp_base)
```

```
2022-01-10 21:12:30,850 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
2022-01-10 21:12:30,988 - climada.entity.exposures.base - INFO - Reading /Users/ckropf/
↪climada/demo/data/exp_demo_today.h5
2022-01-10 21:12:31,068 - climada.entity.exposures.base - INFO - Matching 50 exposures␣
↪with 2500 centroids.
2022-01-10 21:12:31,070 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
```

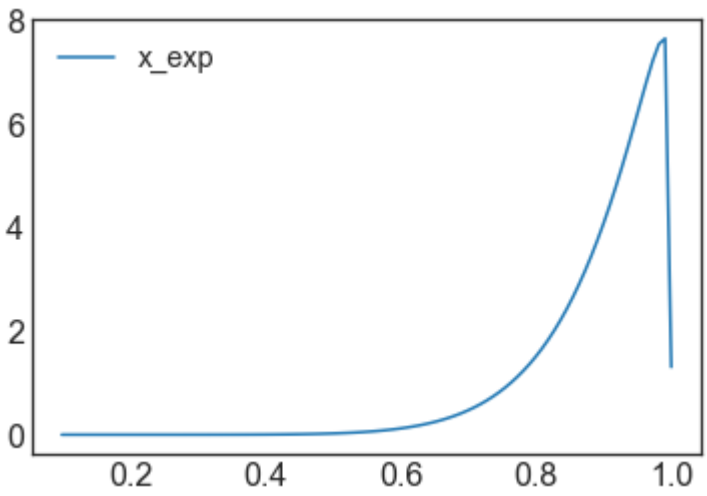
[25]:
```
# Visualization of the parametrized impact function
impf_func(G=0.8, v_half=80, vmin=30,k=5).plot();
```

```
2022-01-10 21:12:31,081 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
```



[26]:
```
#Define the InputVars

import scipy as sp
from climada.engine.unsequa import InputVar

exp_distr = {"x_exp": sp.stats.beta(10, 1.1)} #This is not really a reasonable␣
↪distribution but is used
                                              #here to show that you can use any scipy␣
↪distribution.

exp_iv = InputVar(exp_func, exp_distr)

impf_distr = {
    "G": sp.stats.truncnorm(0.5, 1.5),
    "v_half": sp.stats.uniform(35, 65),
    "vmin": sp.stats.uniform(0, 15),
    "k": sp.stats.uniform(1, 4)
    }
impf_iv = InputVar(impf_func, impf_distr)
```

[27]:
```
import matplotlib.pyplot as plt
ax = exp_iv.plot(figsize=(6,4));
plt.yticks(fontsize=16);
plt.xticks(fontsize=16);
```



Compute uncertainty and sensitivity using default methods

First, we define the UncImpact object with our uncertainty variables.

[28]:
```
from climada.engine.unsequa import CalcImpact

calc_imp = CalcImpact(exp_iv, impf_iv, haz)
```

Next, we generate samples for the uncertainty parameters using the default methods. Note that depending on the chosen Salib method, the effective number of samples differs from the input variable N. For the default 'saltelli', with `calc_second_order=True`, the effective number is N(2D+2), with D the number of uncertainty parameters. See SAlib for more information.

[29]:
```
output_imp = calc_imp.make_sample(N=2**7, sampling_kwargs={'skip_values': 2**8})
output_imp.get_samples_df().tail()
```

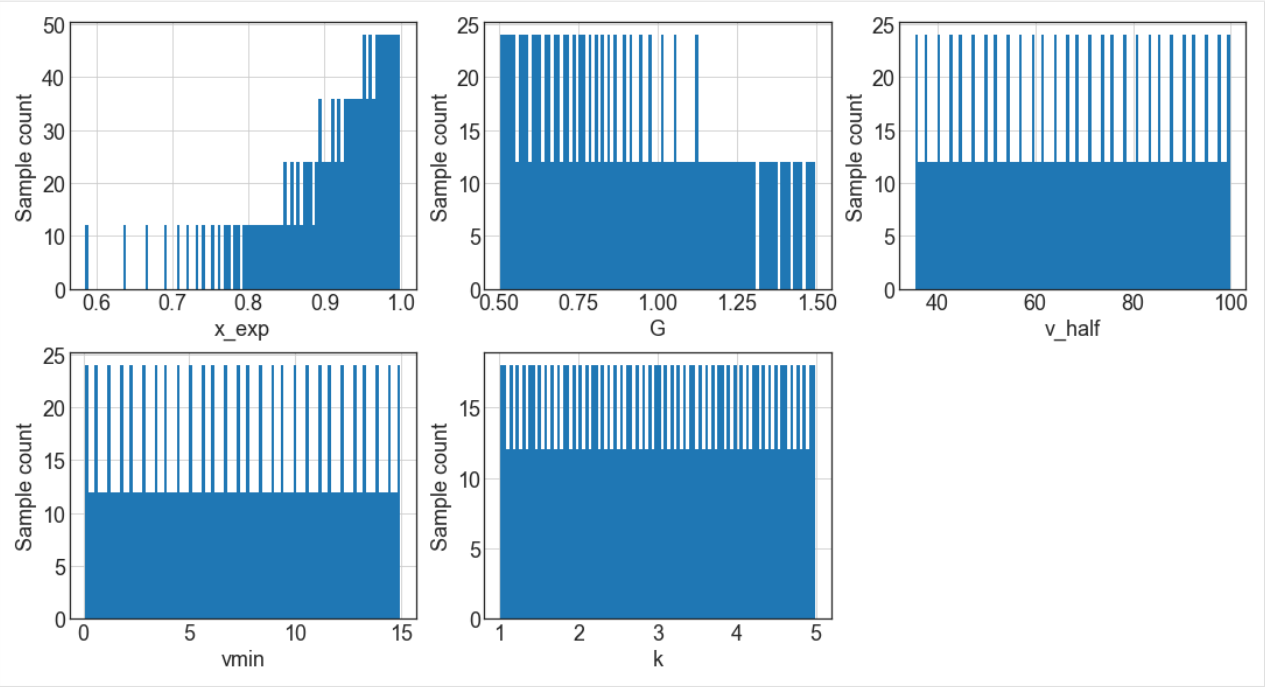
```
2022-01-10 21:09:59,022 - climada.engine.unsequa.calc_base - INFO - Effective number of␣
↪made samples: 1536
```

[29]:

	x_exp	G	v_half	vmin	k
1531	0.876684	1.242977	53.662109	2.080078	4.539062
1532	0.876684	0.790617	44.013672	2.080078	4.539062
1533	0.876684	0.790617	53.662109	13.681641	4.539062
1534	0.876684	0.790617	53.662109	2.080078	3.960938
1535	0.876684	0.790617	53.662109	2.080078	4.539062

The resulting samples can be visualized in plots.

[30]:
```
output_imp.plot_sample(figsize=(15,8));
```



Now we can compute the value of the impact metrics for all the samples. In this example, we additionaly chose to restrict the return periods 50, 100, and 250 years. By default, `eai_exp` and `at_event` are not stored.

[31]:
```
output_imp = calc_imp.uncertainty(output_imp, rp = [50, 100, 250])
```

```
2022-01-10 21:10:00,114 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
2022-01-10 21:10:00,116 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2022-01-10 21:10:00,117 - climada.engine.impact - INFO - Calculating damage for 50␣
↪assets (>0) and 216 events.
2022-01-10 21:10:00,122 - climada.engine.unsequa.calc_base - INFO -

Estimated computaion time: 0:00:16.896000
```

The distributions of metrics ouputs are stored as dictionaries of pandas dataframe. The metrics are directly taken from the output of `climada.impact.calc`. For each metric, on dataframe is made.

[32]:
```
#All the computed uncertainty metrics attribute
output_imp.uncertainty_metrics
```

[32]:
```
['aai_agg', 'freq_curve', 'tot_value']
```

[33]:
```
#One uncertainty dataframe
output_imp.get_unc_df('aai_agg').tail()
```

[33]:
```
          aai_agg
1531  2.905571e+09
1532  3.755172e+09
1533  1.063119e+09
```

(continues on next page)

(continued from previous page)

```
1534  2.248718e+09
1535  1.848139e+09
```

Accessing the uncertainty is in general done via the method `get_uncertainty()`. If none are specified, all metrics are returned.

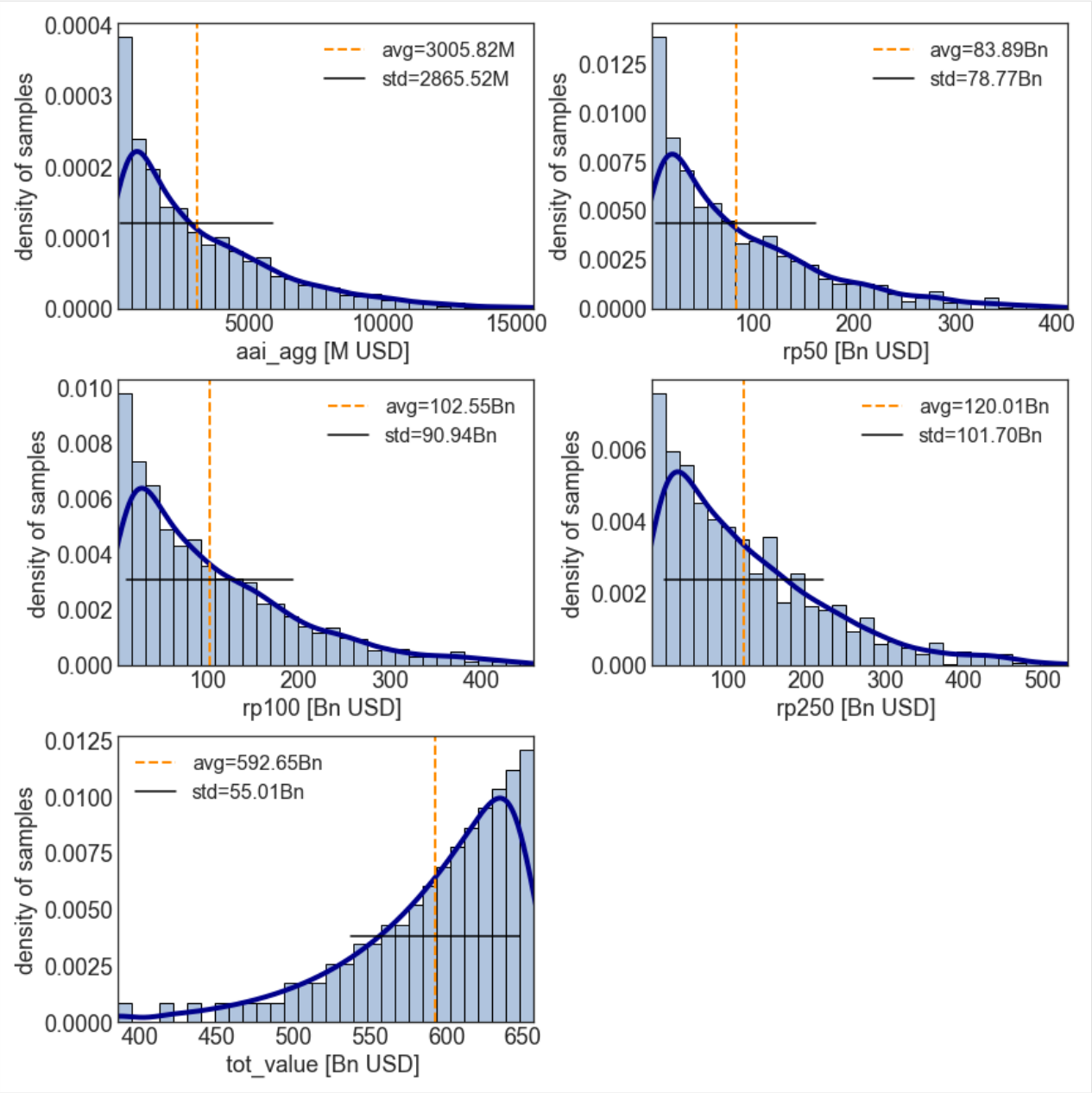
[34]:
```
output_imp.get_uncertainty().tail()
```

[34]:

	aai_agg	rp50	rp100	rp250	tot_value
1531	2.905571e+09	8.324391e+10	1.162643e+11	1.510689e+11	5.760281e+11
1532	3.755172e+09	1.096005e+11	1.460838e+11	1.809413e+11	5.760281e+11
1533	1.063119e+09	2.892734e+10	4.720869e+10	6.807561e+10	5.760281e+11
1534	2.248718e+09	6.468855e+10	8.653474e+10	1.085266e+11	5.760281e+11
1535	1.848139e+09	5.294874e+10	7.395191e+10	9.609003e+10	5.760281e+11

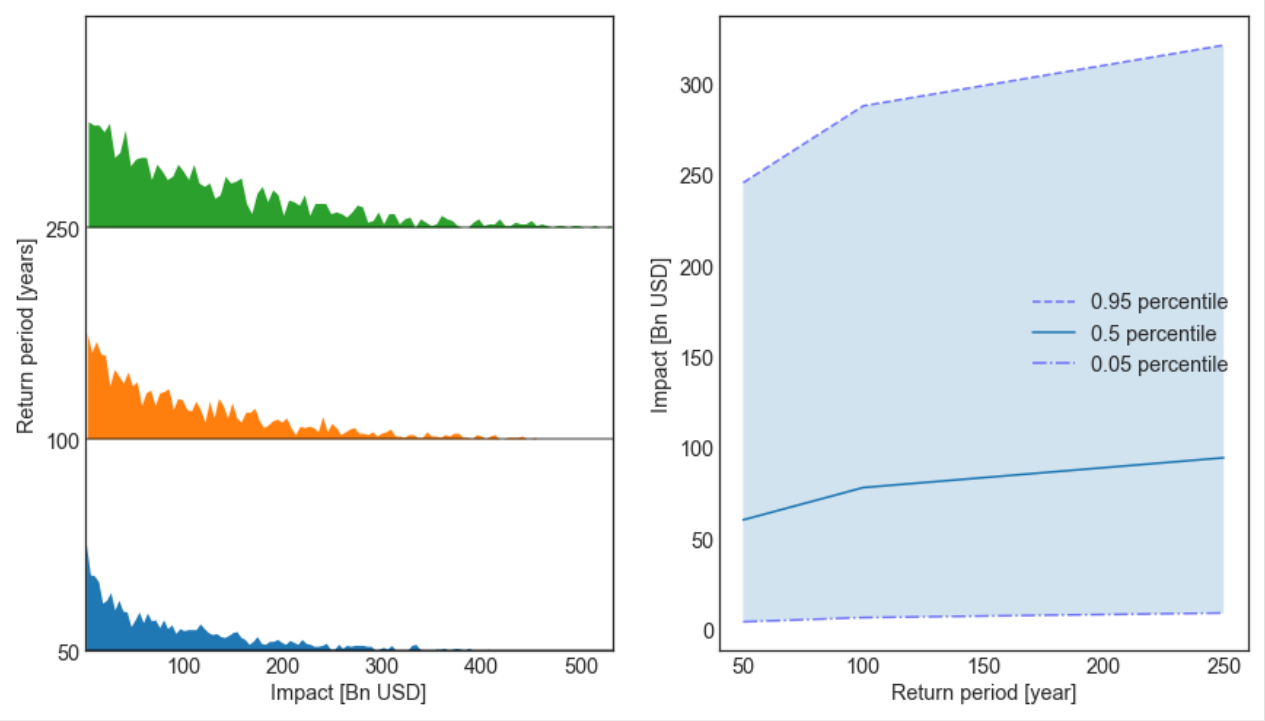
The distributions of the one-dimensioanl metrics (`eai_exp` and `at_event` are never shown with this method) can be vizualised with plots.

[35]:
```
output_imp.plot_uncertainty(figsize=(12,12));
```



[36]:
```
# Specific plot for the return period distributions
output_imp.plot_rp_uncertainty(figsize=(14.3,8));
```

```
No handles with labels found to put in legend.
```



Now that a distribution of the impact metrics has been computed for each sample, we can also compute the sensitivity indices for each metrics to each uncertainty parameter. Note that the chosen method for the sensitivity analysis should correpond to its sampling partner as defined in the SAlib package.

The sensitivity indices dictionnaries outputs from the SAlib methods are stored in the same structure of nested dictionnaries as the metrics distributions. Note that depending on the chosen sensitivity analysis method the returned indices dictionnary will return specific types of sensitivity indices with specific names. Please get familiar with SAlib for more information.

Note that in our case, several of the second order sensitivity indices are negative. For the default method `sobol`, this indicates that the algorithm has not converged and cannot give realiable values for these sensitivity indices. If this happens, please use a larger number of samples. Here we will focus on the first-order indices.

[37]:
```
output_imp = calc_imp.sensitivity(output_imp)
```

Similarly to the uncertainty case, the data is stored in dataframe attributes.

[38]:
```
output_imp.sensitivity_metrics
```

[38]:
```
['aai_agg', 'freq_curve', 'tot_value']
```

[39]:
```
output_imp.get_sens_df('aai_agg').tail()
```

[39]:
```
         si param  param2  aai_agg
65  S2_conf     k   x_exp      NaN
66  S2_conf     k       G      NaN
67  S2_conf     k  v_half      NaN
68  S2_conf     k    vmin      NaN
69  S2_conf     k       k      NaN
```

To obtain the sensitivity interms of a particular sensitivity index, use the method `get_sensisitivity()`. If none is specified, the value of the index for all metrics is returned.

[40]:
```
output_imp.get_sensitivity('S1')
```

[40]:

	si	param	param2	aai_agg	rp50	rp100	rp250	tot_value
0	S1	x_exp	None	0.001040	0.000993	0.000930	0.001150	1.005253
1	S1	G	None	0.073408	0.075781	0.084662	0.093718	0.000000
2	S1	v_half	None	0.514220	0.553640	0.596659	0.619366	0.000000
3	S1	vmin	None	0.012642	0.014407	0.012068	0.010065	0.000000
4	S1	k	None	0.213491	0.189862	0.134867	0.095861	0.000000

Sometimes, it is useful to simply know what is the largest sensitivity index for each metric.

[41]:
```
output_imp.get_largest_si(salib_si='S1')
```

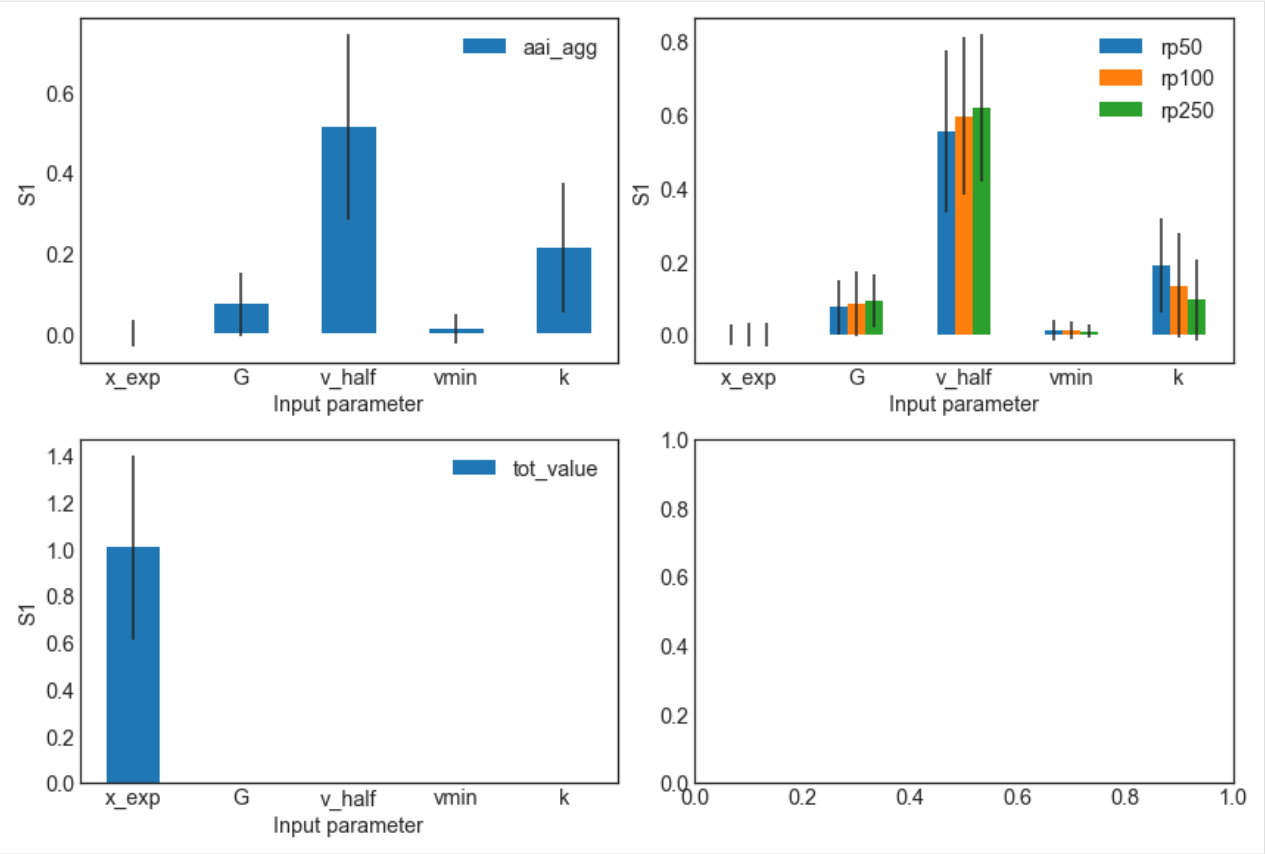
[41]:

	metric	param	param2	si
0	aai_agg	v_half	None	0.514220
1	rp50	v_half	None	0.553640
2	rp100	v_half	None	0.596659
3	rp250	v_half	None	0.619366
4	tot_value	x_exp	None	1.005253

The value of the sensitivity indices can be plotted for each metric that is one-dimensional (`eai_exp` and `at_event` are not shown in this plot).

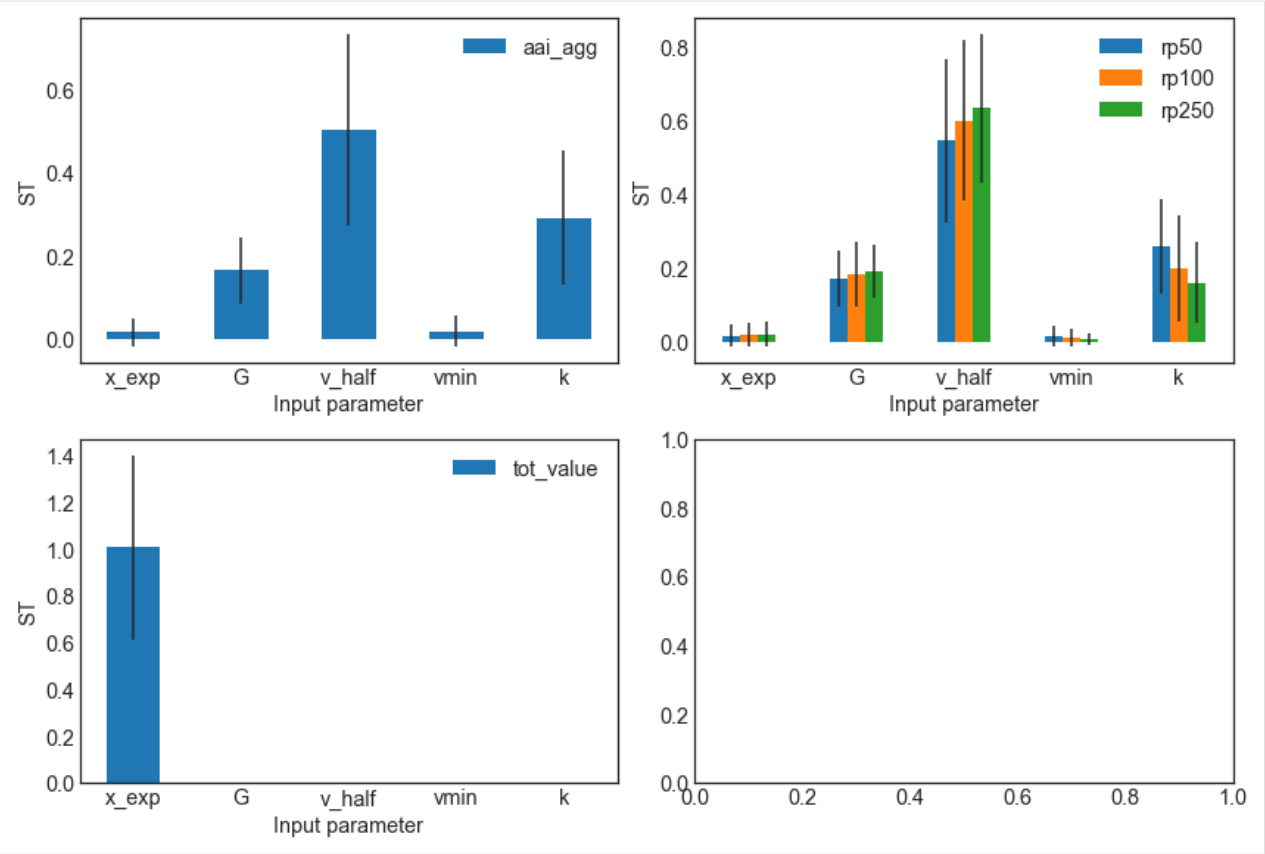
As expected, the `tot_value` of the exposure is only dependent on the exposure parameter `x_exp`. We further see that both the errors in `freq_curve` and in `aai_agg` are mostly determined by `x_exp` and `v_half`. Finally, we see small differences in the sensitivity of the different return periods.

[42]:
```
# Default for 'sobol' is to plot 'S1' sensitivity index.
output_imp.plot_sensitivity(figsize=(12,8));
```



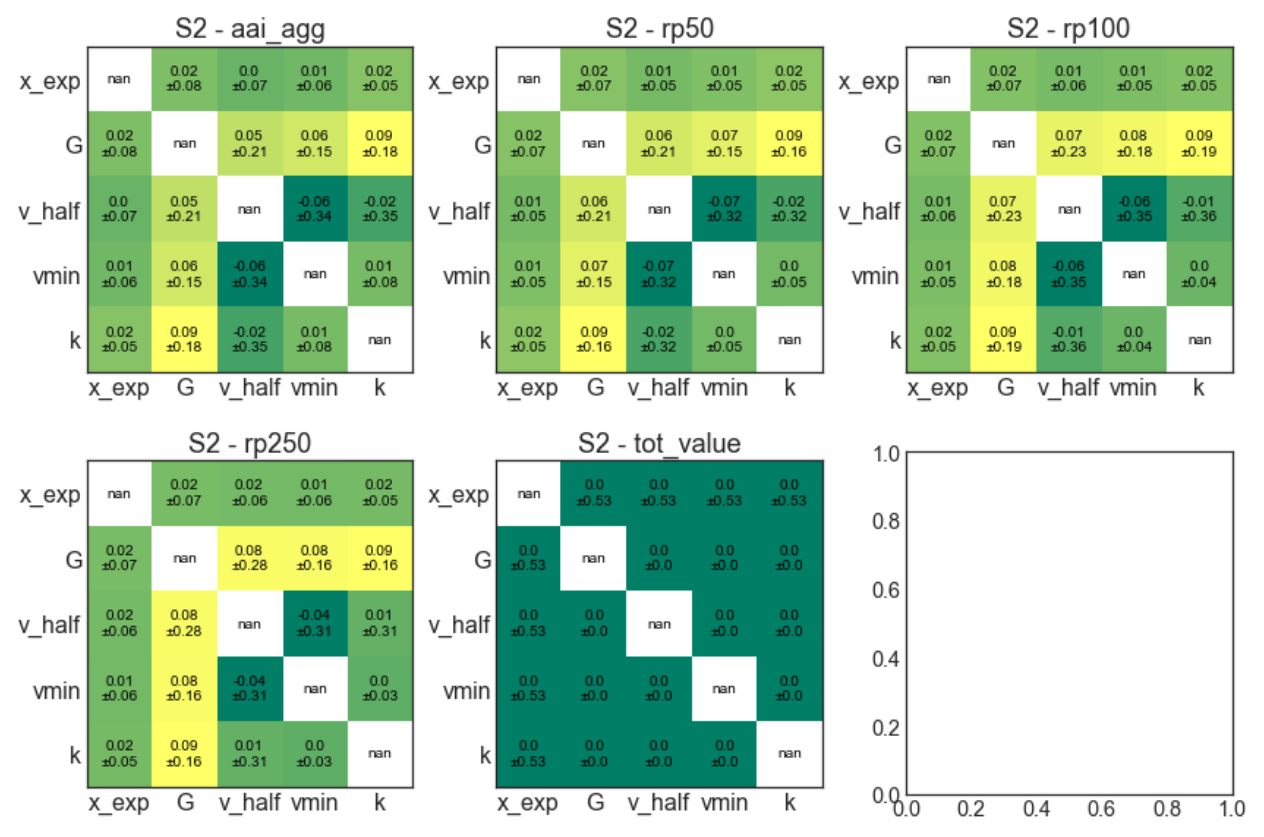
Note that since we have quite a few measures, the imp_meas_fut and imp_meas_pres plots are too crowded. We can select only the other metrics easily. In addition, instead of showing first order sensitivity 'S1', we can plot the total sensitivity 'ST'.

[43]:
```
output_imp.plot_sensitivity(salib_si = 'ST', figsize=(12,8));
```



One can also vizualise the second-order sensitivity indices in the form of a correlation matrix.

[44]:
```
output_imp.plot_sensitivity_second_order(figsize=(12,8));
```



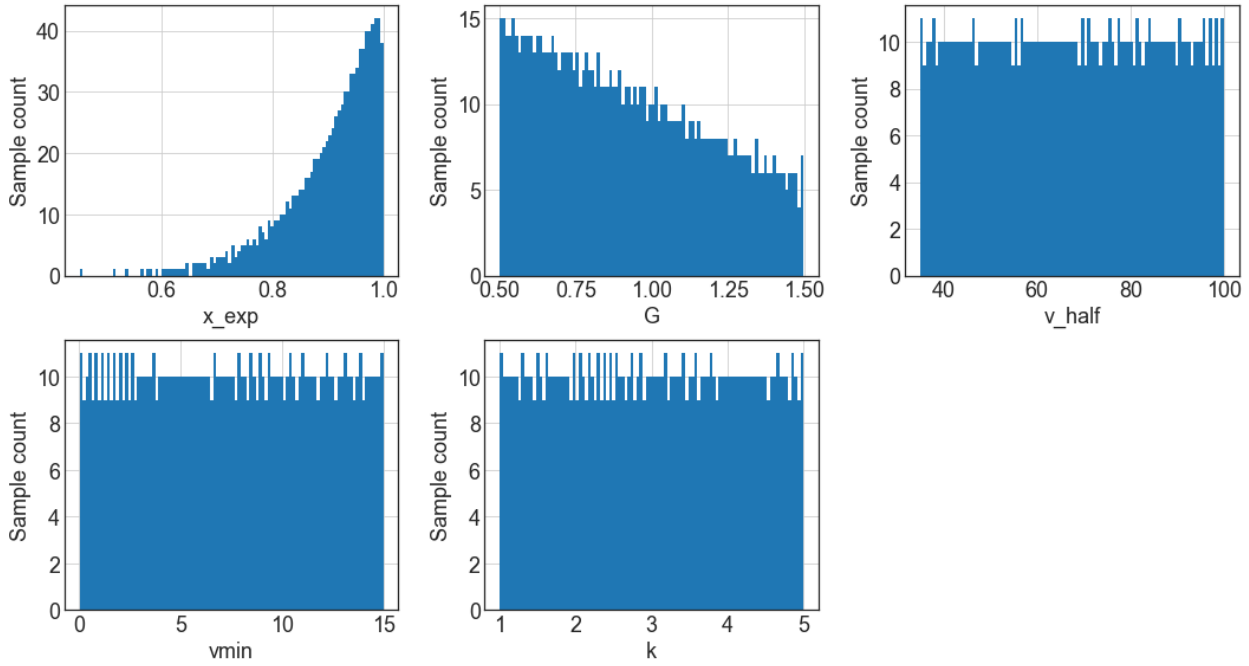
A few non-default parameters

We shall use the same uncertainty variables as in the previous section but show a few possibilities to use non-default method arguments.

[45]:
```
# Sampling method "latin" hypercube instead of `saltelli`.
from climada.engine.unsequa import CalcImpact

calc_imp2 = CalcImpact(exp_iv, impf_iv, haz)
output_imp2 = calc_imp2.make_sample(N=1000, sampling_method='latin')
```

```
2022-01-10 21:12:35,887 - climada.engine.unsequa.calc_base - INFO - Effective number of␣
↪made samples: 1000
```

[46]:
```
output_imp2.plot_sample(figsize=(15,8));
```



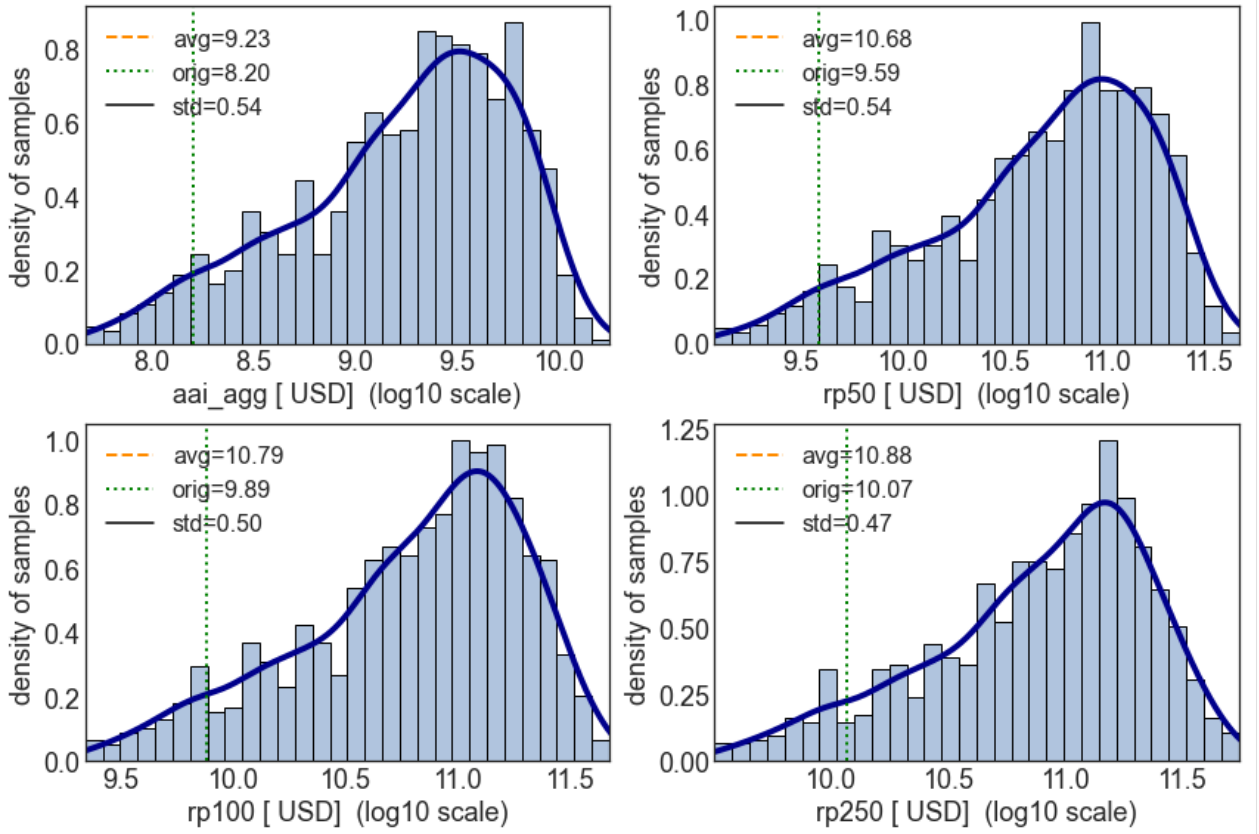
[47]:
```
# Compute also the distribution of the metric `eai_exp`
# To speed-up the comutations, we use a ProcessPool for parallel computations
from pathos.pools import ProcessPool as Pool
pool = Pool()
output_imp2 = calc_imp2.uncertainty(output_imp2, rp = [50, 100, 250], calc_eai_exp=True,␣
↪calc_at_event=True, pool=pool)
pool.close() #Do not forget to close your pool!
pool.join()
pool.clear()
```

```
2022-02-11 16:35:43,535 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
```

[48]:
```
# Add the original value of the impacts (without uncertainty) to the uncertainty plot
from climada.engine import Impact
imp = Impact()
imp.calc(exp_base, impf_func(), haz)
aai_agg_o = imp.aai_agg
freq_curve_o = imp.calc_freq_curve([50, 100, 250]).impact
orig_list = [aai_agg_o] + list(freq_curve_o) +[1]
```

```
2022-01-10 21:12:47,695 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
2022-01-10 21:12:47,697 - climada.engine.impact - INFO - Exposures matching centroids␣
↪found in centr_TC
2022-01-10 21:12:47,699 - climada.engine.impact - INFO - Calculating damage for 50␣
↪assets (>0) and 216 events.
```

[49]:
```
# plot the aai_agg and freq_curve uncertainty only
# use logarithmic x-scale
output_imp2.plot_uncertainty(metric_list=['aai_agg', 'freq_curve'], orig_list=orig_list,␣
↪log=True, figsize=(12,8));
```

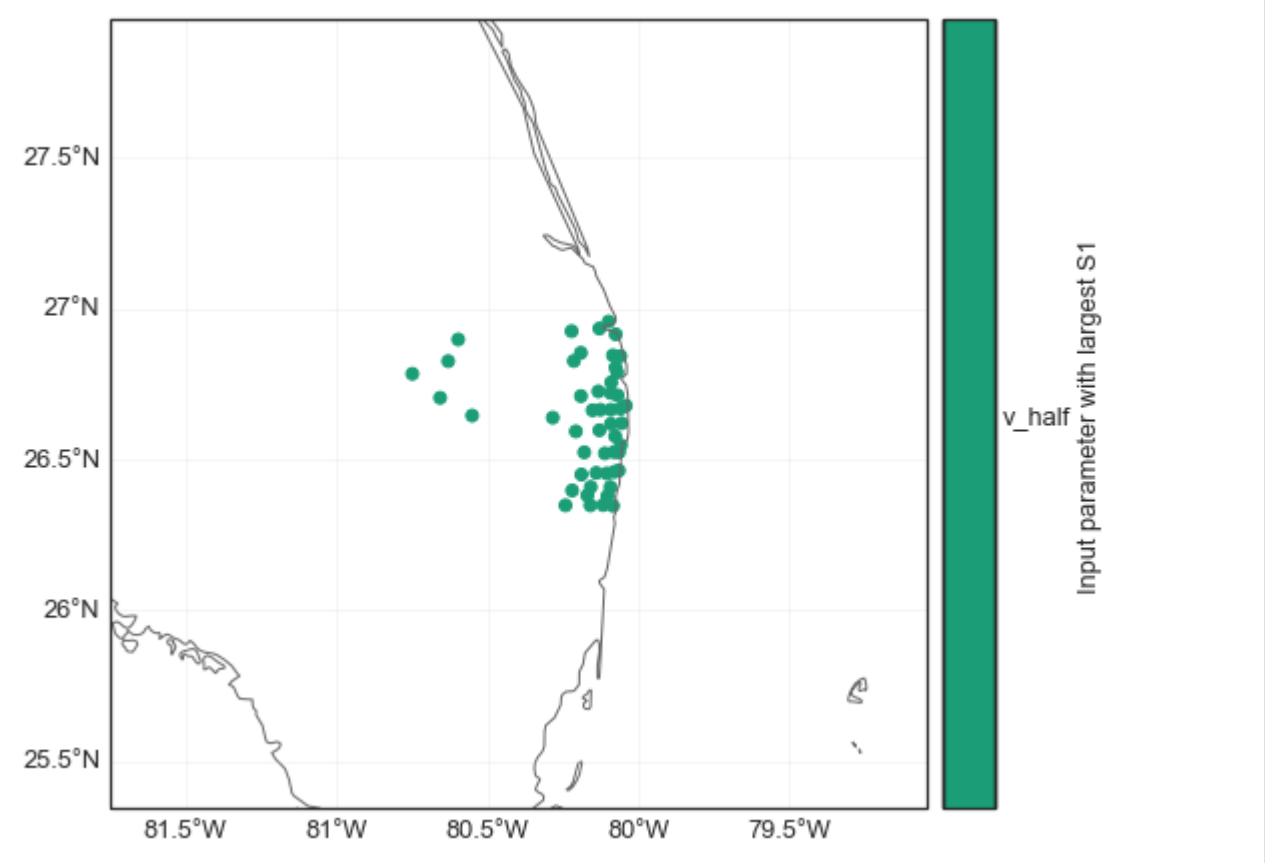


[50]:
```
# Use the method 'rbd_fast' which is recommend in pair with 'latin'. In addition, change␣
↪one of the kwargs
# (M=15) of the salib sampling method.
output_imp2 = calc_imp2.sensitivity(output_imp2, sensitivity_method='rbd_fast',␣
↪sensitivity_kwargs = {'M': 15})
```

```
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪rbd_fast.py:106: RuntimeWarning: invalid value encountered in double_scalars
  return D1 / V
```

Since we computed the distribution and sensitivity indices for the total impact at each exposure point, we can plot a map of the largest sensitivity index in each exposure location. For every location, the most sensitive parameter is `v_half`, meaning that the average annual impact at each location is most sensitivity to the ucnertainty in the impact function slope scaling parameter.

[51]:
```
output_imp2.plot_sensitivity_map();
```



[52]:
```
output_imp2.get_largest_si(salib_si='S1', metric_list=['eai_exp']).tail()
```

[52]:

	metric	param	param2	si
45	45	v_half	None	0.479974
46	46	v_half	None	0.479974
47	47	v_half	None	0.479974
48	48	v_half	None	0.475221
49	49	v_half	None	0.479974

5.13.6 CalcCostBenefit

The uncertainty and sensitivity analysis for CostBenefit is completely analogous to the Impact case. It is slightly more complex as there are more input variables.

Set the InputVars

[53]:
```
import copy
from climada.util.constants import ENT_DEMO_TODAY, ENT_DEMO_FUTURE, HAZ_DEMO_H5
from climada.entity import Entity
from climada.hazard import Hazard

# Entity today has an uncertainty in the total asset value
def ent_today_func(x_ent):
```

(continues on next page)

(continued from previous page)

```
    #In-function imports needed only for parallel computing on Windows
    from climada.entity import Entity
    from climada.util.constants import ENT_DEMO_TODAY
    entity = Entity.from_excel(ENT_DEMO_TODAY)
    entity.exposures.ref_year = 2018
    entity.exposures.gdf.value *= x_ent
    return entity

# Entity in the future has a +- 10% uncertainty in the cost of all the adapatation␣
↪measures
def ent_fut_func(m_fut_cost):
    #In-function imports needed only for parallel computing on Windows
    from climada.entity import Entity
    from climada.util.constants import ENT_DEMO_FUTURE
    entity = Entity.from_excel(ENT_DEMO_FUTURE)
    entity.exposures.ref_year = 2040
    for meas in entity.measures.get_measure('TC'):
        meas.cost *= m_fut_cost
    return entity

haz_base = Hazard.from_hdf5(HAZ_DEMO_H5)
# The hazard intensity in the future is also uncertainty by a multiplicative factor
def haz_fut(x_haz_fut, haz_base):
    #In-function imports needed only for parallel computing on Windows
    import copy
    from climada.hazard import Hazard
    from climada.util.constants import HAZ_DEMO_H5
    haz = copy.deepcopy(haz_base)
    haz.intensity = haz.intensity.multiply(x_haz_fut)
    return haz
from functools import partial
haz_fut_func = partial(haz_fut, haz_base=haz_base)
```

```
2022-01-10 21:12:58,058 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
```

Check that costs for measures are changed as desired.

[54]:
```
costs_1 = [meas.cost for meas in ent_fut_func(1).measures.get_measure('TC')]
costs_05 = [meas.cost for meas in ent_fut_func(0.5).measures.get_measure('TC')]
print(f"\nThe cost for m_fut_cost=1 are {costs_1}\n"
      f"The cost for m_fut_cost=0.5 are {costs_05}");
```

```
The cost for m_fut_cost=1 are [1311768360.8515418, 1728000000.0, 8878779433.630093,␣
↪9200000000.0]
The cost for m_fut_cost=0.5 are [655884180.4257709, 864000000.0, 4439389716.815046,␣
↪4600000000.0]
```

```
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/openpyxl/
↪worksheet/_reader.py:312: UserWarning: Unknown extension is not supported and will be␣
↪removed
  warn(msg)
```

Define the InputVars

[55]:
```
import scipy as sp
from climada.engine.unsequa import InputVar

haz_today = haz_base

haz_fut_distr = {"x_haz_fut": sp.stats.uniform(1, 3),
                 }
haz_fut_iv = InputVar(haz_fut_func, haz_fut_distr)

ent_today_distr = {"x_ent":  sp.stats.uniform(0.7, 1)}
ent_today_iv = InputVar(ent_today_func, ent_today_distr)

ent_fut_distr = {"m_fut_cost":  sp.stats.norm(1, 0.1)}
ent_fut_iv = InputVar(ent_fut_func, ent_fut_distr)
```

[56]:
```
ent_avg = ent_today_iv.evaluate()
ent_avg.exposures.gdf.head()
```

[56]:
```
    latitude  longitude         value  deductible         cover  impf_TC  \
0  26.933899 -80.128799  1.671301e+10           0  1.392750e+10        1
1  26.957203 -80.098284  1.511528e+10           0  1.259606e+10        1
2  26.783846 -80.748947  1.511528e+10           0  1.259606e+10        1
3  26.645524 -80.550704  1.511528e+10           0  1.259606e+10        1
4  26.897796 -80.596929  1.511528e+10           0  1.259606e+10        1

     Value_2010
0  5.139301e+09
1  4.647994e+09
2  4.647994e+09
3  4.647994e+09
4  4.647994e+09
```

Compute cost benefit uncertainty and sensitivity using default methods

For examples of how to use non-defaults please see the *impact example*

[57]:
```
from climada.engine.unsequa import CalcCostBenefit

unc_cb = CalcCostBenefit(haz_input_var=haz_today, ent_input_var=ent_today_iv,
                         haz_fut_input_var=haz_fut_iv, ent_fut_input_var=ent_fut_iv)
```

[58]:
```
output_cb= unc_cb.make_sample(N=10, sampling_kwargs={'calc_second_order':False})
output_cb.get_samples_df().tail()
```

```
2022-01-10 21:13:07,531 - climada.engine.unsequa.calc_base - INFO - Effective number of␣
↪made samples: 50
```

[58]:
```
       x_ent  x_haz_fut  m_fut_cost
45  1.263477   3.071289    1.012517
46  1.658008   3.071289    1.012517
47  1.263477   1.372070    1.012517
```

(continues on next page)

(continued from previous page)

```
48  1.263477   3.071289    1.067757
49  1.658008   1.372070    1.067757
```

For longer computations, it is possible to use a pool for parallel computation.

[68]:
```
from pathos.pools import ProcessPool as Pool

#without pool
output_cb = unc_cb.uncertainty(output_cb)

#with pool
#pool = Pool()
#output_cb = unc_cb.uncertainty(output_cb, pool=pool)
#pool.close() #Do not forget to close your pool!
#pool.join()
#pool.clear()
#If you have issues with the pool in jupyter, please restart the kernel or use without␣
↪pool.
```

The output of `CostBenefit.calc` is rather complex in its structure. The metrics dictionary inherits this complexity.

[60]:
```
#Top level metrics keys
macro_metrics = output_cb.uncertainty_metrics
macro_metrics
```

[60]:
```
['imp_meas_present',
 'imp_meas_future',
 'tot_climate_risk',
 'benefit',
 'cost_ben_ratio']
```

[61]:
```
# The benefits and cost_ben_ratio are available for each measure
output_cb.get_uncertainty(metric_list=['benefit', 'cost_ben_ratio']).tail()
```

[61]:
```
    Mangroves Benef  Beach nourishment Benef  Seawall Benef  \
45     8.814039e+09             6.914137e+09   7.514788e+08
46     8.966634e+09             7.038734e+09   7.514788e+08
47     3.768293e+09             3.046279e+09   4.742905e+06
48     8.814039e+09             6.914137e+09   7.514788e+08
49     3.920888e+09             3.170876e+09   4.742905e+06

    Building code Benef  Mangroves CostBen  Beach nourishment CostBen  \
45         4.050774e+10           0.150690                   0.253051
46         4.058393e+10           0.148126                   0.248572
47         2.438938e+09           0.352464                   0.574350
48         4.050774e+10           0.158911                   0.266857
49         2.515132e+09           0.357228                   0.581884

    Seawall CostBen  Building code CostBen
45        11.962963               0.229960
46        11.962963               0.229528
47      1895.444529               3.819348
48        12.615625               0.242506
```

(continues on next page)

(continued from previous page)

```
49      1998.854177               3.905703
```

[62]:
```
# The impact_meas_present and impact_meas_future provide values of the cost_meas, risk_
↪transf, risk,
# and cost_ins for each measure
output_cb.get_uncertainty(metric_list=['imp_meas_present']).tail()
```

[62]:
```
    no measure - risk - present  no measure - risk_transf - present  \
45                 9.696915e+07                                 0.0
46                 1.272486e+08                                 0.0
47                 9.696915e+07                                 0.0
48                 9.696915e+07                                 0.0
49                 1.272486e+08                                 0.0

    no measure - cost_meas - present  no measure - cost_ins - present  \
45                                 0                                 0
46                                 0                                 0
47                                 0                                 0
48                                 0                                 0
49                                 0                                 0

    Mangroves - risk - present  Mangroves - risk_transf - present  \
45                4.841883e+07                                 0
46                6.353802e+07                                 0
47                4.841883e+07                                 0
48                4.841883e+07                                 0
49                6.353802e+07                                 0

    Mangroves - cost_meas - present  Mangroves - cost_ins - present  \
45                     1.311768e+09                               1
46                     1.311768e+09                               1
47                     1.311768e+09                               1
48                     1.311768e+09                               1
49                     1.311768e+09                               1

    Beach nourishment - risk - present  \
45                        5.732646e+07
46                        7.522713e+07
47                        5.732646e+07
48                        5.732646e+07
49                        7.522713e+07

    Beach nourishment - risk_transf - present  \
45                                          0
46                                          0
47                                          0
48                                          0
49                                          0

    Beach nourishment - cost_meas - present  \
45                             1.728000e+09
46                             1.728000e+09
```

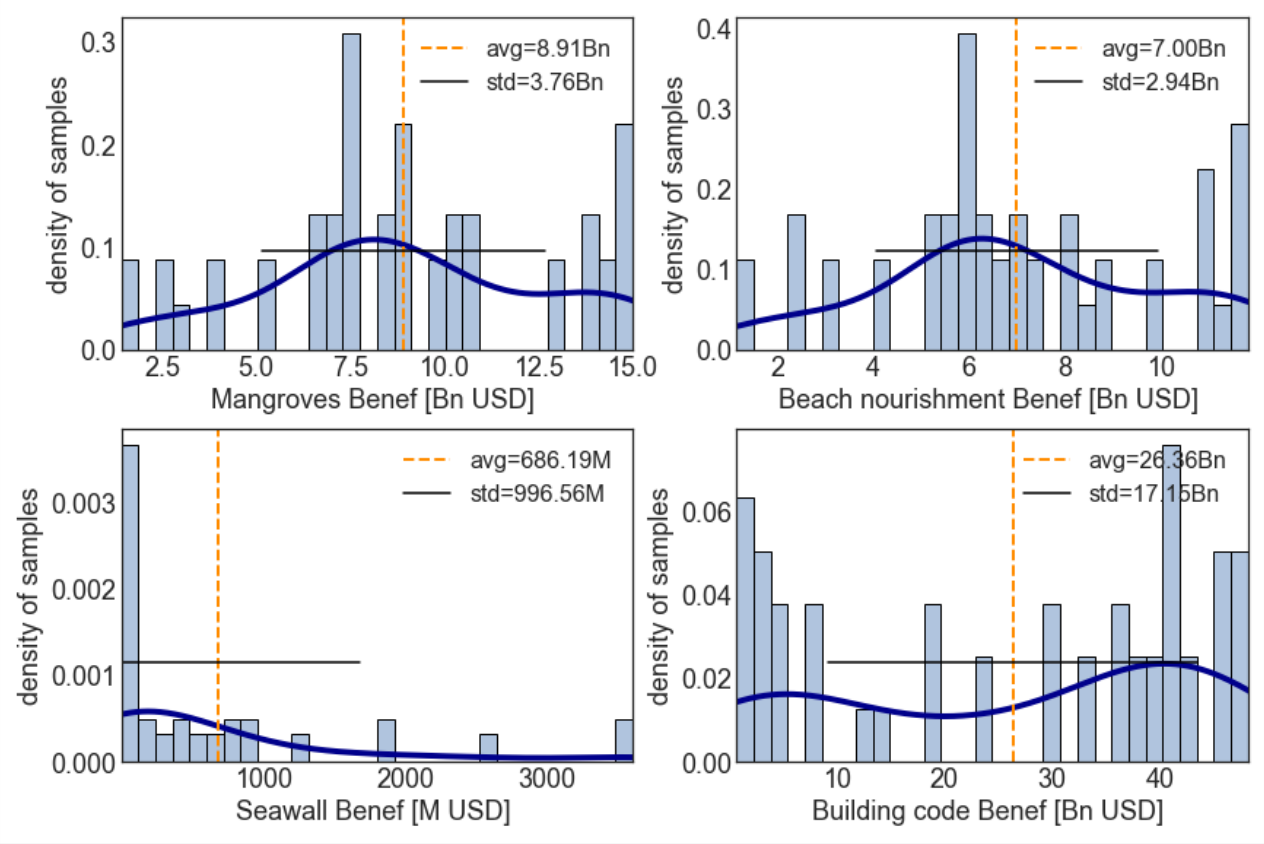
(continues on next page)

(continued from previous page)

```
47                              1.728000e+09
48                              1.728000e+09
49                              1.728000e+09

    Beach nourishment - cost_ins - present  Seawall - risk - present  \
45                                       1              9.696915e+07
46                                       1              1.272486e+08
47                                       1              9.696915e+07
48                                       1              9.696915e+07
49                                       1              1.272486e+08

    Seawall - risk_transf - present  Seawall - cost_meas - present  \
45                                0                   8.878779e+09
46                                0                   8.878779e+09
47                                0                   8.878779e+09
48                                0                   8.878779e+09
49                                0                   8.878779e+09

    Seawall - cost_ins - present  Building code - risk - present  \
45                             1                    7.272686e+07
46                             1                    9.543644e+07
47                             1                    7.272686e+07
48                             1                    7.272686e+07
49                             1                    9.543644e+07

    Building code - risk_transf - present  \
45                                     0
46                                     0
47                                     0
48                                     0
49                                     0

    Building code - cost_meas - present  Building code - cost_ins - present
45                         9.200000e+09                                   1
46                         9.200000e+09                                   1
47                         9.200000e+09                                   1
48                         9.200000e+09                                   1
49                         9.200000e+09                                   1
```

We can plot the distributions for the top metrics or our choice.

[63]:
```
# tot_climate_risk and benefit
output_cb.plot_uncertainty(metric_list=['benefit'], figsize=(12,8));
```



Analogously to the impact example, now that we have a metric distribution, we can compute the sensitivity indices. Since we used the default sampling method, we can use the default sensitivity analysis method. However, since we used `calc_second_order = False` for the sampling, we need to specify the same for the sensitivity analysis.

[64]:
```
output_cb = unc_cb.sensitivity(output_cb, sensitivity_kwargs={'calc_second_order':False})
```

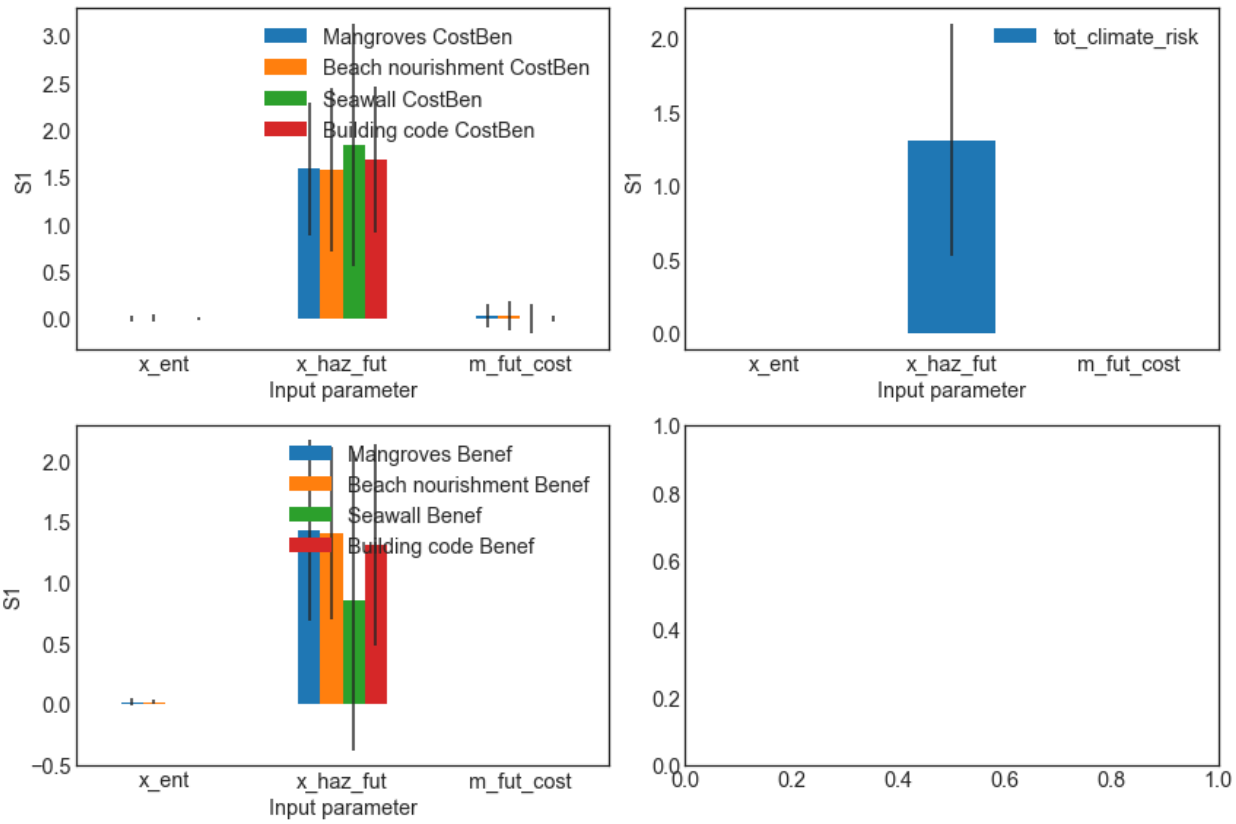
```
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪sobol.py:87: RuntimeWarning: invalid value encountered in true_divide
  Y = (Y - Y.mean()) / Y.std()
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪sobol.py:137: RuntimeWarning: invalid value encountered in double_scalars
  return np.mean(B * (AB - A), axis=0) / np.var(np.r_[A, B], axis=0)
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪sobol.py:137: RuntimeWarning: invalid value encountered in true_divide
  return np.mean(B * (AB - A), axis=0) / np.var(np.r_[A, B], axis=0)
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪sobol.py:143: RuntimeWarning: invalid value encountered in double_scalars
  return 0.5 * np.mean((A - AB) ** 2, axis=0) / np.var(np.r_[A, B], axis=0)
/Users/ckropf/opt/anaconda3/envs/climada_310/lib/python3.8/site-packages/SALib/analyze/
↪sobol.py:143: RuntimeWarning: invalid value encountered in true_divide
  return 0.5 * np.mean((A - AB) ** 2, axis=0) / np.var(np.r_[A, B], axis=0)
```

The sensitivity indices can be plotted. For the default method 'sobol', by default the 'S1' sensitivity index is plotted.

Note that since we have quite a few measures, the plot must be adjusted a bit or dropped. Also see that for many metrics, the sensitivity to certain uncertainty parameters appears to be 0. However, this result is to be treated with care. Indeed, we used for demonstration purposes a rather too low number of samples, which is indicated by large confidence intervals (vertical black lines) for most sensitivity indices. For a more robust result the analysis should be repeated with

more samples.

[66]:
```
#plot only certain metrics
axes = output_cb.plot_sensitivity(metric_list=['cost_ben_ratio','tot_climate_risk',
↪'benefit'], figsize=(12,8));
```



5.14 Helper methods for InputVar

This tutorial complements the general tutorial on the uncertainty and sensitivity analysis module *unsequa*.

The InputVar class provides a few helper methods to generate generic uncertainty input variables for exposures, impact function sets, hazards, and entities (including measures cost and disc rates).

Table of Contents

[1]:
```
import warnings
warnings.filterwarnings('ignore') #Ignore warnings for making the tutorial's pdf.
```

5.14.1 Exposures

The following types of uncertainties can be added:

- ET: scale the total value (homogeneously) The value at each exposure point is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_totvalue
- EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator.
- EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents.

If a bounds is None, this parameter is assumed to have no uncertainty.

Example: single exposures

[2]:
```
#Define the base exposure
from climada.util.constants import EXP_DEMO_H5
from climada.entity import Exposures
exp_base = Exposures.from_hdf5(EXP_DEMO_H5)
```

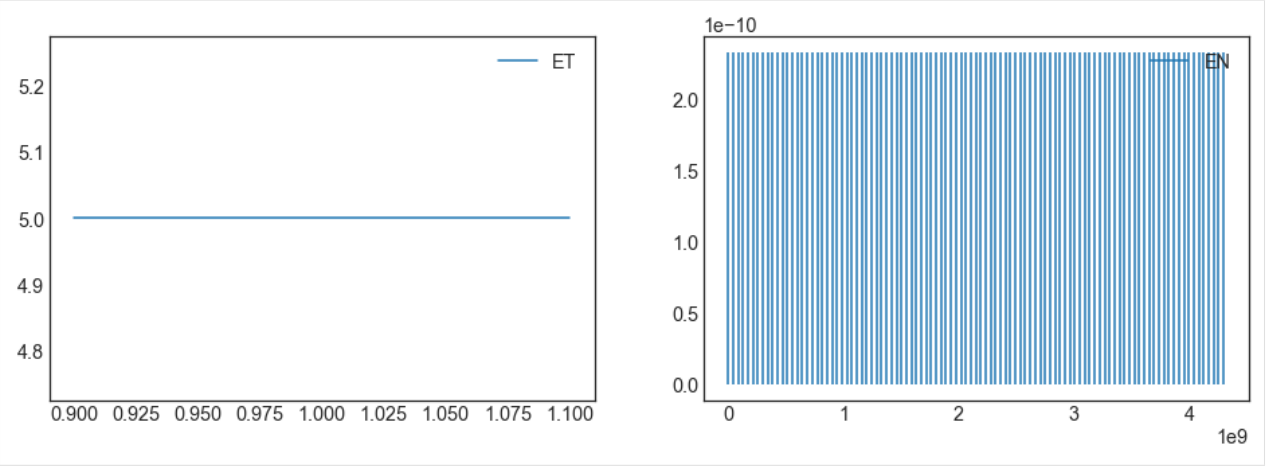
```
2022-01-10 21:10:32,810 - climada.entity.exposures.base - INFO - Reading /Users/ckropf/
↪climada/demo/data/exp_demo_today.h5
```

[3]:
```
from climada.engine.unsequa import InputVar
bounds_totval = [0.9, 1.1] #+- 10% noise on the total exposures value
bounds_noise = [0.9, 1.2] #-10% - +20% noise each exposures point
exp_iv = InputVar.exp([exp_base], bounds_totval, bounds_noise)
```

[4]:
```
#The difference in total value between the base exposure and the average input␣
↪uncertainty exposure
#due to the random noise on each exposures point (the average change in the total value␣
↪is 1.0).
avg_exp = exp_iv.evaluate()
(sum(avg_exp.gdf['value']) - sum(exp_base.gdf['value'])) / sum(exp_base.gdf['value'])
```

[4]:
```
0.03700231587024304
```

[5]:
```
#The values for EN are seeds for the random number generator for the noise sampling and
#thus are uniformly sampled numbers between (0, 2**32-1)
exp_iv.plot();
```



Example: list of litpop exposures with different exponents

[6]:
```
#Define a generic method to make litpop instances with different exponent pairs.
from climada.entity import LitPop
def generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs,
                         choice_mn, **litpop_kwargs):
    #In-function imports needed only for parallel computing on Windows
    from climada.entity import LitPop
    litpop_base = []
    for [m, n] in choice_mn:
        print('\n Computing litpop for m=%d, n=%d \n' %(m, n))
        litpop_kwargs['exponents'] = (m, n)
        exp = LitPop.from_countries(**litpop_kwargs)
        exp.gdf['impf_' + haz.tag.haz_type] = impf_id
        exp.gdf.drop('impf_', axis=1, inplace=True)
        if value_unit is not None:
            exp.value_unit = value_unit
        exp.assign_centroids(haz, **assign_centr_kwargs)
        litpop_base.append(exp)
    return litpop_base
```

[7]:
```
#Define the parameters of the LitPop instances
tot_pop = 11.317e6
impf_id = 1
value_unit = 'people'
litpop_kwargs = {
    'countries' : ['CUB'],
    'res_arcsec' : 300,
    'reference_year' : 2020,
    'fin_mode' : 'norm',
    'total_values' : [tot_pop]
}
assign_centr_kwargs={}

# The hazard is needed to assign centroids
```

(continues on next page)

(continued from previous page)

```
from climada.util.constants import HAZ_DEMO_H5
from climada.hazard import Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
```

```
2022-01-10 21:10:33,708 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
```

[8]:
```
#Generate the LitPop list

choice_mn = [[0, 0.5], [0, 1], [0, 2]] #Choice of exponents m,n

litpop_list = generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs, choice_
↪mn, **litpop_kwargs)
```

```
 Computing litpop for m=0, n=0

2022-01-10 21:10:33,998 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:33,999 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:35,544 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:35,545 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:35,545 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:35,546 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:35,546 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:35,549 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:35,551 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:10:35,577 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=0, n=1

2022-01-10 21:10:35,824 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:35,825 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:37,238 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:37,239 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:37,239 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:37,239 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:37,240 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:37,243 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:37,245 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
```

(continues on next page)

(continued from previous page)

```
2022-01-10 21:10:37,269 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=0, n=2

2022-01-10 21:10:37,499 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:37,501 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:38,888 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:38,889 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:38,889 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:38,890 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:38,891 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:38,894 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:38,895 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:10:38,919 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.
```

[9]:
```
from climada.engine.unsequa import InputVar
bounds_totval = [0.9, 1.1] #+- 10% noise on the total exposures value
litpop_iv = InputVar.exp(exp_list = litpop_list,
                         bounds_totval=bounds_totval)
```

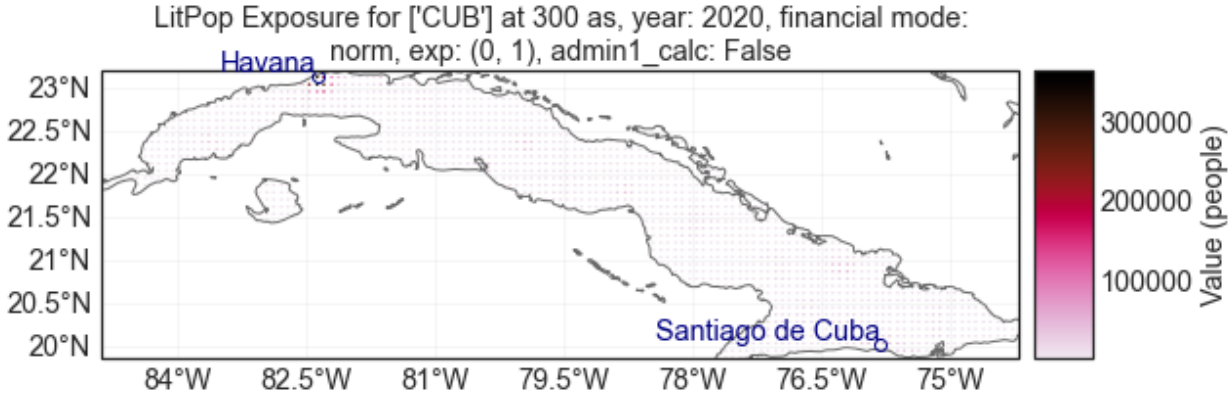
[10]:
```
# To choose n=0.5, we have to set EL=1 (the index of 0.5 in choice_n = [0, 0.5, 1, 2])
pop_half = litpop_iv.evaluate(ET=1, EL=1)
```

[11]:
```
pop_half.gdf.tail()
```

[11]:

	value	geometry	latitude	longitude	region_id	impf_TC	centr_TC
1383	1713.015083	POINT (-78.29167 22.45833)	22.458333	-78.291667	192	1	431
1384	1085.168934	POINT (-79.20833 22.62500)	22.625000	-79.208333	192	1	476
1385	950.764517	POINT (-79.62500 22.79167)	22.791667	-79.625000	192	1	524
1386	1129.619078	POINT (-79.45833 22.70833)	22.708333	-79.458333	192	1	475
1387	300.552289	POINT (-80.79167 23.20833)	23.208333	-80.791667	192	1	617

[12]:
```
pop_half.plot_hexbin();
```

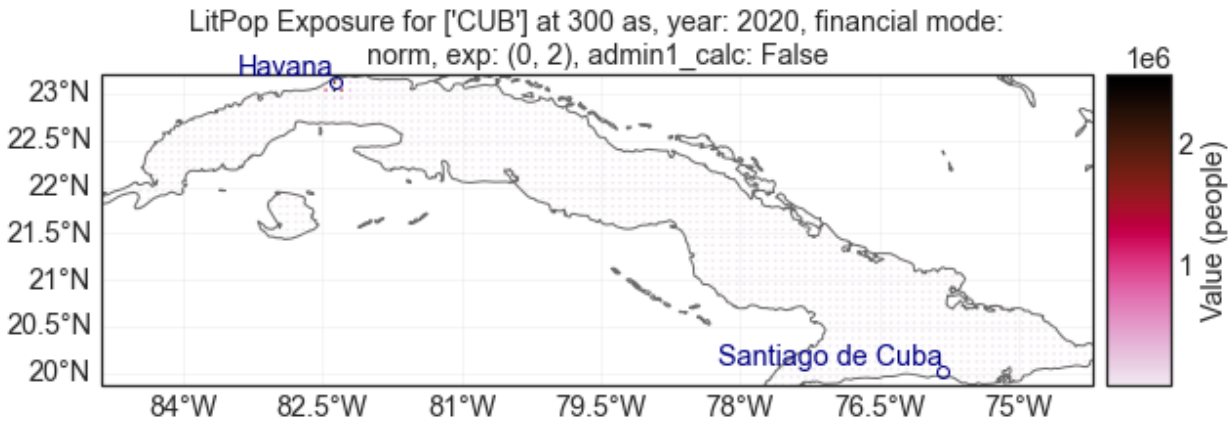


[13]:
```
# To choose n=1, we have to set EL=2 (the index of 1 in choice_n = [0, 0.5, 1, 2])
pop_one = litpop_iv.evaluate(ET=1, EL=2)
```

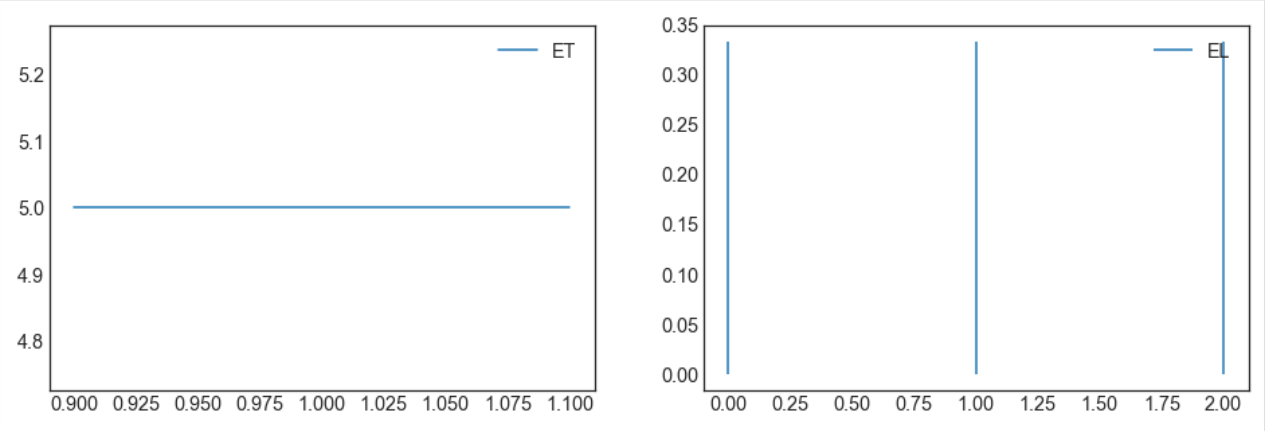
[14]:
```
pop_one.gdf.tail()
```

[14]:
```
          value                    geometry   latitude  longitude  region_id  \
1383  56.491498  POINT (-78.29167 22.45833)  22.458333 -78.291667        192
1384  22.670204  POINT (-79.20833 22.62500)  22.625000 -79.208333        192
1385  17.402300  POINT (-79.62500 22.79167)  22.791667 -79.625000        192
1386  24.565452  POINT (-79.45833 22.70833)  22.708333 -79.458333        192
1387   1.739005  POINT (-80.79167 23.20833)  23.208333 -80.791667        192

      impf_TC  centr_TC
1383        1       431
1384        1       476
1385        1       524
1386        1       475
1387        1       617
```

[15]:
```
pop_one.plot_hexbin();
```



[16]:
```
#The values for EN are seeds for the random number generator for the noise sampling and
#thus are uniformly sampled numbers between (0, 2**32-1)
litpop_iv.plot();
```



5.14.2 Hazard

The following types of uncertainties can be added:

- HE: sub-sampling events from the total event set For each sub-sample, n_ev events are sampled with replacement. HE is the value of the seed for the uniform random number generator.
- HI: scale the intensity of all events (homogeneously) The instensity of all events is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_int
- HF: scale the frequency of all events (homogeneously) The frequency of all events is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_freq

If a bounds is None, this parameter is assumed to have no uncertainty.

[17]:
```
#Define the base exposure
from climada.util.constants import HAZ_DEMO_H5
from climada.hazard import Hazard
haz_base = Hazard.from_hdf5(HAZ_DEMO_H5)
```

```
2022-01-10 21:10:43,742 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
```

[18]:
```
from climada.engine.unsequa import InputVar
bounds_freq = [0.9, 1.1] #+- 10% noise on the frequency of all events
bounds_int = None #No uncertainty on the intensity
n_ev = None
haz_iv = InputVar.haz(haz_base, n_ev=n_ev, bounds_freq=bounds_freq, bounds_int=bounds_
↪int)
```

[19]:
```
#The difference in frequency for HF=1.1 is indeed 10%.
haz_high_freq = haz_iv.evaluate(HE=n_ev, HI=None, HF = 1.1)
(sum(haz_high_freq.frequency) - sum(haz_base.frequency)) / sum(haz_base.frequency)
```

[19]:
```
0.10000000000000736
```

[20]:
```
bounds_freq = [0.9, 1.1] #+- 10% noise on the frequency of all events
bounds_int = None #No uncertainty on the intensity
```

(continues on next page)

(continued from previous page)

```
n_ev = round(0.8 * haz_base.size) #sub-sample with re-draw events to obtain hazards with␣
↪n=0.8*tot_number_events
haz_iv = InputVar.haz(haz_base, n_ev=n_ev, bounds_freq=bounds_freq, bounds_int=bounds_
↪int)
```

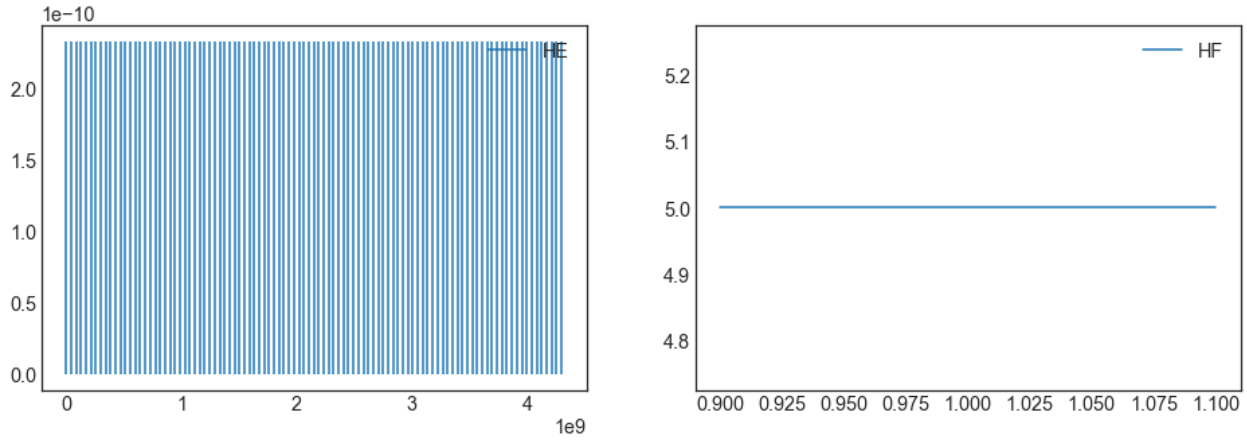
Note that the HE is not a univariate distribution, but for each sample corresponds to the names of the sub-sampled events. However, to simplify the data stream, the HE is saved as the seed for the random number generator that made the smaple. Hence, the value of HE is a label for the given sample. If really needed, the exact chosen events can be obtained as follows.

[21]:
```
import numpy as np
HE = 2618981871 #The random seed (number between 0 and 2**32)
rng = np.random.RandomState(int(HE)) #Initialize a random state with the seed
chosen_ev = list(rng.choice(haz_base.event_name, int(n_ev))) #Obtain the corresponding␣
↪events
```

[22]:
```
#The first event is
chosen_ev[0]
```

[22]:
```
'1998209N11335'
```

[23]:
```
#The values for HE are seeds for the random number generator for the noise sampling and
#thus are uniformly sampled numbers between (0, 2**32-1)
haz_iv.plot();
```



The number of events per sub-sample is equal to n_ev

[24]:
```
#The number of events per sample is equal to n_ev
haz_sub = haz_iv.evaluate(HE=928165924, HI=None, HF = 1.1)
#The number for HE is irrelevant, as all samples have the same n_Ev
haz_sub.size - n_ev
```

[24]:
```
0
```

5.14.3 ImpactFuncSet

The following types of uncertainties can be added: - MDD: scale the mdd (homogeneously) The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd - PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa - IFi: shift the intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_int

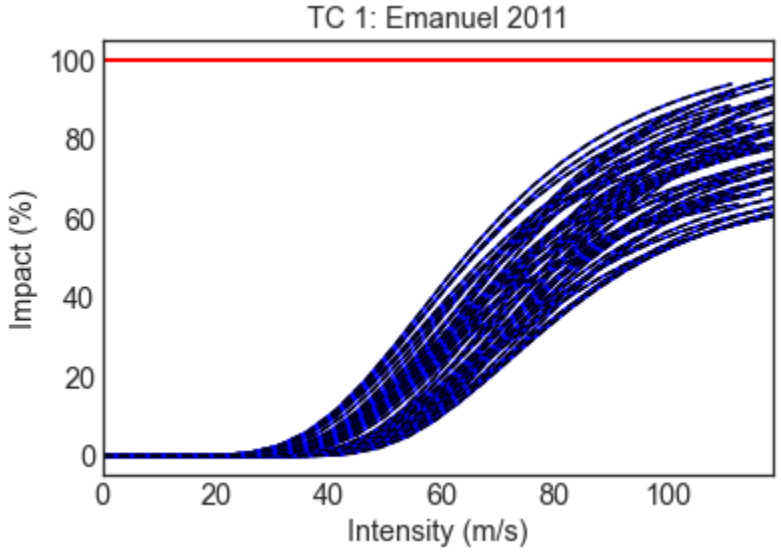
If a bounds is None, this parameter is assumed to have no uncertainty.

[25]:
```
from climada.entity import ImpactFuncSet, ImpfTropCyclone
impf = ImpfTropCyclone.from_emanuel_usa()
impf_set_base = ImpactFuncSet()
impf_set_base.append(impf)
```

It is necessary to specify the hazard type and the impact function id. For simplicity, the default uncertainty input variable only looks at the uncertainty on one single impact function.

[26]:
```
from climada.engine.unsequa import InputVar
bounds_impfi = [-10, 10] #-10 m/s ; +10m/s uncertainty on the intensity
bounds_mdd = [0.7, 1.1] #-30% - +10% uncertainty on the mdd
bounds_paa = None #No uncertainty in the paa
impf_iv = InputVar.impfset(impf_set_base,
                           bounds_impfi=bounds_impfi,
                           bounds_mdd=bounds_mdd,
                           bounds_paa=bounds_paa,
                           haz_id_dict={'TC': [1]})
```

[27]:
```
#Plot the impact function for 50 random samples (note for the expert, these are not␣
↪global)
n = 50
ax = impf_iv.evaluate().plot()
inten = impf_iv.distr_dict['IFi'].rvs(size=n)
mdd = impf_iv.distr_dict['MDD'].rvs(size=n)
for i, m in zip(inten, mdd):
    impf_iv.evaluate(IFi=i, MDD=m).plot(axis=ax)
ax.get_legend().remove()
```



5.14.4 Entity

The following types of uncertainties can be added: - DR: value of constant discount rate (homogeneously) The value of the discounts in each year is sampled uniformly from a distribution with (min, max) = bounds_disc - CO: scale the cost (homogeneously) The cost of all measures is multiplied by the same number sampled uniformly from a distribution with (min, max) = bounds_cost - ET: scale the total value (homogeneously) The value at each exposure point is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_totval - EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator. - EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents. - MDD: scale the mdd (homogeneously) The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd - PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa - IFi: shift the intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_int

If a bounds is None, this parameter is assumed to have no uncertainty.

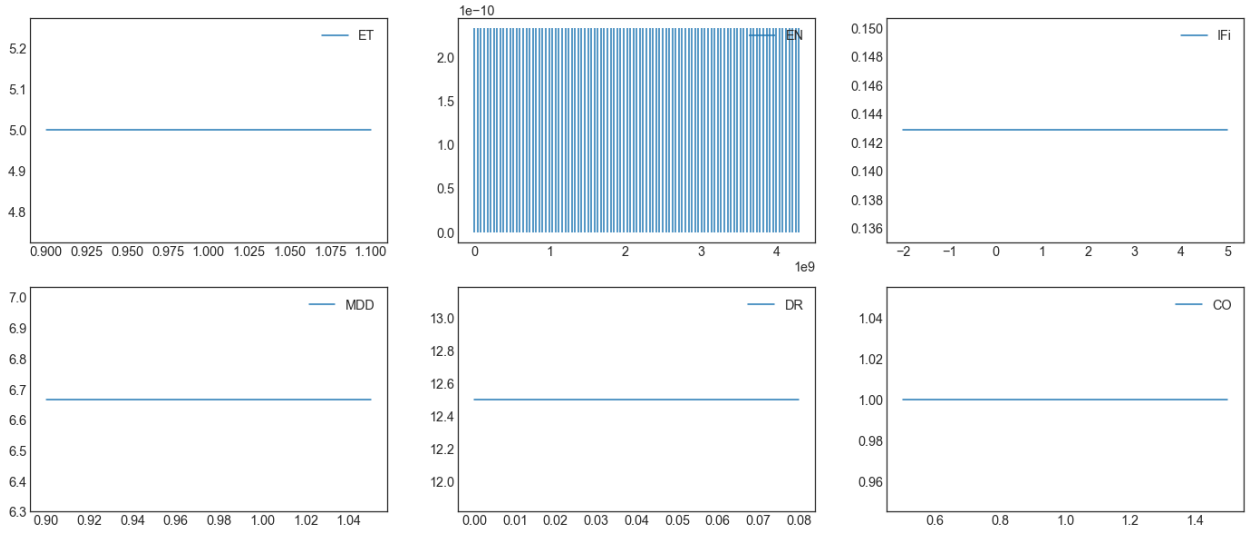
Example: single exposures

[28]:
```
from climada.entity import Entity
from climada.util.constants import ENT_DEMO_TODAY
ent = Entity.from_excel(ENT_DEMO_TODAY)
ent.exposures.ref_year = 2018
ent.check()
```

```
2022-01-10 21:10:47,184 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:47,185 - climada.entity.exposures.base - INFO - geometry not set.
2022-01-10 21:10:47,185 - climada.entity.exposures.base - INFO - region_id not set.
2022-01-10 21:10:47,185 - climada.entity.exposures.base - INFO - centr_ not set.
```

[29]:
```
from climada.engine.unsequa import InputVar
ent_iv = InputVar.ent(
    impf_set = ent.impact_funcs,
    disc_rate = ent.disc_rates,
    exp_list = [ent.exposures],
    meas_set = ent.measures,
    bounds_disc=[0, 0.08],
    bounds_cost=[0.5, 1.5],
    bounds_totval=[0.9, 1.1],
    bounds_noise=[0.3, 1.9],
    bounds_mdd=[0.9, 1.05],
    bounds_paa=None,
    bounds_impfi=[-2, 5],
    haz_id_dict={'TC': [1]}
    )
```

[30]:
```
ent_iv.plot();
```



Example: list of Litpop exposures with different exponents

[31]:
```
#Define a generic method to make litpop instances with different exponent pairs.
from climada.entity import LitPop
def generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs,
                         choice_mn, **litpop_kwargs):
    #In-function imports needed only for parallel computing on Windows
    from climada.entity import LitPop
    litpop_base = []
    for [m, n] in choice_mn:
        print('\n Computing litpop for m=%d, n=%d \n' %(m, n))
        litpop_kwargs['exponents'] = (m, n)
        exp = LitPop.from_countries(**litpop_kwargs)
        exp.gdf['impf_' + haz.tag.haz_type] = impf_id
        exp.gdf.drop('impf_', axis=1, inplace=True)
        if value_unit is not None:
```

(continues on next page)

(continued from previous page)

```
            exp.value_unit = value_unit
        exp.assign_centroids(haz, **assign_centr_kwargs)
        litpop_base.append(exp)
    return litpop_base
```

[32]:
```
#Define the parameters of the LitPop instances
impf_id = 1
value_unit = None
litpop_kwargs = {
    'countries' : ['CUB'],
    'res_arcsec' : 300,
    'reference_year' : 2020,
}
assign_centr_kwargs={}

# The hazard is needed to assign centroids
from climada.util.constants import HAZ_DEMO_H5
from climada.hazard import Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
```

```
2022-01-10 21:10:47,851 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
```

[33]:
```
#Generate the LitPop list

choice_mn = [[1, 0.5], [0.5, 1], [1, 1]] #Choice of exponents m,n

litpop_list = generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs, choice_
↪mn, **litpop_kwargs)
```

```
 Computing litpop for m=1, n=0

2022-01-10 21:10:48,109 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:48,111 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:49,491 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:10:50,565 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:10:50,570 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:10:50,582 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:50,583 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:50,583 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:50,584 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:50,584 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:50,589 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
```

(continues on next page)

(continued from previous page)

```
2022-01-10 21:10:50,592 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:10:50,624 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=0, n=1

2022-01-10 21:10:50,872 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:50,874 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:52,432 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:10:52,863 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:10:52,867 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:10:52,877 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:52,878 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:52,878 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:52,879 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:52,879 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:52,884 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:52,886 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:10:52,914 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=1, n=1

2022-01-10 21:10:53,159 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:53,161 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:54,584 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:10:54,997 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:10:55,001 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:10:55,011 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:55,012 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:55,012 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:55,013 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:55,013 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:55,018 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:55,020 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
```

(continues on next page)

(continued from previous page)

```
2022-01-10 21:10:55,046 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.
```

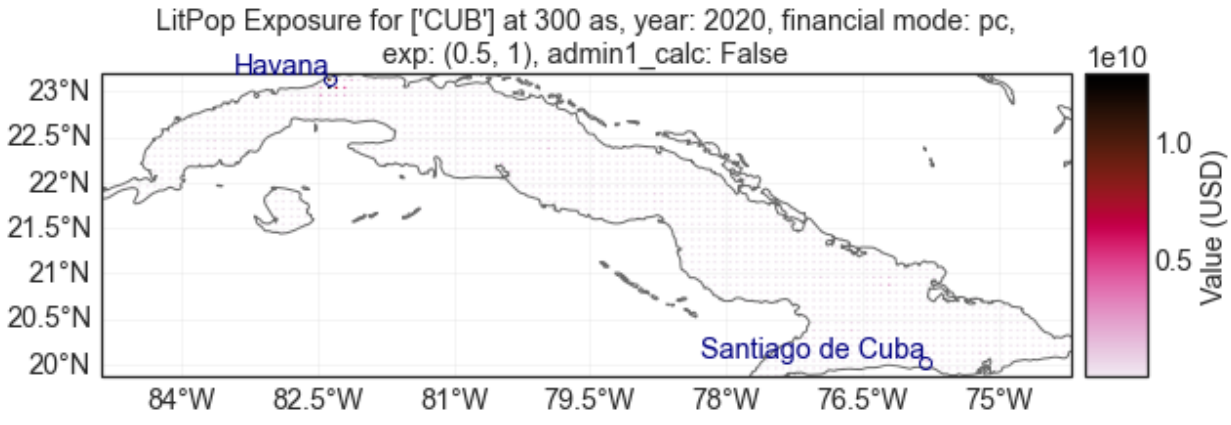
[34]:
```
from climada.entity import Entity
from climada.util.constants import ENT_DEMO_TODAY
ent = Entity.from_excel(ENT_DEMO_TODAY)
ent.exposures.ref_year = 2020
ent.check()
```

```
2022-01-10 21:10:55,139 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:55,140 - climada.entity.exposures.base - INFO - geometry not set.
2022-01-10 21:10:55,140 - climada.entity.exposures.base - INFO - region_id not set.
2022-01-10 21:10:55,141 - climada.entity.exposures.base - INFO - centr_ not set.
```

[35]:
```
from climada.engine.unsequa import InputVar
ent_iv = InputVar.ent(
    impf_set = ent.impact_funcs,
    disc_rate = ent.disc_rates,
    exp_list = litpop_list,
    meas_set = ent.measures,
    bounds_disc=[0, 0.08],
    bounds_cost=[0.5, 1.5],
    bounds_totval=[0.9, 1.1],
    bounds_noise=[0.3, 1.9],
    bounds_mdd=[0.9, 1.05],
    bounds_paa=None,
    bounds_impfi=[-2, 5],
    haz_id_dict={'TC': [1]}
    )
```

[36]:
```
ent_iv.evaluate().exposures.plot_hexbin()
```

[36]:
```
<GeoAxesSubplot:title={'center':"LitPop Exposure for ['CUB'] at 300 as, year: 2020,␣
↪financial mode: pc,\nexp: (0.5, 1), admin1_calc: False"}>
```



5.14.5 Entity Future

The following types of uncertainties can be added: - CO: scale the cost (homogeneously) The cost of all measures is multiplied by the same number sampled uniformly from a distribution with (min, max) = bounds_cost - EG: scale the exposures growth (homogeneously) The value at each exposure point is multiplied by a number sampled uniformly from a distribution with - EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator. - EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents. - MDD: scale the mdd (homogeneously) The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd - PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa - IFi: shift the impact function intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_impfi

If a bounds is None, this parameter is assumed to have no uncertainty.

Example: single exposures

[37]:
```
from climada.entity import Entity
from climada.util.constants import ENT_DEMO_FUTURE

ent_fut = Entity.from_excel(ENT_DEMO_FUTURE)
ent_fut.exposures.ref_year = 2040
ent_fut.check()
```

```
2022-01-10 21:10:57,507 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:57,507 - climada.entity.exposures.base - INFO - geometry not set.
2022-01-10 21:10:57,508 - climada.entity.exposures.base - INFO - region_id not set.
2022-01-10 21:10:57,509 - climada.entity.exposures.base - INFO - centr_ not set.
```

[38]:
```
entfut_iv = InputVar.entfut(
    impf_set = ent_fut.impact_funcs,
    exp_list = [ent_fut.exposures],
    meas_set = ent_fut.measures,
    bounds_cost=[0.6, 1.2],
    bounds_eg=[0.8, 1.5],
    bounds_noise=None,
    bounds_mdd=[0.7, 0.9],
    bounds_paa=[1.3, 2],
    haz_id_dict={'TC': [1]}
    )
```

Example: list of exposures

[39]:
```
#Define a generic method to make litpop instances with different exponent pairs.
from climada.entity import LitPop
def generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs,
                         choice_mn, **litpop_kwargs):
    #In-function imports needed only for parallel computing on Windows
    from climada.entity import LitPop
    litpop_base = []
    for [m, n] in choice_mn:
        print('\n Computing litpop for m=%d, n=%d \n' %(m, n))
        litpop_kwargs['exponents'] = (m, n)
        exp = LitPop.from_countries(**litpop_kwargs)
        exp.gdf['impf_' + haz.tag.haz_type] = impf_id
        exp.gdf.drop('impf_', axis=1, inplace=True)
        if value_unit is not None:
            exp.value_unit = value_unit
        exp.assign_centroids(haz, **assign_centr_kwargs)
        litpop_base.append(exp)
    return litpop_base
```

[40]:
```
#Define the parameters of the LitPop instances
impf_id = 1
value_unit = None
litpop_kwargs = {
    'countries' : ['CUB'],
    'res_arcsec' : 300,
    'reference_year' : 2040,
}
assign_centr_kwargs={}

# The hazard is needed to assign centroids
from climada.util.constants import HAZ_DEMO_H5
from climada.hazard import Hazard
haz = Hazard.from_hdf5(HAZ_DEMO_H5)
```

```
2022-01-10 21:10:57,529 - climada.hazard.base - INFO - Reading /Users/ckropf/climada/
↪demo/data/tc_fl_1990_2004.h5
```

[41]:
```
#Generate the LitPop list

choice_mn = [[1, 0.5], [0.5, 1], [1, 1]] #Choice of exponents m,n

litpop_list = generate_litpop_base(impf_id, value_unit, haz, assign_centr_kwargs, choice_
↪mn, **litpop_kwargs)
```

```
 Computing litpop for m=1, n=0

2022-01-10 21:10:57,798 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...
```

(continues on next page)

(continued from previous page)

```
2022-01-10 21:10:57,799 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2040. Using nearest available year for GPW data: 2020
2022-01-10 21:10:57,800 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:10:59,239 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:10:59,680 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:10:59,683 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:10:59,693 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:10:59,693 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:10:59,694 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:10:59,694 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:10:59,695 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:10:59,698 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:10:59,700 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:10:59,726 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=0, n=1

2022-01-10 21:10:59,981 - climada.entity.exposures.litpop.litpop - INFO -
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:10:59,982 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2040. Using nearest available year for GPW data: 2020
2022-01-10 21:10:59,983 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:11:01,391 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:11:01,816 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:11:01,819 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:11:01,829 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:11:01,829 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:11:01,829 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:11:01,830 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:11:01,830 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:11:01,833 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:11:01,835 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:11:01,861 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.

 Computing litpop for m=1, n=1

2022-01-10 21:11:02,111 - climada.entity.exposures.litpop.litpop - INFO -
```

(continues on next page)

(continued from previous page)

```
 LitPop: Init Exposure for country: CUB (192)...

2022-01-10 21:11:02,113 - climada.entity.exposures.litpop.gpw_population - WARNING -␣
↪Reference year: 2040. Using nearest available year for GPW data: 2020
2022-01-10 21:11:02,113 - climada.entity.exposures.litpop.gpw_population - INFO - GPW␣
↪Version v4.11
2022-01-10 21:11:03,498 - climada.util.finance - WARNING - No data available for country.
↪ Using non-financial wealth instead
2022-01-10 21:11:03,929 - climada.util.finance - INFO - GDP CUB 2020: 1.074e+11.
2022-01-10 21:11:03,933 - climada.util.finance - WARNING - No data for country, using␣
↪mean factor.
2022-01-10 21:11:03,944 - climada.entity.exposures.base - INFO - Hazard type not set in␣
↪impf_
2022-01-10 21:11:03,945 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:11:03,945 - climada.entity.exposures.base - INFO - cover not set.
2022-01-10 21:11:03,945 - climada.entity.exposures.base - INFO - deductible not set.
2022-01-10 21:11:03,946 - climada.entity.exposures.base - INFO - centr_ not set.
2022-01-10 21:11:03,950 - climada.entity.exposures.base - INFO - Matching 1388 exposures␣
↪with 2500 centroids.
2022-01-10 21:11:03,951 - climada.util.coordinates - INFO - No exact centroid match␣
↪found. Reprojecting coordinates to nearest neighbor closer than the threshold = 100
2022-01-10 21:11:03,979 - climada.util.interpolation - WARNING - Distance to closest␣
↪centroid is greater than 100km for 77 coordinates.
```

[42]:
```
from climada.entity import Entity
from climada.util.constants import ENT_DEMO_FUTURE

ent_fut = Entity.from_excel(ENT_DEMO_FUTURE)
ent_fut.exposures.ref_year = 2040
ent_fut.check()
```

```
2022-01-10 21:11:04,102 - climada.entity.exposures.base - INFO - category_id not set.
2022-01-10 21:11:04,103 - climada.entity.exposures.base - INFO - geometry not set.
2022-01-10 21:11:04,103 - climada.entity.exposures.base - INFO - region_id not set.
2022-01-10 21:11:04,104 - climada.entity.exposures.base - INFO - centr_ not set.
```

[43]:
```
from climada.engine.unsequa import InputVar
entfut_iv = InputVar.entfut(
    impf_set = ent_fut.impact_funcs,
    exp_list = litpop_list,
    meas_set = ent_fut.measures,
    bounds_cost=[0.6, 1.2],
    bounds_eg=[0.8, 1.5],
    bounds_noise=None,
    bounds_mdd=[0.7, 0.9],
    bounds_paa=[1.3, 2],
    haz_id_dict={'TC': [1]}
    )
```

5.15 Forecast class

This class deals with weather forecasts and uses CLIMADA Impact.calc() to forecast impacts of weather events on society. It mainly does one thing: - it contains all plotting and other functionality that are specific for weather forecasts, impact forecasts and warnings

The class is different from the Impact class especially because features of the Impact class like Exceedence frequency curves, annual average impact etc, do not make sense if the hazard is e.g. a 5 day weather forecast. As the class is relatively new, there might be future changes to the datastructure, the methods, and the parameters used to call the methods.

5.15.1 Example: forecast of building damages due to wind in Switzerland

Before using the forecast class, hazard, exposure and vulnerability need to be created. The hazard looks at the weather forecast from today for an event with two days lead time (meaning the day after tomorrow). generate_WS_forecast_hazard is used to download a current weather forecast for wind gust from opendata.dwd.de. An Impact funtion for building damages due to storms is created. And with only a few lines of code, a LitPop exposure for Switzerland is generated, and the impact is calculated with a default impact function. With a further line of code, the mean damage per grid point for the day after tomorrow is plotted on a map.

[2]:
```
from datetime import datetime
from cartopy import crs as ccrs

from climada.util.config import CONFIG
from climada.engine.forecast import Forecast
from climada.hazard.storm_europe import StormEurope, generate_WS_forecast_hazard
from climada.entity.impact_funcs.storm_europe import ImpfStormEurope
from climada.entity import ImpactFuncSet
from climada.entity import LitPop
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-2-8371ba553ad9> in <module>
      2 from cartopy import crs as ccrs
      3
----> 4 from climada.util.config import CONFIG
      5 from climada.engine.forecast import Forecast
      6 from climada.hazard.storm_europe import StormEurope, generate_WS_forecast_hazard

C:\shortpaths\GitHub\climada_python\climada\__init__.py in <module>
     22 from pathlib import Path
     23
---> 24 from .util.config import CONFIG, setup_logging
     25 from .util.constants import *
     26

C:\shortpaths\GitHub\climada_python\climada\util\__init__.py in <module>
     20 """
     21 import logging
---> 22 from pint import UnitRegistry
     23
     24 from .config import *
```

(continues on next page)

(continued from previous page)

```
~\anaconda3\envs\climada_env_20210809\lib\site-packages\pint\__init__.py in <module>
     12 """
     13
---> 14 from .context import Context
     15 from .errors import (  # noqa: F401
     16     DefinitionSyntaxError,

~\anaconda3\envs\climada_env_20210809\lib\site-packages\pint\context.py in <module>
     13 from collections import ChainMap, defaultdict
     14
---> 15 from .definitions import Definition, UnitDefinition
     16 from .errors import DefinitionSyntaxError
     17 from .util import ParserHelper, SourceIterator, to_units_container

~\anaconda3\envs\climada_env_20210809\lib\site-packages\pint\definitions.py in <module>
     11 from collections import namedtuple
     12
---> 13 from .converters import LogarithmicConverter, OffsetConverter, ScaleConverter
     14 from .errors import DefinitionSyntaxError
     15 from .util import ParserHelper, UnitsContainer, _is_dim

~\anaconda3\envs\climada_env_20210809\lib\site-packages\pint\converters.py in <module>
     10
     11
---> 12 from .compat import HAS_NUMPY, exp, log  # noqa: F401
     13
     14

~\anaconda3\envs\climada_env_20210809\lib\site-packages\pint\compat.py in <module>
    161 # xarray (DataArray, Dataset, Variable)
    162 try:
--> 163     from xarray import DataArray, Dataset, Variable
    164
    165     upcast_types += [DataArray, Dataset, Variable]

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\__init__.py in <module>
      1 import pkg_resources
      2
----> 3 from . import testing, tutorial, ufuncs
      4 from .backends.api import (
      5     load_dataarray,

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\testing.py in <module>
      6 import numpy as np
      7
----> 8 from xarray.core import duck_array_ops, formatting, utils
      9 from xarray.core.dataarray import DataArray
     10 from xarray.core.dataset import Dataset

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\core\duck_array_ops.py in
↪<module>
     13 import pandas as pd
```

(continues on next page)

(continued from previous page)

```
     14
---> 15 from . import dask_array_compat, dask_array_ops, dtypes, npcompat, nputils
     16 from .nputils import nanfirst, nanlast
     17 from .pycompat import (

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\core\dask_array_compat.py␣
↪in <module>
      3 import numpy as np
      4
----> 5 from .pycompat import dask_version
      6
      7 try:

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\core\pycompat.py in
↪<module>
     53
     54
---> 55 dsk = DuckArrayModule("dask")
     56 dask_version = dsk.version
     57 dask_array_type = dsk.type

~\anaconda3\envs\climada_env_20210809\lib\site-packages\xarray\core\pycompat.py in __
↪init__(self, mod)
     23
     24             if mod == "dask":
---> 25                 duck_array_type = (import_module("dask.array").Array,)
     26             elif mod == "pint":
     27                 duck_array_type = (duck_array_module.Quantity,)

~\anaconda3\envs\climada_env_20210809\lib\importlib\__init__.py in import_module(name,␣
↪package)
    125                 break
    126             level += 1
--> 127     return _bootstrap._gcd_import(name[level:], package, level)
    128
    129

~\anaconda3\envs\climada_env_20210809\lib\site-packages\dask\array\__init__.py in
↪<module>
      1 try:
      2     from ..base import compute
----> 3     from . import backends, fft, lib, linalg, ma, overlap, random
      4     from .blockwise import atop, blockwise
      5     from .chunk_types import register_chunk_type

~\anaconda3\envs\climada_env_20210809\lib\site-packages\dask\array\fft.py in <module>
      5
      6 try:
----> 7     import scipy
      8     import scipy.fftpack
      9 except ImportError:
```

(continues on next page)

(continued from previous page)

```
~\anaconda3\envs\climada_env_20210809\lib\site-packages\scipy\__init__.py in <module>
    134
    135     # Allow distributors to run custom init code
--> 136     from . import _distributor_init
    137
    138     from scipy._lib import _pep440

~\anaconda3\envs\climada_env_20210809\lib\site-packages\scipy\_distributor_init.py in
↪<module>
     57             os.chdir(libs_path)
     58             for filename in glob.glob(os.path.join(libs_path, '*dll')):
---> 59                 WinDLL(os.path.abspath(filename))
     60         finally:
     61             os.chdir(owd)

~\anaconda3\envs\climada_env_20210809\lib\ctypes\__init__.py in __init__(self, name,␣
↪mode, handle, use_errno, use_last_error, winmode)
    371
    372         if handle is None:
--> 373             self._handle = _dlopen(self._name, mode)
    374         else:
    375             self._handle = handle

FileNotFoundError: Could not find module 'C:\Users\ThomasRoosli\anaconda3\envs\climada_
↪env_20210809\lib\site-packages\scipy\.libs\libbanded5x.
↪UGR6EUQPIWHQH7SL62IWIXB5545VDNQZ.gfortran-win_amd64.dll' (or one of its dependencies).␣
↪Try using the full path with constructor syntax.
```

[ ]:
```
#generate hazard
hazard, haz_model, run_datetime, event_date = generate_WS_forecast_hazard()
# #generate hazard with with forecasts from past dates (works only if the files have␣
↪already been downloaded)
# hazard, haz_model, run_datetime, event_date = generate_WS_forecast_hazard(
#     run_datetime=datetime(2021,3,7),
#     event_date=datetime(2021,3,11))
```

[ ]:
```
#generate vulnerability
impact_function = ImpfStormEurope.from_welker()
impact_function_set = ImpactFuncSet()
impact_function_set.append(impact_function)
```

[ ]:
```
#generate exposure and save to file
filename_exp = CONFIG.local_data.save_dir.dir() / ('exp_litpop_Switzerland.hdf5')
if filename_exp.exists():
    exposure = LitPop.from_hdf5(filename_exp)
else:
    exposure = LitPop.from_countries('Switzerland', reference_year=2020)
    exposure.write_hdf5(filename_exp)
```

[ ]:
```
#create and calculate Forecast
CH_WS_forecast = Forecast({run_datetime: hazard}, exposure, impact_function_set)
```

(continues on next page)

(continued from previous page)

```
CH_WS_forecast.calc()
```

[ ]:
```
CH_WS_forecast.plot_imp_map(save_fig=False,close_fig=False,proj=ccrs.epsg(2056))
```

Here you see a different plot highlighting the spread of the impact forecast calculated from the different ensemble members of the weather forecast.

[ ]:
```
CH_WS_forecast.plot_hist(save_fig=False,close_fig=False)
```

It is possible to color the pixels depending on the probability that a certain threshold of impact is reach at a certain grid point

[ ]:
```
CH_WS_forecast.plot_exceedence_prob(threshold=5000, save_fig=False, close_fig=False,
↪proj=ccrs.epsg(2056))
```

It is possible to color the cantons of Switzerland with warning colors, based on aggregated forecasted impacts in their area.

[ ]:
```
import fiona
from cartopy.io import shapereader
from climada.util.config import CONFIG


#create a file containing the polygons of Swiss cantons using natural earth
cantons_file = CONFIG.local_data.save_dir.dir() / 'cantons.shp'
adm1_shape_file = shapereader.natural_earth(resolution='10m',
                                    category='cultural',
                                    name='admin_1_states_provinces')
if not cantons_file.exists():
    with fiona.open(adm1_shape_file, 'r') as source:
        with fiona.open(
                cantons_file, 'w',
                **source.meta) as sink:

            for f in source:
                if f['properties']['adm0_a3'] == 'CHE':
                    sink.write(f)
CH_WS_forecast.plot_warn_map(str(cantons_file),
                             decision_level = 'polygon',
                             thresholds=[100000,500000,
                                         1000000,5000000],
                             probability_aggregation='mean',
                             area_aggregation='sum',
                             title="Building damage warning",
                             explain_text="warn level based on aggregated damages",
                             save_fig=False,
                             close_fig=False,
                             proj=ccrs.epsg(2056))
```

5.15.2 Example 2: forecast of wind warnings in Switzerland

Instead of a fully fledged socio-economic impact of storms, one can also simplify the hazard, exposure, vulnerability model, by looking at a "neutral" exposure (=1 at every gridpoint) and using a step function as impact function to arrive at warn levels. It also shows how the attributes hazard, exposure or vulnerability can be set before calling calc(), and are then considered in the forecast instead of the defined defaults.

[ ]:
```
from pandas import DataFrame
import numpy as np
from climada.entity.exposures import Exposures
from climada.entity.impact_funcs import ImpactFunc, ImpactFuncSet
import climada.util.plot as u_plot

### generate exposure
# find out which hazard coord to consider
CHE_borders = u_plot._get_borders(np.stack([exposure.gdf.latitude.values,
                                            exposure.gdf.longitude.values],
                                           axis=1)
                                  )
centroid_selection = np.logical_and(np.logical_and(hazard.centroids.lat >= CHE_
↪borders[2],
                                                   hazard.centroids.lat <= CHE_
↪borders[3]),
                                    np.logical_and(hazard.centroids.lon >= CHE_
↪borders[0],
                                                   hazard.centroids.lon <= CHE_
↪borders[1])
                                   )
# Fill DataFrame with values for a "neutral" exposure (value = 1)

exp_df = DataFrame()
exp_df['value'] = np.ones_like(hazard.centroids.lat[centroid_selection]) # provide value
exp_df['latitude'] = hazard.centroids.lat[centroid_selection]
exp_df['longitude'] = hazard.centroids.lon[centroid_selection]
exp_df['impf_WS'] = np.ones_like(hazard.centroids.lat[centroid_selection], int)
# Generate Exposures
exp = Exposures(exp_df)
exp.check()
exp.value_unit = 'warn_level'

### generate impact functions
## impact functions for hazard based warnings
imp_fun_low = ImpactFunc()
imp_fun_low.haz_type = 'WS'
imp_fun_low.id = 1
imp_fun_low.name = 'warn_level_low_elevation'
imp_fun_low.intensity_unit = 'm/s'
imp_fun_low.intensity = np.array([0.0, 19.439,
                                  19.44, 24.999,
                                  25.0, 30.549,
                                  30.55, 38.879,
                                  38.88, 100.0])
imp_fun_low.mdd = np.array([1.0, 1.0,
```

(continues on next page)

(continued from previous page)

```
                            2.0, 2.0,
                            3.0, 3.0,
                            4.0, 4.0,
                            5.0, 5.0])
imp_fun_low.paa = np.ones_like(imp_fun_low.mdd)
imp_fun_low.check()
# fill ImpactFuncSet
impf_set = ImpactFuncSet()
impf_set.append(imp_fun_low)
```

[ ]:
```
#create and calculate Forecast
warn_forecast = Forecast({run_datetime: hazard}, exp, impf_set)
warn_forecast.calc()
```

The each grid point now has a warnlevel between 1-5 assigned for each event. Now the cantons can be colored based on a threshold on a grid point level. for each warning level it is assessed if 50% of grid points in the area of a canton has at least a 50% probability of reaching the specified threshold.

[ ]:
```
warn_forecast.plot_warn_map(cantons_file,
                            thresholds=[2,3,4,5],
                            decision_level = 'exposure_point',
                            probability_aggregation=0.5,
                            area_aggregation=0.5,
                            title="DWD ICON METEOROLOGICAL WARNING",
                            explain_text="warn level based on wind gust thresholds",
                            save_fig=False,
                            close_fig=False,
                            proj=ccrs.epsg(2056))
```

5.15.3 Example: Tropical Cylcone

It would be nice to add an example using the tropical cyclone forecasts from the class TCForecast. This has not yet been done.

[ ]:

[ ]:

[ ]:

5.16 Calculate probabilistic impact yearset

This module generates a yearly impact `yimp` object which contains probabilistic annual impacts for a specified amount of years (`sampled_years`). The impact values are extracted from a given impact `imp` object that contains impact values per event. The amount of `sampled_years` can be specified as an integer or as a list of years to be sampled for. The amount of events per sampled year (`events_per_year`) are determined with a Poisson distribution centered around n_events per year (`lam` = sum(event_impacts.frequency). Then, the probabilistic events occuring in each sampled year are sampled uniformaly from the input `imp` object and summed up per year. Thus, the `yimp` object contains the sum of sampled (event) impacts for each sampled year. In contrast to the expected annual impact (eai), an `yimp` object

contains an impact for EACH sampled year and this value differs among years. The number of events_per_year and the selected_events are saved in a sampling vector (`sampling_vect`).

The function impact_yearsets performs all these computational steps, taking an `imp` and the number of sampled_years (`sampled_years`) as input. The output of the function is the `yimp` object and the `sampling_vect`. Moreover, a `sampling_vect` (generated in a previous run) can be provided as optional input and the user can define `lam` and decide whether a correction factor shall be applied (the default is applying the correction factor). Reapplying the same sampling_vect does not only allow to reproduce the generated `yimp`, but also for a physically consistent way of sampling impacts caused by different hazards. The correction factor that is applied when the optional input `correction_fac=` True is a scaling of the computed `yimp` that assures that the eai(`yimp`) = eai(`imp`).

To make the process more transparent, this tutorial shows the single computations that are performed when generating an `yimp` object for a dummy event_impacts object.

[1]:
```
import numpy as np

import climada.util.yearsets as yearsets
from climada.engine import Impact

# dummy event_impacts object containing 10 event_impacts with the values 10-110
# and the frequency 0.2 (Return period of 5 years)
imp = Impact()
imp.at_event = np.arange(10,110,10)
imp.frequency = np.array(np.ones(10)*0.2)

# the number of years to sample impacts for (length(yimp.at_event) = sampled_years)
sampled_years = 10

# sample number of events per sampled year
lam = np.sum(imp.frequency)
events_per_year = yearsets.sample_from_poisson(sampled_years, lam)
events_per_year
```

[1]:
```
array([2, 2, 2, 0, 4, 5, 4, 2, 3, 1])
```

[2]:
```
# generate the sampling vector
sampling_vect = yearsets.sample_events(events_per_year, imp.frequency)
sampling_vect
```

[2]:
```
[array([8, 3]),
 array([7, 0]),
 array([4, 6]),
 array([], dtype=int32),
 array([5, 9, 1, 2]),
 array([1, 6, 0, 7, 2]),
 array([4, 9, 5, 8]),
 array([9, 8]),
 array([5, 3, 4]),
 array([1])]
```

[3]:
```
# calculate the impact per year
imp_per_year = yearsets.compute_imp_per_year(imp, sampling_vect)
imp_per_year
```

[3]:
```
[130, 90, 120, 0, 210, 210, 300, 190, 150, 20]
```

[4]:
```
# calculate the correction factor
correction_factor = yearsets.calculate_correction_fac(imp_per_year, imp)
correction_factor
```

[4]:
```
0.7746478873239436
```

[5]:
```
# compare the resulting yimp with our step-by-step computation without applying the␣
↪correction factor:
yimp, sampling_vect = yearsets.impact_yearset(imp, sampled_years=list(range(1,11)),␣
↪correction_fac=False)

print('The yimp.at_event values equal our step-by-step computed imp_per_year:')
print('yimp.at_event = ', yimp.at_event)
print('imp_per_year = ', imp_per_year)
```

```
The yimp.at_event values equal our step-by-step computed imp_per_year:
yimp.at_event =  [90, 240, 150, 70, 40, 90, 60, 90, 170, 110]
imp_per_year =  [130, 90, 120, 0, 210, 210, 300, 190, 150, 20]
```

[6]:
```
# and here the same comparison with applying the correction factor (default settings):
yimp, sampling_vect = yearsets.impact_yearset(imp, sampled_years=list(range(1,11)))

print('The same can be shown for the case of applying the correction factor.'
      'The yimp.at_event values equal our step-by-step computed imp_per year:')
print('yimp.at_event = ', yimp.at_event)
print('imp_per_year = ', imp_per_year/correction_factor)
```

```
The same can be shown for the case of applying the correction factor.The yimp.at_event␣
↪values equal our step-by-step computed imp_per year:
yimp.at_event =  [ 54.54545455  47.72727273  95.45454545 109.09090909   0.
  40.90909091  27.27272727   0.         109.09090909  27.27272727]
imp_per_year =  [167.81818182 116.18181818 154.90909091   0.         271.09090909
 271.09090909 387.27272727 245.27272727 193.63636364  25.81818182]
```

5.17 Data API

This tutorial is separated into three main parts: the first two parts shows how to find and get data to do impact calculations and should be enough for most users. The third part provides more detailed information on how the API is built.

5.17.1 Contents

- *Server*
- *Client*
- *Metadata*
- *Download*

5.17.2 Finding datasets

[1]:
```
from climada.util.api_client import Client
client = Client()
```

Data types and data type groups

The datasets are first separated into 'data_type_groups', which represent the main classes of CLIMADA (exposures, hazard, vulnerability, ...). So far, data is available for exposures and hazard. Then, data is separated into data_types, representing the different hazards and exposures available in CLIMADA

[2]:
```
import pandas as pd
data_types = client.list_data_type_infos()

dtf = pd.DataFrame(data_types)
dtf.sort_values(['data_type_group', 'data_type'])
```

[2]:
```
          data_type data_type_group  status description  \
3   crop_production       exposures  active        None
0            litpop       exposures  active        None
5         centroids          hazard  active        None
2       river_flood          hazard  active        None
4      storm_europe          hazard  active        None
1  tropical_cyclone          hazard  active        None

                                        properties
3  [{'property': 'crop', 'mandatory': True, 'desc...
0  [{'property': 'res_arcsec', 'mandatory': False...
5                                                 []
2  [{'property': 'res_arcsec', 'mandatory': False...
4  [{'property': 'country_iso3alpha', 'mandatory'...
1  [{'property': 'res_arcsec', 'mandatory': True,...
```

Datasets and Properties

For each data type, the single datasets can be differentiated based on properties. The following function provides a table listing the properties and possible values. This table does not provide information on properties that can be combined but the search can be refined in order to find properties to query a unique dataset. Note that a maximum of 10 property values are shown here, but many more countries are available for example.

[3]:
```
litpop_dataset_infos = client.list_dataset_infos(data_type='litpop')
```

[4]:
```
all_properties = client.get_property_values(litpop_dataset_infos)
```

[5]:
```
all_properties.keys()
```

[5]:
```
dict_keys(['res_arcsec', 'exponents', 'fin_mode', 'spatial_coverage', 'country_iso3alpha
↪', 'country_name', 'country_iso3num'])
```

Refining the search:

[6]:
```
# as datasets are usually available per country, chosing a country or global dataset␣
↪reduces the options
# here we want to see which datasets are available for litpop globally:
client.get_property_values(litpop_dataset_infos, known_property_values = {'spatial_
↪coverage':'global'})
```

[6]:
```
{'res_arcsec': ['150'],
 'exponents': ['(0,1)', '(1,1)', '(3,0)'],
 'fin_mode': ['pop', 'pc'],
 'spatial_coverage': ['global']}
```

[7]:
```
#and here for Switzerland:
client.get_property_values(litpop_dataset_infos, known_property_values = {'country_name':
↪'Switzerland'})
```

[7]:
```
{'res_arcsec': ['150'],
 'exponents': ['(3,0)', '(0,1)', '(1,1)'],
 'fin_mode': ['pc', 'pop'],
 'spatial_coverage': ['country'],
 'country_iso3alpha': ['CHE'],
 'country_name': ['Switzerland'],
 'country_iso3num': ['756']}
```

5.17.3 Basic impact calculation

We here show how to make a basic impact calculation with tropical cyclones for Haiti, for the year 2040, rcp4.5 and generated with 10 synthetic tracks. For more technical details on the API, see below.

Wrapper functions to open datasets as CLIMADA objects

The wrapper functions client.get_hazard()

gets the dataset information, downloads the data and opens it as a hazard instance

[8]:
```
tc_dataset_infos = client.list_dataset_infos(data_type='tropical_cyclone')
client.get_property_values(tc_dataset_infos, known_property_values = {'country_name':
↪'Haiti'})
```

[8]:
```
{'res_arcsec': ['150'],
 'climate_scenario': ['rcp26', 'rcp45', 'rcp85', 'historical', 'rcp60'],
 'ref_year': ['2040', '2060', '2080'],
 'nb_synth_tracks': ['50', '10'],
 'spatial_coverage': ['country'],
```

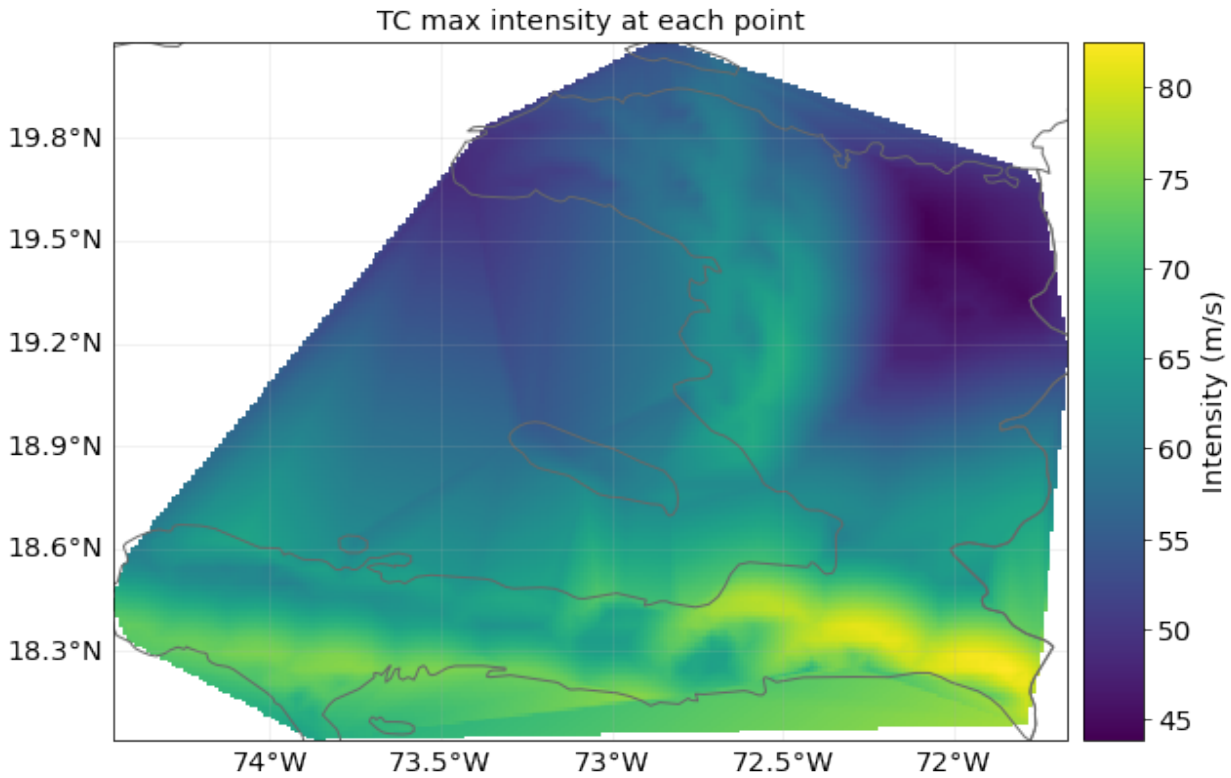
(continues on next page)

(continued from previous page)

```
 'tracks_year_range': ['1980_2020'],
 'country_iso3alpha': ['HTI'],
 'country_name': ['Haiti'],
 'country_iso3num': ['332'],
 'resolution': ['150 arcsec']}
```

[9]:
```
client = Client()
tc_haiti = client.get_hazard('tropical_cyclone', properties={'country_name': 'Haiti',
↪'climate_scenario': 'rcp45', 'ref_year':'2040', 'nb_synth_tracks':'10'})
tc_haiti.plot_intensity(0)
```

[9]:
```
<GeoAxesSubplot:title={'center':'TC max intensity at each point'}>
```



The wrapper functions client.get_litpop_default()

gets the default litpop, with exponents (1,1) and 'produced capital' as financial mode. If no country is given, the global dataset will be downloaded.

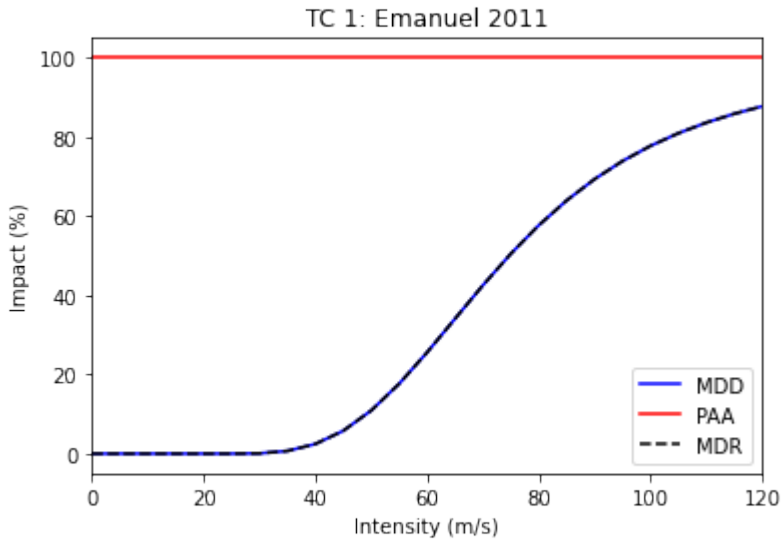
[10]:
```
litpop_default = client.get_property_values(litpop_dataset_infos, known_property_values␣
↪= {'fin_mode':'pc', 'exponents':'(1,1)'})
```

[11]:
```
litpop = client.get_litpop_default(country='Haiti')
```

Get the default impact function for tropical cyclones

[12]:
```
from climada.entity.impact_funcs import ImpactFuncSet, ImpfTropCyclone

imp_fun = ImpfTropCyclone.from_emanuel_usa()
imp_fun.check()
imp_fun.plot()

imp_fun_set = ImpactFuncSet()
imp_fun_set.append(imp_fun)

litpop.impact_funcs = imp_fun_set
```

```
2022-01-31 22:30:21,359 - climada.entity.impact_funcs.base - WARNING - For intensity = 0,
↪ mdd != 0 or paa != 0. Consider shifting the origin of the intensity scale. In impact.
↪calc the impact is always null at intensity = 0.
```



Calculate the impact

[13]:
```
from climada.engine import Impact
impact = Impact()
impact.calc(litpop, imp_fun_set, tc_haiti)
```

Getting other Exposures

[14]:
```
crop_dataset_infos = client.list_dataset_infos(data_type='crop_production')

client.get_property_values(crop_dataset_infos)
```

[14]:
```
{'crop': ['whe', 'soy', 'ric', 'mai'],
 'irrigation_status': ['noirr', 'firr'],
 'unit': ['USD', 'Tonnes'],
 'spatial_coverage': ['global']}
```

[15]:
```
rice_exposure = client.get_exposures(exposures_type='crop_production', properties = {
↪'crop':'ric', 'unit': 'USD','irrigation_status': 'noirr'})
```

5.17.4 Technical Information

For programmatical access to the CLIMADA data API there is a specific REST call wrapper class: `climada.util.client.Client`.

Server

The CLIMADA data file server is hosted on https://data.iac.ethz.ch that can be accessed via a REST API at https://climada.ethz.ch. For REST API details, see the documentation.

Client

[16]:
```
Client?
```

```
Init signature: Client()
Docstring:
Python wrapper around REST calls to the CLIMADA data API server.

Init docstring:
Constructor of Client.

Data API host and chunk_size (for download) are configurable values.
Default values are 'climada.ethz.ch' and 8096 respectively.
File:           c:\users\me\polybox\workshop\climada_python\climada\util\api_client.py
Type:           type
Subclasses:
```

[17]:
```
client = Client()
client.chunk_size
```

[17]:
```
8192
```

The url to the API server and the chunk size for the file download can be configured in 'climada.conf'. Just replace the corresponding default values:

```
"data_api": {
    "host": "https://climada.ethz.ch",
    "chunk_size": 8192,
    "cache_db": "{local_data.system}/.downloads.db"
}
```

The other configuration value affecting the data_api client, `cache_db`, is the path to an SQLite database file, which is keeping track of the files that are successfully downloaded from the api server. Before the Client attempts to download any file from the server, it checks whether the file has been downloaded before and if so, whether the previously downloaded file still looks good (i.e., size and time stamp are as expected). If all of this is the case, the file is simply read from disk without submitting another request.

Metadata

Unique Identifiers

Any dataset can be identified with **data_type**, **name** and **version**. The combination of the three is unique in the API servers' underlying database. However, sometimes the name is already enough for identification. All datasets have a UUID, a universally unique identifier, which is part of their individual url. E.g., the uuid of the dataset https://climada.ethz.ch/rest/dataset/b1c76120-4e60-4d8f-99c0-7e1e7b7860ec is "b1c76120-4e60-4d8f-99c0-7e1e7b7860ec". One can retrieve their meta data by:

[18]:
```
client.get_dataset_info_by_uuid('b1c76120-4e60-4d8f-99c0-7e1e7b7860ec')
```

[18]:
```
DatasetInfo(uuid='b1c76120-4e60-4d8f-99c0-7e1e7b7860ec', data_
↪type=DataTypeShortInfo(data_type='litpop', data_type_group='exposures'), name='LitPop_
↪assets_pc_150arcsec_SGS', version='v1', status='active', properties={'res_arcsec': '150
↪', 'exponents': '(3,0)', 'fin_mode': 'pc', 'spatial_coverage': 'country', 'date_
↪creation': '2021-09-23', 'climada_version': 'v2.2.0', 'country_iso3alpha': 'SGS',
↪'country_name': 'South Georgia and the South Sandwich Islands', 'country_iso3num': '239
↪'}, files=[FileInfo(uuid='b1c76120-4e60-4d8f-99c0-7e1e7b7860ec', url='https://data.iac.
↪ethz.ch/climada/b1c76120-4e60-4d8f-99c0-7e1e7b7860ec/LitPop_assets_pc_150arcsec_SGS.
↪hdf5', file_name='LitPop_assets_pc_150arcsec_SGS.hdf5', file_format='hdf5', file_
↪size=1086488, check_sum='md5:27bc1846362227350495e3d946dfad5e')], doi=None,␣
↪description="LitPop asset value exposure per country: Gridded physical asset values by␣
↪country, at a resolution of 150 arcsec. Values are total produced capital values␣
↪disaggregated proportionally to the cube of nightlight intensity (Lit^3, based on NASA␣
↪Earth at Night). The following values were used as parameters in the LitPop.from_
↪countries() method:{'total_values': 'None', 'admin1_calc': 'False','reference_year':
↪'2018', 'gpw_version': '4.11'}Reference: Eberenz et al., 2020. https://doi.org/10.5194/
↪essd-12-817-2020", license='Attribution 4.0 International (CC BY 4.0)', activation_
↪date='2021-09-13 09:08:28.358559+00:00', expiration_date=None)
```

or by filtering:

Data Set Status

The datasets of climada.ethz.ch may have the following stati: - **active**: the deault for real life data - **preliminary**: when the dataset is already uploaded but some information or file is still missing - **expired**: when a dataset is inactivated again - **test_dataset**: data sets that are used in unit or integration tests have this status in order to be taken seriously by accident When collecting a list of datasets with `get_datasets`, the default dataset status will be 'active'. With the argument `status=None` this filter can be turned off.

DatasetInfo Objects and DataFrames

As stated above `get_dataset` (or `get_dataset_by_uuid`) return a `DatasetInfo` object and `get_datasets` a list thereof.

[19]:
```
from climada.util.api_client import DatasetInfo
DatasetInfo?
```

```
Init signature:
DatasetInfo(
    uuid: str,
    data_type: climada.util.api_client.DataTypeShortInfo,
    name: str,
    version: str,
    status: str,
    properties: dict,
    files: list,
    doi: str,
    description: str,
```

(continues on next page)

(continued from previous page)

```
Docstring:      dataset data from CLIMADA data API.
File:           c:\users\me\polybox\workshop\climada_python\climada\util\api_client.py
Type:           type
Subclasses:
```

where files is a list of `FileInfo` objects:

[20]:
```
from climada.util.api_client import FileInfo
FileInfo?
```

```
Init signature:
FileInfo(
    uuid: str,
    url: str,
    file_name: str,
    file_format: str,
    file_size: int,
    check_sum: str,
) -> None
Docstring:      file data from CLIMADA data API.
File:           c:\users\me\polybox\workshop\climada_python\climada\util\api_client.py
Type:           type
Subclasses:
```

Convert into DataFrame

There are conveinience functions to easily convert datasets into pandas DataFrames, `get_datasets` and `expand_files`:

[21]:
```
client.into_datasets_df?
```

```
Signature: client.into_datasets_df(dataset_infos)
Docstring:
Convenience function providing a DataFrame of datasets with properties.

Parameters
----------
dataset_infos : list of DatasetInfo
     as returned by list_dataset_infos

Returns
-------
pandas.DataFrame
    of datasets with properties as found in query by arguments
File:      c:\users\me\polybox\workshop\climada_python\climada\util\api_client.py
Type:      function
```

[22]:
```
from climada.util.api_client import Client
client = Client()
```

(continues on next page)

(continued from previous page)

```
litpop_datasets = client.list_dataset_infos(data_type='litpop', properties={'country_name
↪': 'South Georgia and the South Sandwich Islands'})
litpop_df = client.into_datasets_df(litpop_datasets)
litpop_df
```

[22]:

```
  data_type data_type_group                                  uuid  \
0    litpop       exposures  b1c76120-4e60-4d8f-99c0-7e1e7b7860ec
1    litpop       exposures  3d516897-5f87-46e6-b673-9e6c00d110ec
2    litpop       exposures  a6864a65-36a2-4701-91bc-81b1355103b5

                            name version  status   doi  \
0  LitPop_assets_pc_150arcsec_SGS      v1  active  None
1        LitPop_pop_150arcsec_SGS      v1  active  None
2            LitPop_150arcsec_SGS      v1  active  None

                                       description  \
0  LitPop asset value exposure per country: Gridd...
1  LitPop population exposure per country: Gridde...
2  LitPop asset value exposure per country: Gridd...

                                    license  \
0  Attribution 4.0 International (CC BY 4.0)
1  Attribution 4.0 International (CC BY 4.0)
2  Attribution 4.0 International (CC BY 4.0)

                   activation_date expiration_date res_arcsec exponents  \
0  2021-09-13 09:08:28.358559+00:00            None        150     (3,0)
1  2021-09-13 09:09:10.634374+00:00            None        150     (0,1)
2  2021-09-13 09:09:30.907938+00:00            None        150     (1,1)

  fin_mode spatial_coverage date_creation climada_version country_iso3alpha  \
0       pc          country    2021-09-23          v2.2.0               SGS
1      pop          country    2021-09-23          v2.2.0               SGS
2       pc          country    2021-09-23          v2.2.0               SGS

                                  country_name country_iso3num
0  South Georgia and the South Sandwich Islands            239
1  South Georgia and the South Sandwich Islands            239
2  South Georgia and the South Sandwich Islands            239
```

Download

The wrapper functions get_exposures or get_hazard fetch the information, download the file and opens the file as a climada object. But one can also just download dataset files using the method `download_dataset` which takes a `DatasetInfo` object as argument and downloads all files of the dataset to a directory in the local file system.

[23]:

```
client.download_dataset?
```

```
Signature:
client.download_dataset(
    dataset,
    target_dir=WindowsPath('C:/Users/me/climada/data'),
    organize_path=True,
)
```

(continues on next page)

(continued from previous page)

```
Docstring:
Download all files from a given dataset to a given directory.

Parameters
----------
dataset : DatasetInfo
    the dataset
target_dir : Path, optional
    target directory for download, by default `climada.util.constants.SYSTEM_DIR`
organize_path: bool, optional
    if set to True the files will end up in subdirectories of target_dir:
    [target_dir]/[data_type_group]/[data_type]/[name]/[version]
    by default True

Returns
-------
download_dir : Path
    the path to the directory containing the downloaded files,
    will be created if organize_path is True
downloaded_files : list of Path
    the downloaded files themselves

Raises
------
Exception
    when one of the files cannot be downloaded
File:      c:\users\me\polybox\workshop\climada_python\climada\util\api_client.py
Type:      method
```

Cache

The method avoids superfluous downloads by keeping track of all downloads in a sqlite db file. The client will make sure that the same file is never downloaded to the same target twice.

Examples

[24]:
```
ds = litpop_datasets[0]
download_dir, ds_files = client.download_dataset(ds)
ds_files[0], ds_files[0].is_file()
```

[24]:
```
(WindowsPath('C:/Users/me/climada/data/exposures/litpop/LitPop_assets_pc_150arcsec_SGS/
↪v1/LitPop_assets_pc_150arcsec_SGS.hdf5'),
 True)
```

CHAPTER

SIX

DEVELOPER GUIDE

6.1 Development and Git and CLIMADA

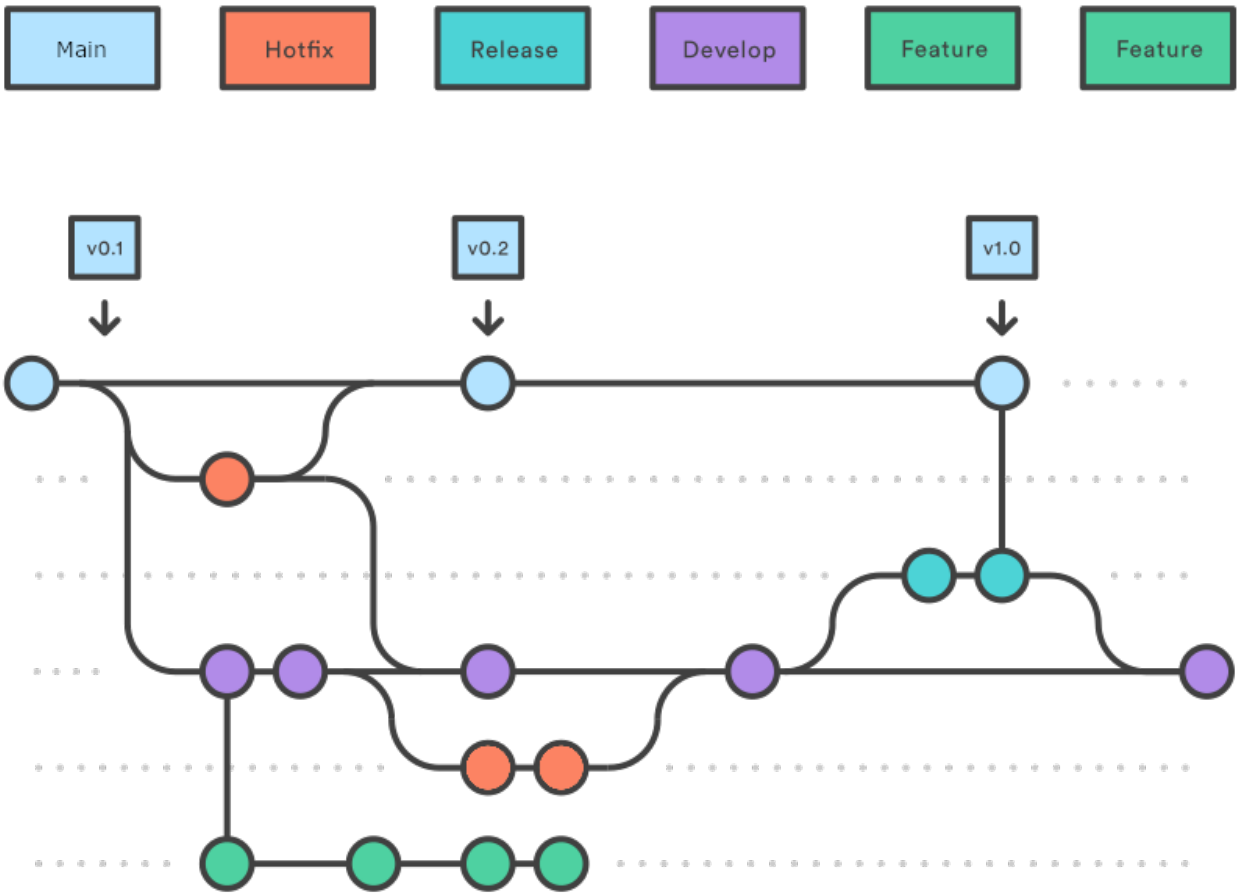
Chris Fairless

Table of Contents

Introduction

6.1.1 Git and GitHub

- Git's not that scary
  - 95% of your work on Git will be done with the same handful of commands
  - (the other 5% will always be done with careful Googling)
  - Almost everything in Git can be undone by design (but use `rebase`, `--force` and `--hard` with care!)
  - Your favourite IDE (Spyder, PyCharm, ...) will have a GUI for working with Git, or you can download a standalone one.
- The Git Book is a great introduction to how Git works and to using it on the command line.
- Consider using a GUI program such as "git desktop" or "Gitkraken" to have a visual git interface, in particular at the beginning. Your python IDE is also likely to have a visual git interface.
- Feel free to ask for help



What I assume you know

I'm assuming you're all familiar with the basics of Git.

- What (and why) is version control
- How to clone a repository
- How to make a commit and push it to GitHub
- What a branch is, and how to make one
- How to merge two branches
- The basics of the GitHub website

If you're not feeling great about this, I recommend - sending me a message so we can arrange an introduction with CLIMADA - exploring the Git Book

Terms we'll be using today

These are terms I'll be using a lot today, so let's make sure we know them

- local versus remote
  - Our **remote** repository is hosted on GitHub. This is the central location where all updates to CLIMADA that we want to share end up. If you're updating CLIMADA for the community, your code will end up here too.
  - Your **local** repository is the copy you have on the machine you're working on, and where you do your work.
  - Git calls the (first, default) remote the `origin`
  - (It's possible to set more than one remote repository, e.g. you might set one up on a network-restricted computing cluster)
- push, pull and pull request
  - You **push** your work when you send it from your local machine to the remote repository
  - You **pull** from the remote repository to update the code on your local machine
  - A **pull request** is a standardised review process on GitHub. Usually it ends with one branch merging into another
- Conflict resolution
  - Sometimes two people have made changes to the same bit of code. Usually this comes up when you're trying to merge branches. The changes have to be manually compared and the code edited to make sure the 'correct' version of the code is kept.

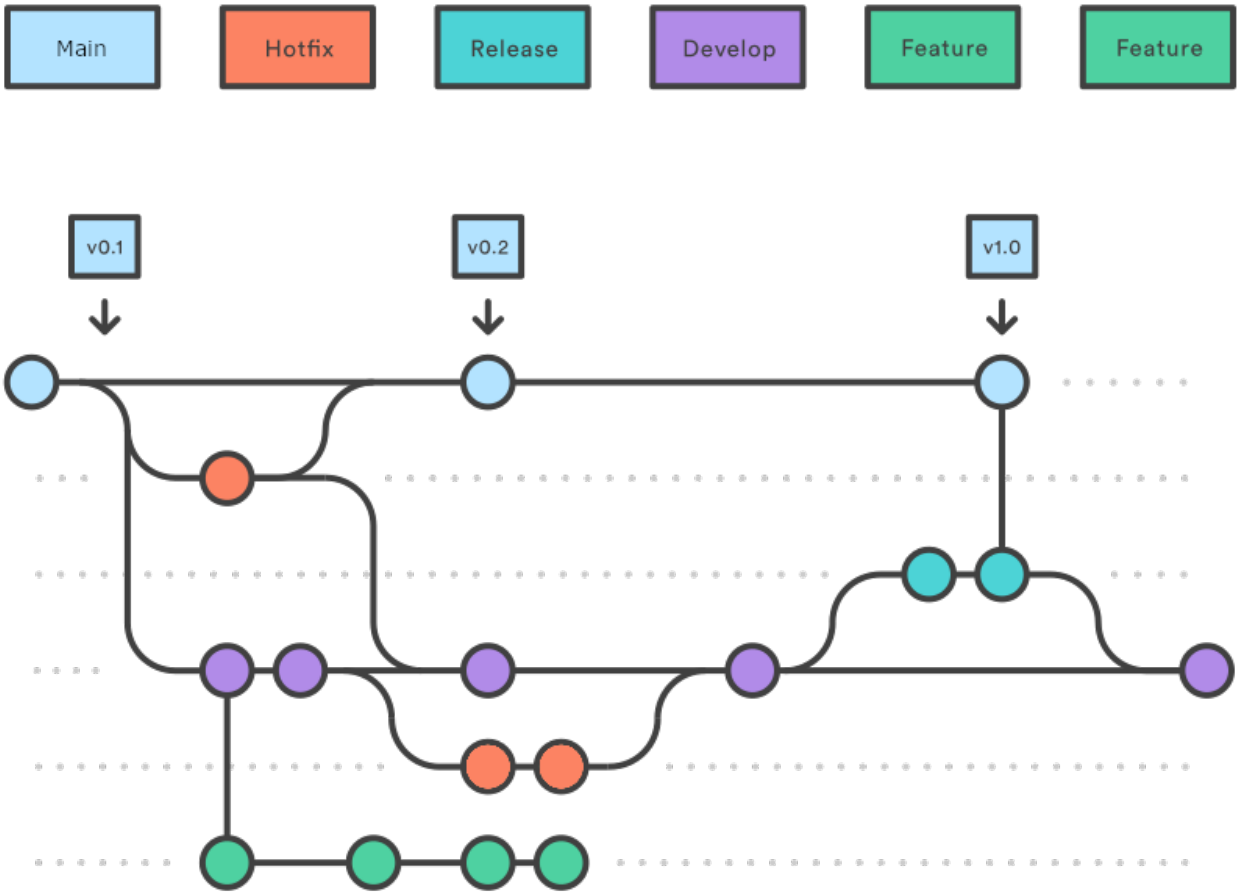
Gitflow

Gitflow is a particular way of using git to organise projects that have - multiple developers - working on different features - with a release cycle

It means that - there's always a stable version of the code available to the public - the chances of two developers' code conflicting are reduced - the process of adding and reviewing features and fixes is more standardised for everyone

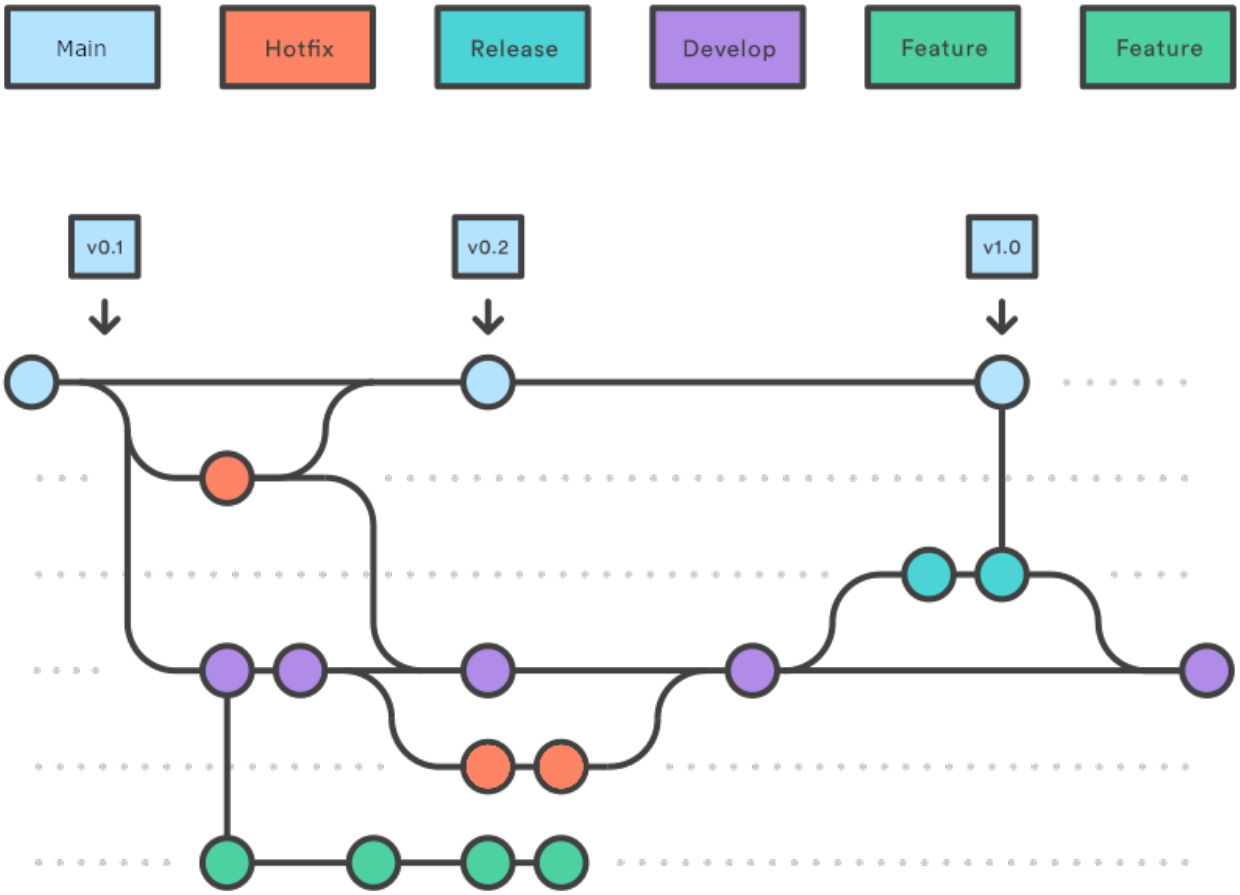
Gitflow is a *convention*, so you don't need any additional software. - ... but if you want you can get some: a popular extension to the git command line tool allows you to issue more intuitive commands for a Gitflow workflow. - Mac/Linux users can install git-flow from their package manager, and it's included with Git for Windows

Gitflow works on the `develop` branch instead of `main`



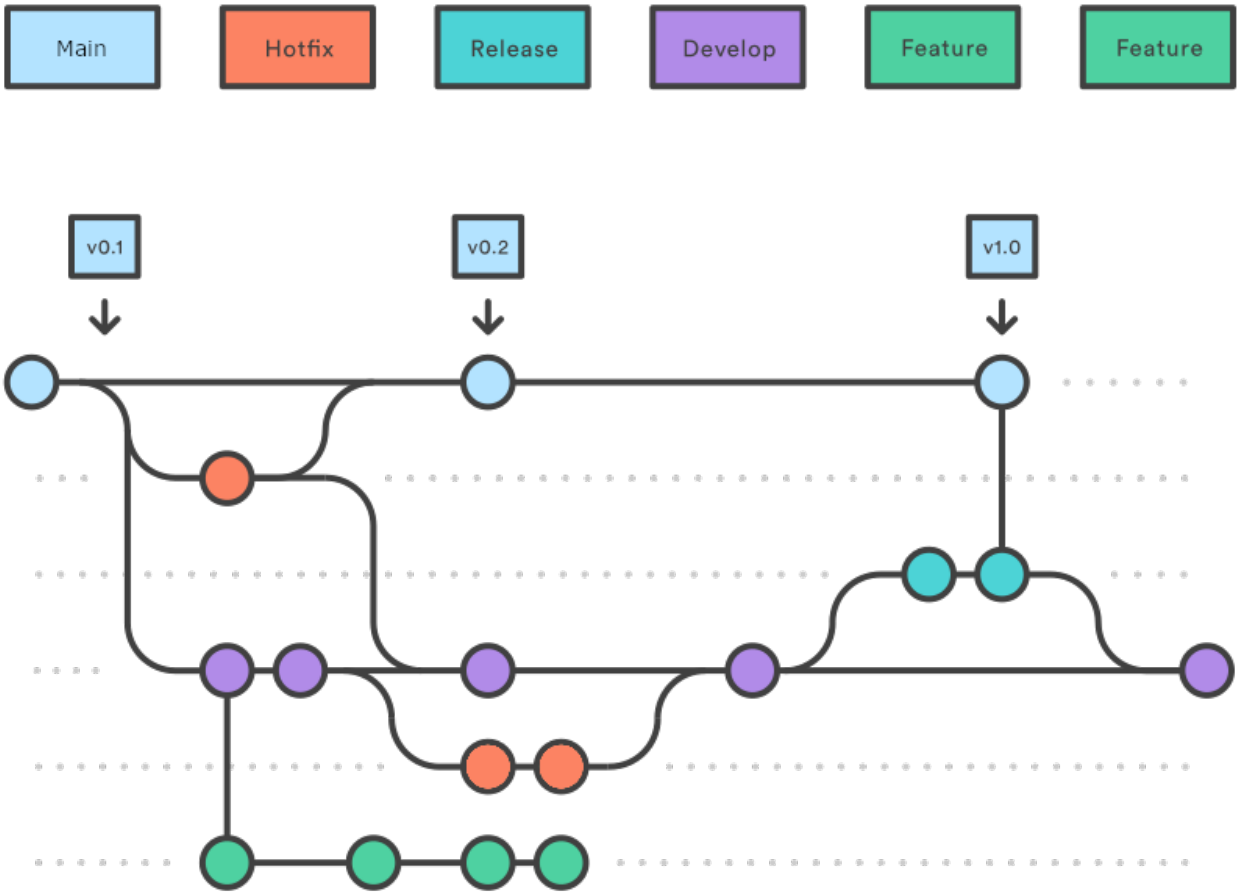
- The critical difference between Gitflow and 'standard' git is that almost all of your work takes place on the `develop` branch, instead of the `main` (formerly `master`) branch.
- The `main` branch is reserved for planned, stable product releases, and it's what the general public download when they install CLIMADA. The developers almost never interact with it.

Gitflow is a feature-based workflow



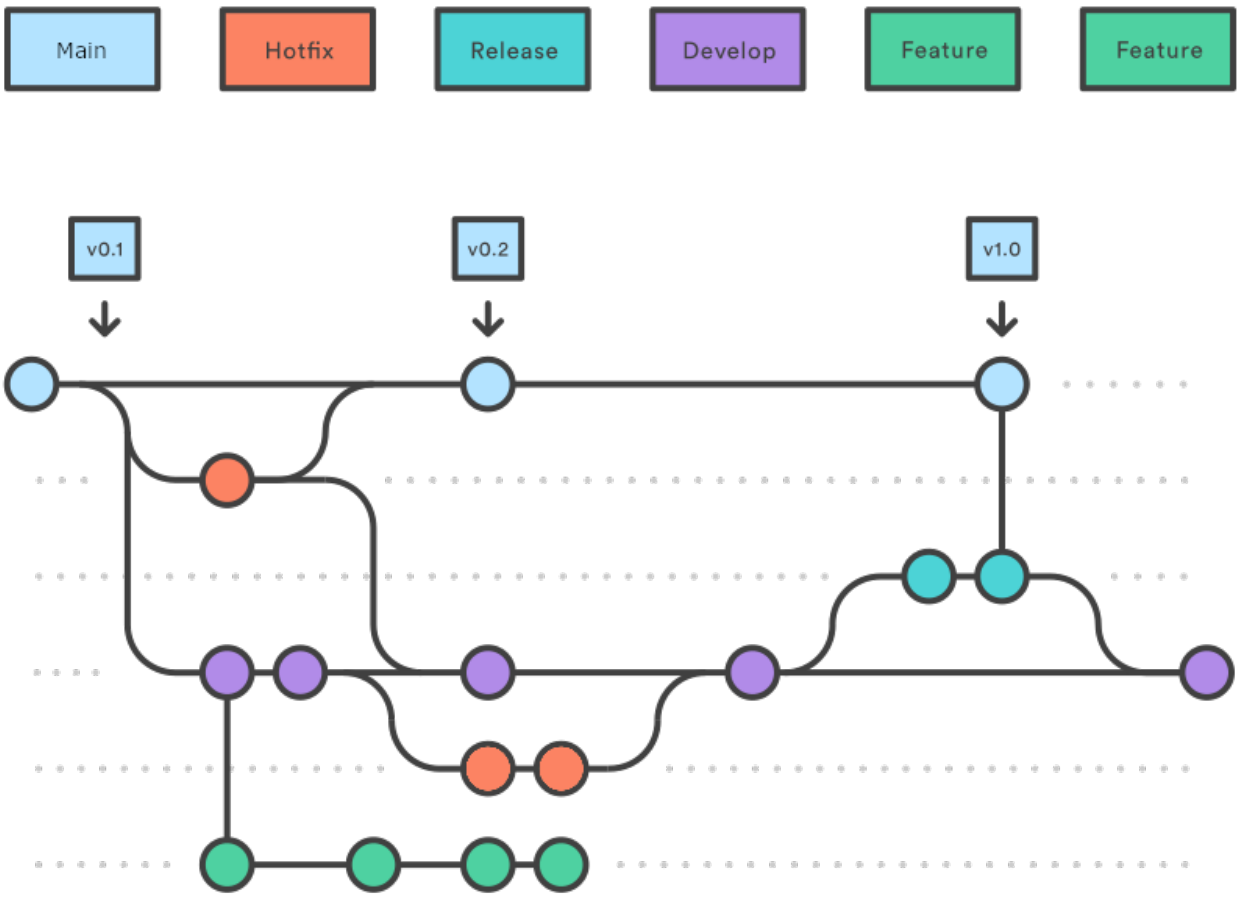
- This is common to many workflows: when you want to add something new to the model you start a new branch, work on it locally, and then merge it back into `develop` **with a pull request** (which we'll cover later).
- By convention we name all CLIMADA feature branches `feature/*` (e.g. `feature/meteorite`).
- Features can be anything, from entire hazard modules to a smarter way to do one line of a calculation. Most of the work you'll do on CLIMADA will be a features of one size or another.
- We'll talk more about developing CLIMADA features later!

Gitflow enables a regular release cycle



- A release is usually more complex than merging `develop` into `main`.
- So for this a `release-*` branch is created from `develop`. We'll all be notified repeatedly when the deadline is to submit (and then to review) pull requests so that you can be included in a release.
- The core developer team (mostly Emanuel) will then make sure tests, bugfixes, documentation and compatibility requirements are met, merging any fixes back into `develop`.
- On release day, the release branch is merged into `main`, the commit is tagged as a release and the release notes are published on the GitHub at https://github.com/CLIMADA-project/climada_python/releases

Everything else is hotfixes



- The other type of branch you'll create is a hotfix.
- Hotfixes are generally small changes to code that do one thing, fixing typos, small bugs, or updating docstrings. They're done in much the same way as features, and are usually merged with a pull request.
- The difference between features and hotfixes is fuzzy and you don't need to worry about getting it right.
- Hotfixes will occasionally be used to fix bugs on the `main` branch, in which case they will merge into both `main` and `develop`.
- Some hotfixes are so simple - e.g. fixing a typo or a docstring - that they don't need a pull request. Use your judgement, but as a rule, if you change what the code does, or how, you should be merging with a pull request.

6.1.2 Installing CLIMADA for development

0. **Install** Git and Anaconda (or Miniconda).

   Also consider installing Git flow. This is included with Git for Windows and has different implementations e.g. here for Windows and Mac.

1. **Clone (or fork)** the project on GitHub

   From the location where you want to create the project folder, run in your terminal:

```
::
```

git clone https://github.com/CLIMADA-project/climada_python.git

2. **Install the packages** in `climada_python/requirements/env_climada.yml` and `climada_python/requirements/env_developer.yml` (see install). You might need to install additional environments contained in `climada_python/requirements` when using specific functionalities.

6.1.3 Features and branches

Planning a new feature

Here we're talking about large features such as new modules, new data sources, or big methodological changes. Any extension to CLIMADA that might affect other developers' work, modify the CLIMADA core, or need a big code review.

Smaller feature branches don't need such formalities. Use your judgment, and if in doubt, let people know.

Talk to the group

- Before starting coding a module, do not forget to coordinate with one of the repo admins (Emanuel, Chahan or David)
- This is the chance to work out the Big Picture stuff that is better when it's planned with the group - possible intersections with other projects, possible conflicts, changes to the CLIMADA core, additional dependencies (see Chahan's presentation later)
- Also talk with others from the core development team (see the GitHub wiki).
- Bring it to a developers meeting - people may be able to help/advise and are always interested in hearing about new projects. You can also find reviewers!
- Also, keep talking! Your plans *will* change :)

Planning the work

- Does the project go in its own repository and import CLIMADA, or does it extend the main CLIMADA repository?
  - The way this is done is slowly changing, so definitely discuss it with the group.
  - Chahan will discuss this later!
- Find a few people who will help to review your code.
  - Ask in a developers' meeting, on Slack (for WCR developers) or message people on the development team (see the GitHub wiki).
  - Let them know roughly how much code will be in the reviews, and when you'll be creating pull requests.
- How can the work split into manageable chunks?
  - A series of smaller pull requests is far more manageable than one big one (and takes off some of the pre-release pressure)
  - Reviewing and spotting issues/improvements/generalisations early is always a good thing.
  - It encourages modularisation of the code: smaller self-contained updates, with documentation and tests.
- Will there be any changes to the CLIMADA core?
  - These should be planned carefully

- Will you need any new dependencies? Are you sure?
  - Chahan will discuss this later!

Working on feature branches

When developing a big new feature, consider creating a feature branch and merging smaller branches into that feature branch with pull requests, keeping the whole process separate from `develop` until it's completed. This makes step-by-step code review nice and easy, and makes the final merge more easily tracked in the history.

e.g. developing the big `feature/meteorite` module you might write `feature/meteorite-hazard` and merge it in, then `feature/meteorite-impact`, then `feature/meteorite-stochastic-events` etc… before finally merging `feature/meteorite` into `develop`. Each of these could be a reviewable pull request.

Make a new branch

For new features in Git flow:

```
git flow feature start feature_name
```

Which is equivalent to (in vanilla git):

```
git checkout -b feature/feature_name
```

Or work on an existing branch:

```
git checkout -b branch_name
```

Follow the python do's and don't and performance guides. Write small readable methods, classes and functions.

get the latest data from the remote repository and update your branch

```
git pull
```

see your locally modified files

```
git status
```

add changes you want to include in the commit

```
git add climada/modified_file.py climada/test/test_modified_file.py
```

commit the changes

```
git commit -m "new functionality of .. implemented"
```

Make unit and integration tests on your code, preferably during development

see Guide on unit and integration tests

6.1.4 Pull requests

We want every line of code that goes into the CLIMADA repository to be reviewed!

Code review: - catches bugs (there are *always* bugs) - lets you draw on the experience of the rest of the team - makes sure that more than one person knows how your code works - helps to unify and standardise CLIMADA's code, so new users find it easier to read and navigate - creates an archived description and discussion of the changes you've made

When to make a pull request

- When you've finished writing a big new class or method (and its tests)
- When you've fixed a bug or made an improvement you want to merge
- When you want to merge a change of code into `develop` or `main`
- When you want to *discuss* a bit of code you've been working on - pull requests aren't only for merging branches

Not all pull requests have to be into `develop` - you can make a pull request into any active branch that suits you.

Pull requests need to be made latest two weeks before a release, see releases.

Step by step pull request!

Let's suppose you've developed a cool new module on the `feature/meteorite` branch and you're ready to merge it into `develop`.

Checklist before you start

- Documentation
- Tests
- Tutorial (if a complete new feature)
- Updated dependencies (if need be)
- Added your name to the AUTHORS file
- (Advanced, optional) interactively rebase/squash recent commits that *aren't yet on GitHub*.

Step by step pull request!

1) Make sure the `develop` branch is up to date on your own machine

```
git checkout develop
git pull
```

2) Merge `develop` into your feature branch and resolve any conflicts

```
git checkout feature/meteorite
git merge develop
```

In the case of more complex conflicts, you may want to speak with others who worked on the same code. Your IDE should have a tool for conflict resolution.

3) Check all the tests pass locally

```
make unit_test
make integ_test
```

4) Perform a static code analysis using pylint with CLIMADA's configuration `.pylintrc` (in the climada root directory). Jenkins executes it after every push. To do it locally, your IDE probably provides a tool, or you can run `make lint` and see the output in `pylint.log`.

5) Push to GitHub. If you're pushing this branch for the first time, use

```
git push -u origin feature/meteorite
```

and if you're updating a branch that's already on GitHub:

```
git push
```

6) Check all the tests pass on the WCR Jenkins server (https://ied-wcr-jenkins.ethz.ch). See Emanuel's presentation for how to do this! You should regularly be pushing your code and checking this!

7) Create the pull request!

- On the CLIMADA GitHub page, navigate to your feature branch (there's a drop-down menu above the file structure, pointing by default to `main`).
- Above the file structure is a branch summary and an icon to the right labelled "Pull request".
- Choose which branch you want to merge with. This will usually be `develop`, but may be another feature branch for more complex feature development.
- Give your pull request an informative title (like a commit message).
- Write a description of the pull request. This can usually be adapted from your branch's commit messages (you wrote informative commit messages, didn't you?), and should give a high-level summary of the changes, specific points you want the reviewers' input on, and explanations for decisions you've made. The code documentation (and any references) should cover the more detailed stuff.
- Assign reviewers in the page's right hand sidebar. Tag anyone who might be interested in reading the code. You should already have found one or two people who are happy to read the whole request and sign it off (they could also be added to 'Assignees').
- Create the pull request.
- Contact the reviewers to let them know the request is live. GitHub's settings mean that they may not be alerted automatically. Maybe also let people know on the WCR Slack!

8) Talk with your reviewers

- Use the comment/chat functionality within GitHub's pull requests - it's useful to have an archive of discussions and the decisions made.
- Take comments and suggestions on board, but you don't need to agree with everything and you don't need to implement everything.
- If you feel someone is asking for too many changes, prioritise, especially if you don't have time for complex rewrites.
- If the suggested changes and or features don't block functionality and you don't have time to fix them, they can be moved to Issues.

- Chase people up if they're slow. People are slow.

9) Once you implement the requested changes, respond to the comments with the corresponding commit implementing each requested change.

10) If the review takes a while, remember to merge `develop` back into the feature branch every now and again (and check the tests are still passing on Jenkins). Anything pushed to the branch is added to the pull request.

11) Once everyone reviewing has said they're satisfied with the code you can merge the pull request using the GitHub interface. Delete the branch once it's merged, there's no reason to keep it. (Also try not to re-use that branch name later.)

12) Update the `develop` branch on your local machine.

How to review a pull request

- Be friendly
- Decide how much time you can spare and the detail you can work in. Tell the author!
- Use the comment/chat functionality within GitHub's pull requests - it's useful to have an archive of discussions and the decisions made.
- Fix the big things first! If there are more important issues, not every style guide has to be stuck to, not every slight increase in speed needs to be pointed out, and test coverage doesn't have to be 100%.
- Make it clear when a change is optional, or is a matter of opinion

At a minimum - Make sure unit and integration tests are passing on Jenkins - (For complete modules) Run the tutorial on your local machine and check it does what it says it does - Check everything is fully documented

At least one reviewer needs to - Review all the changes in the pull request. Read what it's supposed to do, check it does that, and make sure the logic is sound. - Check that the code follows the CLIMADA style guidelines `#TODO: link` - If the code is implementing an algorithm it should be referenced in the documentation. Check it's implemented correctly. - Try to think of edge cases and ways the code could break. See if there's appropriate error handling in cases where the function might behave unexpectedly. - (Optional) suggest easy ways to speed up the code, and more elegant ways to achieve the same goal.

There are a few ways to suggest changes - As questions and comments on the pull request page - As code suggestions (max a few lines) in the code review tools on GitHub. The author can then approve and commit the changes from GitHub pull request page. This is great for typos and little stylistic changes. - If you decide to help the author with changes, you can either push them to the same branch, or create a new branch and make a pull request with the changes back into the branch you're reviewing. This lets the author review it and merge.

6.1.5 General tips and tricks

6.1.6 Ask for help with Git

- Git isn't intuitive, and rewinding or resetting is always work. If you're not certain what you're doing, or if you think you've messed up, send someone a message.

Don't push or commit to `develop` or `main`

- Almost all new additions to CLIMADA should be merged into the `develop` branch with a pull request.
- You won't merge into the `main` branch, except for emergency hotfixes (which should be communicated to the team).
- You won't merge into the `develop` branch without a pull request, except for small documentation updates and typos.
- The above points mean you should never need to push the `main` or `develop` branches.

So if you find yourself on the `main` or `develop` branches typing `git merge ...` or `git push` stop and think again - you should probably be making a pull request.

This can be difficult to undo, so contact someone on the team if you're unsure!

Commit more often than you think, and use informative commit messages

- Committing often makes mistakes less scary to undo

```
git reset --hard HEAD
```

- Detailed commit messages make writing pull requests really easy
- Yes it's boring, but *trust me*, everyone (usually your future self) will love you when they're rooting through the git history to try and understand why something was changed

Commit message syntax guidelines

Basic syntax guidelines taken from here https://chris.beams.io/posts/git-commit/ (on 17.06.2020)

- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line (e.g. "Add new tests")
- Wrap the body at 72 characters (most editors will do this automatically)
- Use the body to explain what and why vs. how
- Separate the subject from body with a blank line (This is best done with a GUI. With the command line you have to use text editor, you cannot do it directly with the git command)
- Put the name of the function/class/module/file that was edited
- When fixing an issue, add the reference gh-ISSUENUMBER to the commit message e.g. "fixes gh-40." or "Closes gh-40." For more infos see here https://docs.github.com/en/enterprise/2.16/user/github/managing-your-work-on-github/closing-issues-using-keywords#about-issue-references.

What not to commit

There are a lot of things that don't belong in the Git repository: - Don't commit data, except for config files and very small files for tests. - Don't commit anything containing passwords or authentication credentials or tokens. (These are annoying to remove from the Git history.) Contact the team if you need to manage authorisations within the code. - Don't commit anything that can be created by the CLIMADA code itself

If files like this are going to be present for other users as well, add them to the repository's `.gitignore`.

Log ideas and bugs as GitHub Issues

If there's a change you might want to see in the code - something that generalises, something that's not quite right, or a cool new feature - it can be set up as a GitHub Issue. Issues are pages for conversations about changes to the codebase and for logging bugs, and act as a 'backlog' for the CLIMADA project.

For a bug, or a question about functionality, make a minimal working example, state which version of CLIMADA you are using, and post it with the Issue.

How not to mess up the timeline

Git builds the repository through incremental edits. This means it's great at keeping track of its history. But there are a few commands that *edit* this history, and if histories get out of sync on different copies of the repository you're going to have a bad time.

- Don't rebase any commits that already exist remotely!
- Don't `--force` anything that exists remotely unless you know what you're doing!
- Otherwise, you're unlikely to do anything irreversible
- You can do what you like with commits that only exist on your machine.

That said, doing an interactive rebase to tidy up your commit history *before* you push it to GitHub is a nice friendly gesture :)

Don't fast forward merges

(This shouldn't be relevant - all your merges into `develop` should be through pull requests, which doesn't fast forward. But:)

Don't fast forward your merges unless your branch is a single commit. Use `git merge --no-ff ...`

The exceptions is when you're merging `develop` into your feature branch.

Merge the remote `develop` branch into your feature branch every now and again

- This way you'll find conflicts early

```
git checkout develop
git pull
git checkout feature/myfeature
git merge develop
```

Create frequent pull requests

I said this already: - It structures your workflow - It's easier for reviewers - If you're going to break something for other people you all know sooner - It saves work for the rest of the team right before a release

Whenever you do something with CLIMADA, make a new local branch

You never know when a quick experiment will become something you want to save for later.

But don't do everything in the CLIMADA repository

- If you're running CLIMADA rather than developing it, create a new folder, initialise a new repository with `git init` and store your scripts and data there
- If you're writing an extension to CLIMADA that doesn't change the model core, create a new folder, initialise a new repository with `git init` and import CLIMADA. You can always add it to the model later if you need to.

Questions

https://xkcd.com/1597/

6.2 CLIMADA Tutorial Template

6.2.1 Content

1. *Why tutorials*
2. *Basic structure*
3. *Good examples*

1. Why tutorials

Main goal: The main goal of the tutorials is it to give a complete overview on: * essential CLIMADA components * introduce newly developed modules and features

More specifically, tutorials should introduce CLIMADA users to the core functionalities and modules and guide users in their application. Hence, each new module created needs to be accompanied with a tutorial. The following sections give an overview of the basic structure desired for CLIMADA tutorials.

Important: A tutorial needs to be included with the final pull request for every new feature.

2. Basic structure

Every tutorial should cover the following main points. Additional features characteristic to the modules presented can and should be added as see fit.

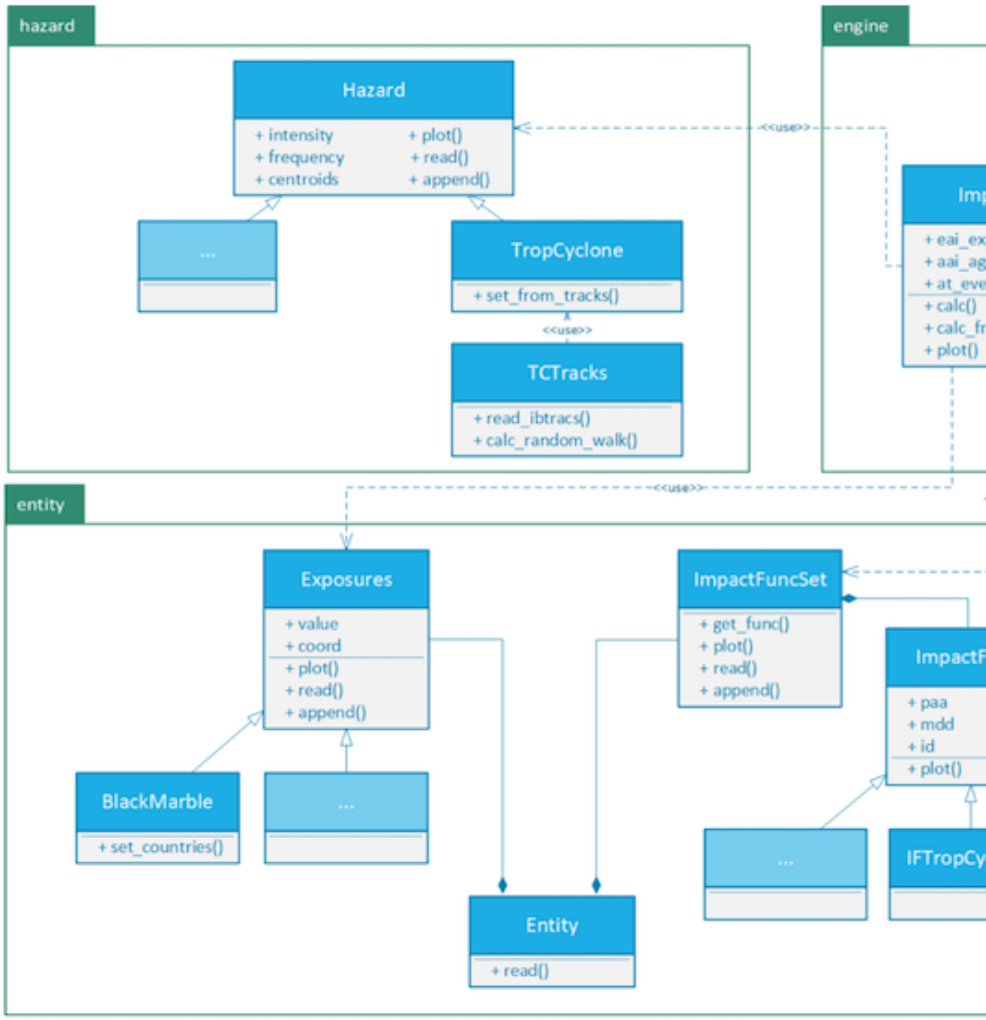
6.2.2 Introduction

- What is the feature presented? Briefly describe the feature and introduce how it's presented in the CLIMADA framework.
- What is its data structure? Present and overview (in the form of a table for example) of where the feature is built into CLIMADA. What class does it belong to, what are the variables of the feature, what is their data structure.
- Table of content: How is this tutorial structured?

6.2.3 Illustration of feature functionality and application

Walk users through the core functions of the module and illustrate how the feature can be used. This obviously is dependent on the feature itself. A few core points should be considered when creating the tutorial: * **SIZE MATTERS!** * each notebook as a total should not exceed the critical (yet vague) size of "a couple MB" * keep the size of data you use as examples in the tutorial in mind * we aim for computational efficiency * a lean, well-organized, concise notebook is more informative than a long, messy all-encompassing one.

- follow the general CLIMADA naming convention for the notebook. For example: "cli-



mada_hazard_TropCyclone.ipynb"

3. Good examples

The following examples can be used as templates and inspiration for your tutorial: * https://github.com/CLIMADA-project/climada_python/blob/tutorial_update/doc/tutorial/climada_entity_Exposures.ipynb * https://github.com/CLIMADA-project/climada_python/blob/tutorial_update/doc/tutorial/climada_hazard_Hazard.ipynb

6.3 Constants and Configuration

6.3.1 Content

1. *Constants*
    1. *Hard Coded*
    2. *Conigurable*
    3. *Where to put constants?*
2. *Configuration*

1. *Config files*
2. *Accessing configuration values*
3. *Default Configuration*
4. *Test Configuration*

1. Constants

Constants are values that, once initialized, are never changed during the runtime of a program. In Python constants are assigned to variables with capital letters by convention, and vice versa, variables with capital letters are supposed to be constants.

In principle there are about four ways to define a constant's value: - *hard coding*: the value is defined in the python code directly - *argument*: the value is taken from an execution argument - *context*: the value is derived from the environmental context of the exection, e.g., the current working directory or the date-time of execution start. - *configuation*: read from a file or database

In CLIMADA, we only use *hard coding* and *configuration* to assign values to constants.

6.3.2 1.A. Hard Coded

Hard coding constants is the prefered way to deal with strings that are used to identify objects or files.

[22]:
```
# suboptimal
my_dict = {'x': 4}
if my_dict['x'] > 3:
    msg = 'well, arh, ...'
msg
```

[22]:
```
'well, arh, ...'
```

[21]:
```
# good
X = 'x'
my_dict = {X: 4}
if my_dict[X] > 3:
    msg = 'yeah!'
msg
```

[21]:
```
'yeah!'
```

[28]:
```
# possibly overdoing it
X = 'x'
Y = "this doesn't mean that every string must be a constant"
my_dict = {X: 4}
if my_dict[X] > 3:
    msg = Y
msg
```

[28]:
```
"this doesn't mean that every string must be a constant"
```

[26]:
```
import pandas as pd
X = 'x'
df = pd.DataFrame({'x':[1,2,3], 'y':[4,5,6]})
```

(continues on next page)

(continued from previous page)

```
try:
    df.X
except:
    from sys import stderr; stderr.write("this does not work\n")
df[X] # this does work but it's less pretty
df.x
```

```
this does not work
```

[26]:
```
0    1
1    2
2    3
Name: x, dtype: int64
```

6.3.3 1.B. Configurable

When it comes to absolute pathes, it is urgently suggested to not use hard coded constant values, for the obvious reasons. But also relative pathes can cause problems. In particular, they may point to a location where the user has not sufficient access permissions. In order to avoid these problems, *all* pathes constants in CLIMADA are supposed to be defined through configuration.
→ pathes must be configurable

The same applies tu urls to external resources, databases or websites. Since they may change at any time, there addresses are supposed to be defined through configuration. Like this it will be possible to access them without the need of tampering with the source code or waiting for a new release.
→ urls must be configurable

Another category of constants that should go into the configuration file are system specifications, such as number of CPU's available for CLIMADA or memory settings.
→ OS settings must be configurable

6.3.4 1.C. Where to put constants?

As a general rule, constants are defined in the module where they intrinsically belong to. If they belong equally to different modules though or they are meant to be used globally, there is the module `climada.util.constants` which is compiling constants CLIMADA wide.

2. Configuration

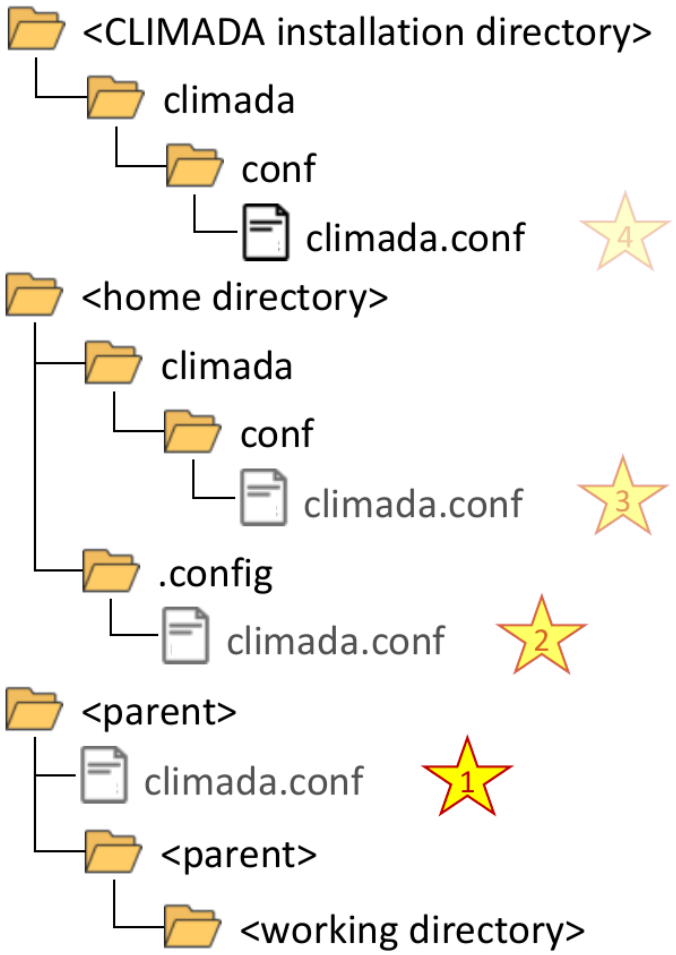
6.3.5 2.A. Configuration files

The proper place to define constants that a user may want (or need) to change without changing the CLIMADA installation are the configuration files.
These are files in *json* format with the name `climada.conf`. There is a default config file that comes with the installation of CLIMADA. But it's possible to have several of them. In this case they are complementing one another.

CLIMADA looks for configuration files upon `import climada`. There are four loacations to look for configuration files: - `climada/conf`, the installation directory - `~/climada/conf`, the user's default climada directory - `~/.config`, the user's configuration directory, - `.`, the current working directory

At each location, the path is followed upwards until a file called `climada.conf` is found or the root of the path is reached. Hence, if e.g., `~/climada/climada.conf` is missing but `~/climada.conf` is present, the latter would be read.

When two config files are defining the same value, the priorities are:
`[..]/./climada.conf` > `~/.config/climada.conf` > `~/climada/conf/climada.conf` > `installation_dir/climada/conf/climada.conf`



A configuration file is a JSON file, with the additional restriction, that all keys must be strings without a '.' (dot) character .

For configuration valules that belong to a particular module it is suggested to reflect the code repositories file structure in the json object. For example if a configuration for `my_config_value` that blongs to the module `climada.util.dates_times` is wanted, it would be defined as

```
{
  "util": {
    "dates_times": {
      "my_config_value": 42
    }
  }
}
```

Configuration string values can be referenced from other configuration values. E.g.

```
{
  "a": "x",
  "b": "{a}y"
}
```

In this example "b" is eventually resolved to "xy".

6.3.6 2.B. Accessing configuration values

Configuration values can be accessed through the (constant) `CONFIG` from the `climada` module:

[1]:
```
from climada import CONFIG
```

[3]:
```
CONFIG.hazard
```

[3]:
```
{drought: {resources: {spei_file_url: http://digital.csic.es/bitstream/10261/153475/8}},␣
↪landslide: {resources: {opensearch: https://pmmpublisher.pps.eosdis.nasa.gov/
↪opensearch, climatology_monthly: https://svs.gsfc.nasa.gov/vis/a000000/a004600/a004631/
↪frames/9600x5400_16x9_30p/MonthlyClimatology/[01-12]_ClimatologyMonthly_032818_
↪9600x5400.tif}, local_data: .}, relative_cropyield: {local_data: ~/climada/data/ISIMIP_
↪crop}, trop_cyclone: {random_seed: 54}}
```

The configuration itself and its attributes have the data type `climada.util.config.Config`

[12]:
```
CONFIG.__class__, CONFIG.hazard.trop_cyclone.random_seed.__class__
```

[12]:
```
(climada.util.config.Config, climada.util.config.Config)
```

The actual configuration values can be accessed as basic types (float, int, str), provided the definition is according to the respective data type:

[4]:
```
CONFIG.hazard.trop_cyclone.random_seed.int()
```

[4]:
```
54
```

[3]:
```
try:
    CONFIG.hazard.trop_cyclone.random_seed.str()
except Exception as e:
    from sys import stderr; stderr.write(f"cannot convert random_seed to str: {e}\n")
```

```
cannot convert random_seed to str: <class 'int'>, not str
```

However, configuration string values can be converted to `pathlib.Path` objects if they are pointing to a directory.

```
[19]: CONFIG.hazard.relative_cropyield.local_data.dir()
```

```
[19]: WindowsPath('C:/Users/me/climada/data/ISIMIP_crop')
```

Note that converting a configuration string to a `Path` object like this will create the specified directory on the fly, unless `dir` is called with the parameter `create=False`.

6.3.7 2.C. Default Configuration

The conifguration file `climada/conf/climada.conf` contains the default configuration.

On the top level it has the following attributes - **local_data**: definition of main pathes for accessing and storing CLIMADA related data - **system**: top directory, where (persistent) climada data is stored

default: `~/climada/data` - **demo**: top directory for data that is downloaded or created in the CLIMADA tutorials

default: `~/climada/demo/data` - **save_dir**: directory where transient (non-persistent) data is stored

default: `./results` - **log_level**: minimum log level showed by logging, one of DEBUG, INFO, WARNING, ERROR or CRITICAL.

default: `INFO` - **max_matrix_size**: maximum matrix size that can be used, can be decreased in order to avoid memory issues

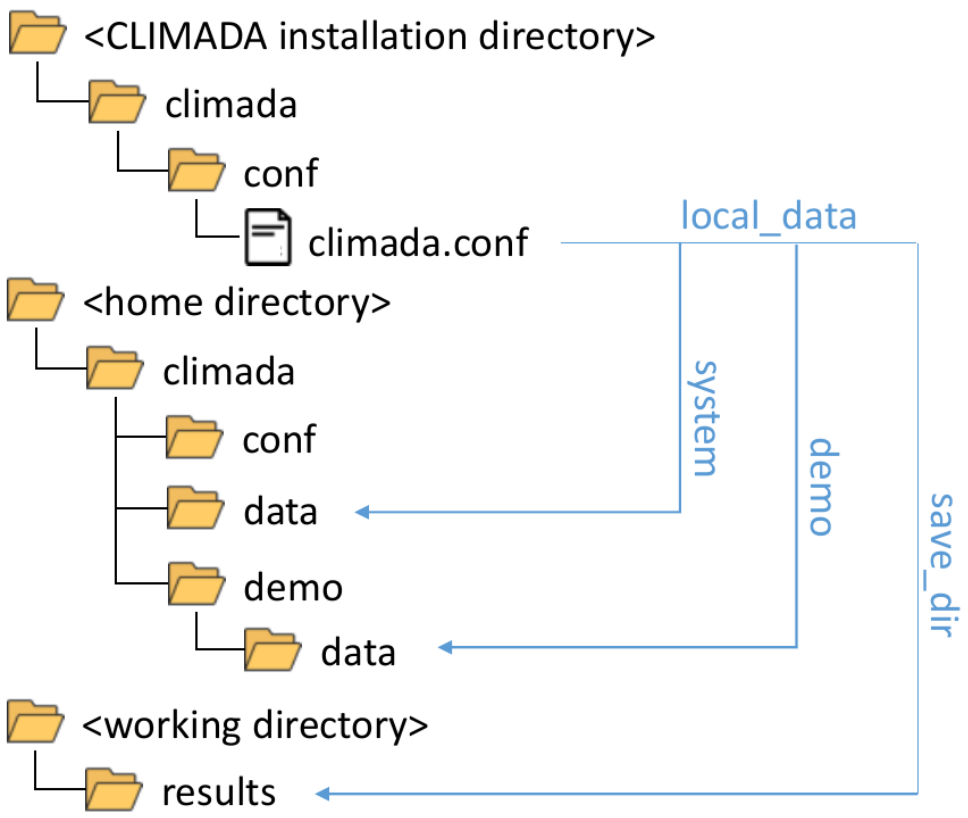
default: `100000000` (1e8) - **exposures**: exposures modules specific configuration - **hazard**: hazard modules specific configuration

```
[5]: CONFIG.__dict__.keys()
```

```
[5]: dict_keys(['_root', '_comment', 'local_data', 'exposures', 'hazard', 'log_level', 'max_
↪matrix_size'])
```

When `import climada` is executed in a python script or shell, data files from the installation directory are copied to the location specified int the current configuration.

This happens only when climada is used for the first time with the current configuration. Subsequent execution will only check for presence of files and won't overwrite existing files.



Thus, the home directory will automatically be populated with a climada directory and several files from the repository when climada is used.

To prevent this and keep the home directory clean, create a config file `~/.config/climada.conf` with customized values for `local_data.system` and `local_data.demo`.

As an example, a file with the following content would suppress creation of directories and copying of files during execution of CLIMADA code:

```
{
    "local_data": {
        "system": "/path/to/installation-dir/climada/data/system",
        "demo": "/path/to/installation-dir/climada/data/demo",
    },
}
```

6.3.8 2.D. Test Configuration

The configuration values for unit and integration tests are not part of the default configuration (*2.C*), since they are irrelevant for the regular CLIMADA user and only aimed for developers.

The default test configuration is defined in the `climada.conf` file of the installation directory. This file contains pathes to files that are read during tests. If they are part of the GitHub repository, their path i.g. starts with the `climada` folder within the installation directory:

```
{
    "_comment": "this is a climada configuration file meant to supersede the default␣
↪configuration in climada/conf during test",
    "test_directory": "./climada",
    "test_data": "{test_directory}/test/data",
    "disc_rates": {
        "test_data": "{test_directory}/entity/disc_rates/test/data"
    },
    ...
}
```

Obviously, the default `test_directory` is given as the relative path to `./climada`. This is fine if (but only if) unit or integration tests are started from the installation directory, which is the case in the automated tests on the CI server.

Developers who intend to start a test from another working directory may have to edit this file and replace the relative path with the absolute path to the installation directory:

```
{
    "_comment": "this is a climada configuration file meant to supersede the default␣
↪configuration in climada/conf during test",
    "test_directory": "/path/to/installation-dir/climada",
    "test_data": "{test_directory}/test/data",
    "disc_rates": {
        "test_data": "{test_directory}/entity/disc_rates/test/data"
    },
    ...
}
```

6.4 Testing and Continuous Integration

6.4.1 Content

1. *Testing CLIMADA*
2. *Notes on Testing*
    1. *Basic Test Procedure*
    2. *Testing Types*
    3. *Unit Tests*
    4. *Integration Tests*

5. *System Tests*
6. *Error Messages*
7. *Dealing with External Resources*
8. *Test Configuration*

3. *Continuous Integration*
    1. *Automated Tests*
    2. *Test Coverage*
    3. *Static Code Analysis*
    4. *Jenkins Projects Overview*

1. Testing CLIMADA

- ### Installation Test

  From the intallation directory run `make install_test` It lasts about 45 seconds. If it succeeds, CLIMADA is properly installed and ready to use.

- ### Unit Tests

  From the intallation directory run `make unit_test` It lasts about 5 minutes and runs unit tests for all modules.

- ### Integration Tests

  From the intallation directory run `make integ_test` It lasts about 45 minutes and runs extensive integration tests, during which also data from external resources is read. An open internet connection is required for a successful test run.

2. Notes on Testing

Any programming code that is meant to be used more than once should have a test, i.e., an additional piece of programming code that is able to check whether the original code is doing what it's supposed to do.

Writing tests is work. As a matter of facts, it can be a *lot* of work, depending on the program often more than writing the original code.
Luckily, it essentially follows always the same basic procedure and a there are a lot of tools and frameworks available to facilitate this work.

In CLIMADA we use the Python in-built *test runner* unittest for execution of the tests and the Jenkins framework for *continuous integration*, i.e., automated test execution and code analysis.

Why do we write test?

- The code is most certainly **buggy** if it's not properly tested.
- Software without tests is **worthless**. It won't be trusted and therefore it won't be used.

When do we write test?

- **Before implementation.** A very good idea. It is called *Test Driven Development*.
- **During implementation.** Test routines can be used to run code even while it's not fully implemented. This is better than running it interactively, because the full context is set up by the test. *By command line:* `python -m unittest climada.x.test_y.TestY.test_z` *Interactively:* `climada.x.test_y.TestY().test_z()`
- **Right after implementation.** In case the coverage analysis shows that there are missing tests, see *Test Coverage*.
- **Later, when a bug was encountered.** Whenever a bug gets fixed, also the tests need to be adapted or amended.

6.4.2 2.A. Basic Test Procedure

- **Test data setup** Creating suitable test data is crucial, but not always trivial. It should be extensive enough to cover all functional requirements and yet as small as possible in order to save resources, both in space and time.
- **Code execution** The main goal of a test is to find bugs *before* the user encounters them. Ultimately every single line of the program should be subject to test. In order to achieve this, it is necessary to run the code with respect to the whole parameter space. In practice that means that even a simple method may require a lot of test code. (Bear this in mind when designing methods or functions: the number of required tests increases dramatically with the number of function parameters!)
- **Result validation** After the code was executed the *actual* result is compared to the *expected* result. The expected result depends on test data, state and parametrization. Therefore result validation can be very extensive. In most cases it won't be practical nor required to validate every single byte. Nevertheless attention should be paid to validate a range of results that is wide enough to discover as many thinkable discrepancies as possible.

6.4.3 2.B. Testing types

Despite the common basic procedure there are many different kinds of tests distinguished. (See *WikiPedia:Software testing*). Very commonly a distinction is made based on levels: - **Unit Test**: tests only a small part of the code, a single function or method, essentially without interaction between modules - **Integration Test**: tests whether different methods and modules work well with each other - **System Test**: tests the whole software at once, using the exposed interface to execute a program

6.4.4 2.C. Unit Tests

Unit tests are meant to check the correctness of program units, i.e., single methods or functions, they are supposed to be fast, simple and easy to write.
For each module in CL

- **Each module in CLIMADA has a counter part containing unit tests.**
  Naming suggestion: `climada.x.y` → `climada.x.test.test_y`
- **Write a test class for each class of the module, plus a test class for the module itself in case it contains (module) functions.**

Naming suggestion: `class X` → `class TestX(unittest.TestCase)`, module `climda.x.y` → `class TestY(unittest.TestCase)`

- **Ideally, each method or function should have at least one test method.**
  Naming suggestion: `def xy()` → `def test_xy()`, `def test_xy_suffix1()`, `def test_xy_suffix2()`
  Functions that are created for the sole purpose of structuring the code do not necessarily have their own unit test.
- **Aim at having very fast unit tests!**
  There will be hundreds of unit tests and in general they are called in corpore and expected to finish after a reaonable amount of time. Less than 10 milisecond is good, 2 seconds is the maximum acceptable duration.
- **A unit test shouldn't call more than one climada method or function.**
  The motivation to combine more than one method in a test is usually creation of test data. Try to provide test data by other means. Define them on the spot (within the code of the test module) or create a file in a test data directory that can be read during the test. If this is too tedious, at least move the data acquisition part to the constructor of the test class.
- **Do not use external resources in unit tests.**
  Methods depending on external resources can be skipped from unit tests. SeeDealing with External Resources.

6.4.5 2.D. Integration Tests

Integration tests are meant to check the correctness of interaction between units of a module or a package.
As a general rule, more work is required to write integration tests than to write unit tests and they have longer runtime.

- **Write integration tests for all intended use cases**.
- **Do not expect external resources to be immutable**. If calling on external resources is part of the workflow to be tested, take into account that they may change over time.
  If the according API has means to indicate the precise version of the requested data, make use of it, otherwise, adapt your expectations and leave room for future changes.
  Example given: your function is ultimately relying on the *current* GDP retrieved from an online data provider, and you test it for Switzerland where it's in about 700 Bio CHF at the moment. Leave room for future development, try to be on a reasonably save side, tolerate a range between 70 Bio CHF and 7000 Bio CHF.
- **Test location**: Integration are written in modules `climada.test.test_xy` or in `climada.x.test.test_y`, like the unit tests.
  For the latter it is required that they do not use external resources and that the tests do not have a runtime longer than 2 seconds.

6.4.6 2.E. System Tests

Integration tests are meant to check whether the whole software package is working correctly.

In CLIMADA, the system test that checks the core functionality of the package is executed by calling `make install_test` from the installation directory.

6.4.7 2.F. Error Messages

When a test fails, make sure the raised exception contains all information that might be helpful to identify the exact problem.
If the error message is ever going to be read by someone else than you while still developing the test, you best assume it will be someone who is completely naive about CLIMADA.

Writing extensive failure messages will eventually save more time than it takes to write them.

Putting the failure information into logs is neither required nor sufficient: the automated tests are built around error messages, not logs.
Anything written to `stdout` by a test method is useful mainly for the developer of the test.

6.4.8 2.G. Dealing with External Resources

Methods depending on external resources (calls a url or database) are ideally atomic and doing nothing else than providing data. If this is the case they can be skipped in unit tests on safe grounds - provided they are tested at some point in higher level tests.

In CLIMADA there are the utility functions `climada.util.files_handler.download_file` and `climada.util.files_handler.download_ftp`, which are assigned to exactly this task for the case of external data being available as files.

Any other method that is calling such a data providing method can be made compliant to unit test rules by having an option to replace them by another method. Like this one can write a dummy method in the test module that provides data, e.g., from a file or hard coded, which be given as the optional argument.

[7]:
```
import climada
def x(download_file=climada.util.files_handler.download_file):
    filepath = download_file('http://real_data.ch')
    return Path(filepath).stat().st_size

import unittest
class TestX(unittest.TestCase):
    def download_file_dummy(url):
        return "phony_data.ch"

    def test_x(self):
        self.assertEqual(44, x(download_file=self.download_file_dummy))
```

- When introducing a new external resource, add a test method in `test_data_api.py`.

6.4.9 2.H. Test Configuration

Use the configuration file `climada.config` in the installation directory to define file pathes and external resources used during tests (see the *Constants and Configuration Guide*).

3. Continuous Integration

The CLIMADA Jenkins server used for continuous integration is at (https://ied-wcr-jenkins.ethz.ch).

6.4.10 3.A. Automated Tests

On Jenkins tests are executed and analyzed automatically, in an unbiased environment. The results are stored and can be compared with previous test runs.
Jenkins has a GUI for monitoring individual tests, full test runs and test result trands.
Developers are requested to watch it. At first when they push commits to the code repository, but also later on, when other changes in data or sources may make it necessary to review and refactor code that once passed all tests. ##### Developer guidelines: - All tests must pass before submitting a pull request. - Integration tests don't run on feature branches in Jenkins, therefore developers are requested to run them locally. - After a pull request was accepted and the changes are merged to the develop branch, integration tests may still fail there and have to be adressed.

6.4.11 3.B. Test Coverage

Jenkins also has an interface for exploring code coverage analysis result.
This shows which part of the code has never been run in any test, by module, by function/method and even by single line of code.

Ulitmately every single line of code should be tested.

- Make sure the coverage of novel code is at 100% before submitting a pull request.

Be aware that the having a code coverage alone does not grant that all required tests have been written!
The following artificial exmple would have a 100% coverage and still obviously misses a test for `y(False)`

[27]:
```
def x(b:bool):
    if b:
        print('been here')
        return 4
    else:
        print('been there')
        return 0

def y(b:bool):
    print('been everywhere')
    return 1/x(b)


import unittest
```

(continues on next page)

(continued from previous page)

```
class TestXY(unittest.TestCase):
    def test_x(self):
        self.assertEqual(x(True), 4)
        self.assertEqual(x(False), 0)

    def test_y(self):
        self.assertEqual(y(True), 0.25)

unittest.TextTestRunner().run(unittest.TestLoader().loadTestsFromTestCase(TestXY));
```

```
..
```

```
been here
been there
been everywhere
been here
```

```

----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

6.4.12 3.C. Static Code Analysis

At last Jenkins provides an elaborate GUI for pylint findings which is especially useful when working in feature branches.

Observe it!

- *High Priority Warnings* are as severe as test failures and must be addressed at once.
- Do not introduce new *Medium Priority Warnings*.
- Try to avoid introducing *Low Priority Warnings*, in any case their total number should not increase.

6.4.13 3.D. Jenkins Projects Overview

- **climada_install_env**

  Branch: **develop** Runs every day at 1:30AM CET

  - creates conda environment from scratch
  - runs core functionality system test (`make install_test`)

- **climada_ci_night**

  Branch: **develop** Runs when climada_install_env has finished successfully

  - runs all test modules
  - runs static code analysis

- **climada_branches**

  Branch: **any** Runs when a commit is pushed to the repository

  - runs all test modules *outside of climada.test*
  - runs static code analysis

- **climada_data_api**

  Branch: **develop** Runs every day at 0:20AM CET

  - tests availability of external data APIs

- **climada_data_api**

  Branch: **develop** No automated running

  - tests executability of CLIMADA tutorial notebooks.

6.5 Reviewer Checklist

- The code must be readable without extra effort from your part. The code should be easily readable (for infos e.g. here)
- Include references to the used algorithms in the docstring
- If the algorithm is new, please include a description in the docstring, or be sure to include a reference as soon as you publish the work
- Variable names should be chosen to be clear. Avoid `item, element, var, list, data` etc... A good variable name makes it immediately clear what it contains.
- Avoid as much as possible hard-coded indices for list (no `x = l[0], y = l[1]`). Rather, use tuple unpacking (see here). Note that tuple unpacking can also be used to update variables. For example, the Fibonacci sequence next number pair can be written as `n1, n2 = n2, n1+n2`.
- Do not use mutable (lists, dictionnaries, ...) as default values for functions and methods. Do not write: `def function(default=[]):`

  but use `def function(default=None): if default is None: default=[]`
- Use pythonic loops, list comprehensions
- Make sure the unit tests are testing all the lines of the code. Do not only check for working cases, but also the most common wrong use cases.
- Check the docstrings (Do they follow the Numpydoc conventions, is everything clearly explained, are the default values given and is it clear why they are set to this value)

- Keep the code simple. Avoid using complex Python functionalities whose use is oppaque to non-expert developers unless necessary. For example, the `@staticmethod` decorator should only be used if really necessary. Another example, for counting the dictionnary `colors = ['red', 'green', 'red', 'blue', 'green', 'red']`, version:

```
d = {}
for color in colors:
        d[color] = d.get(color, 0) + 1
```

  is perfectly fine, no need to complicate it to a maybe more pythonic version `d = collections.defaultdict(int) for color in colors: d[color] += 1`

- Did the code writer perform a static code analysis? Does the code respect Pep8 (see also the pylint config file)?
- Did the code writer perform a profiling and checked that there are no obviously ineficient (computation time-wise and memore-wise) parts in the code?

6.6 Coding in Python: Dos and Don'ts

6.6.1 Content

0. *To Code or Not to Code?*
1. *Clean Code* 1.1 *PEP 8 Quickie: Code Layout* 1.2 *PEP 8 Quickie: Basic Naming Conventions* 1.3 *PEP 8 Quickie: Programming Recommendations* 1.4 *Static Code Analysis and PyLint* 1.5 *A few more best practices* 1.6 *Pythonic Code*
2. *Commenting & Documenting* 2.1 *What is what* 2.2 *Numpy-style docstrings*
3. *Exception Handling and Logging* 3.1 *Exception Handling* 3.2 *Logging*
4. *Importing*
5. *How to structure a method or function*
6. *Debugging*

0. To Code or Not to Code?

Before you start implementing functions which then go into the climada code base, you have to ask yourself a few questions:

Has something similar already been implemented? This is far from trivial to answer! First, search for functions in the same module where you'd be implementing the new piece of code. Then, search in the `util` folders, there's a lot of functions in some of the scripts! You could also search the index (a list of all functions and global constants) in the climada documentation for key-words that may be indicative of the functionality you're looking for.

Don't expect this process to be fast!

Even if you want to implement *just* a small helper function, which might take 10mins to write, it may take you 30mins to check the existing code base! That's part of the game! Even if you found something, most likely, it's not the *exact* same thing which you had in mind. Then, ask yourself how you can re-use what's there, or whether you can easily add another option to the existing method to also fit your case, and only if it's nearly impossible or highly unreadable to do so, write your own implementation.

Can my code serve others? You probably have a very specific problem in mind. Yet, think about other use-cases, where people may have a similar problem, and try to either directly account for those, or at least make it easy to

configure to other cases. Providing keyword options and hard-coding as few things as possible is usually a good thing. For example, if you want to write a daily aggregation function for some time-series, consider that other people might find it useful to have a general function that can also aggregate by week, month or year.

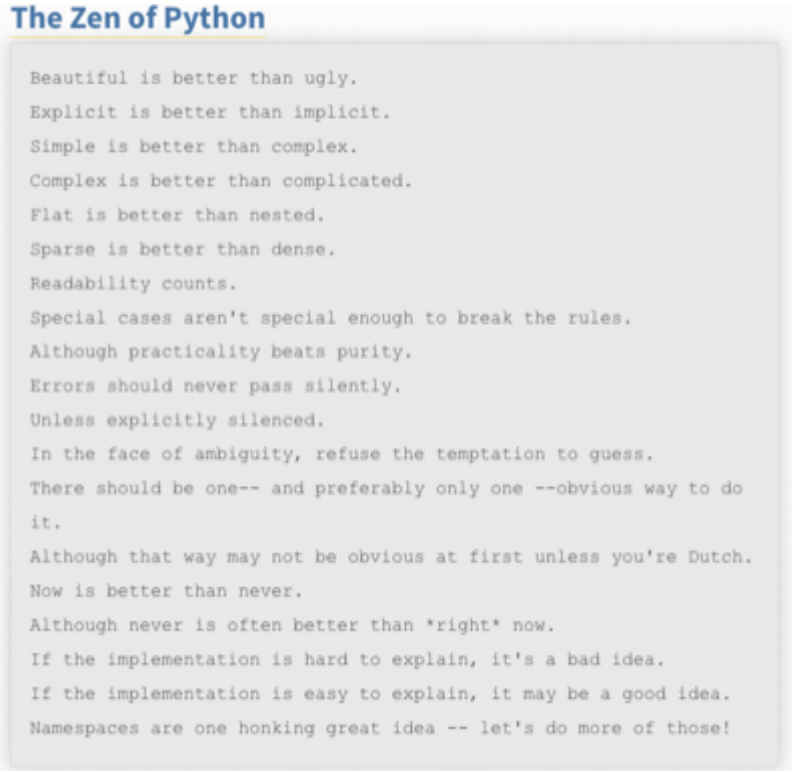
Can I get started? Before you finally start coding, be sure about placing them in a sensible location. Functions in non-util modules are actually specific for that module (e.g. a file-reader function is probably not river-flood specific, so put it into the `util` section, not the `RiverFlood` module, even if that's what you're currently working on)! If unsure, talk with other people about where your code should go.

If you're implementing more than just a function or two, or even an entirely new module, the planning process should be talked over with someone doing climada-administration.

1. Clean Code

A few basic principles:

- Follow the PEP 8 Style Guide. It contains, among others, recommendations on:
  - code layout
  - basic naming conventions
  - programming recommendations
  - commenting (in detail described in Chapter 4)
  - varia
- Perform a static code analysis - or: PyLint is your friend
- Follow the best practices of *Correctness - Tightness - Readability*
- Adhere to principles of pythonic coding (idiomatic coding, the "python way")



6.6.2 1.1 PEP 8 Quickie: Code Layout

- *Indentation*: 4 spaces per level. For continuation lines, decide between vertical alignment & hanging indentation as shown here:
- *Line limit*: maximum of 79 characters (docstrings & comments 72).
- *Blank lines*:
  - **Two**: Surround top-level function and class definitions;
  - **One:** Surround Method definitions inside a class
  - **Several:** may be used (sparingly) to separate groups of related functions
  - **None:** Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).
- *Whitespaces*:
  - **None** immediately inside parentheses, brackets or braces; after trailing commas; for keyword assignments in functions.
  - **Do** for assignments (`i = i + 1`), around comparisons (>=, ==, etc.), around booleans (`and`, `or`, `not`)
- There's more in the PEP 8 guide!

6.6.3 1.2 PEP 8 Quickie: Basic Naming Conventions

A short typology: b (single lowercase letter); B (single uppercase letter); lowercase; lower_case_with_underscores; UPPERCASE; UPPER_CASE_WITH_UNDERSCORES; CapitalizedWords (or CapWords, or CamelCase); mixedCase; Capitalized_Words_With_Underscores (ugly!)

A few basic rules: - packages and modules: short, all-lowercase names. Underscores can be used in the module name if it improves readability. E.g. `numpy`, `climada` - classes: use the CapWords convention. E.g. `RiverFlood` - functions, methods and variables: lowercase, with words separated by underscores as necessary to improve readability. E.g. `from_raster()`, `dst_meta` - function- and method arguments: Always use `self` for the first argument to instance methods,`cls` for the first argument to class methods. - constants: all capital letters with underscores, e.g. `DEF_VAR_EXCEL`

Use of underscores - `_single_leading_underscore`: weak "internal use" indicator. E.g. `from M import *` does not import objects whose names start with an underscore. A side-note to this: Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public. Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed. Public attributes should have no leading underscores. - `single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g. `tkinter.Toplevel(master, class_='ClassName')` - `__double_leading_and_trailing_underscore__`: "magic" objects or attributes that live in user-controlled namespaces. E.g. `__init__`, `__import__` or `__file__`. Never invent such names; only use them as documented.

There are many more naming conventions, some a bit messy. Have a look at the PEP8 style guide for more cases.

6.6.4 1.3 PEP 8 Quickie: Programming Recommendations

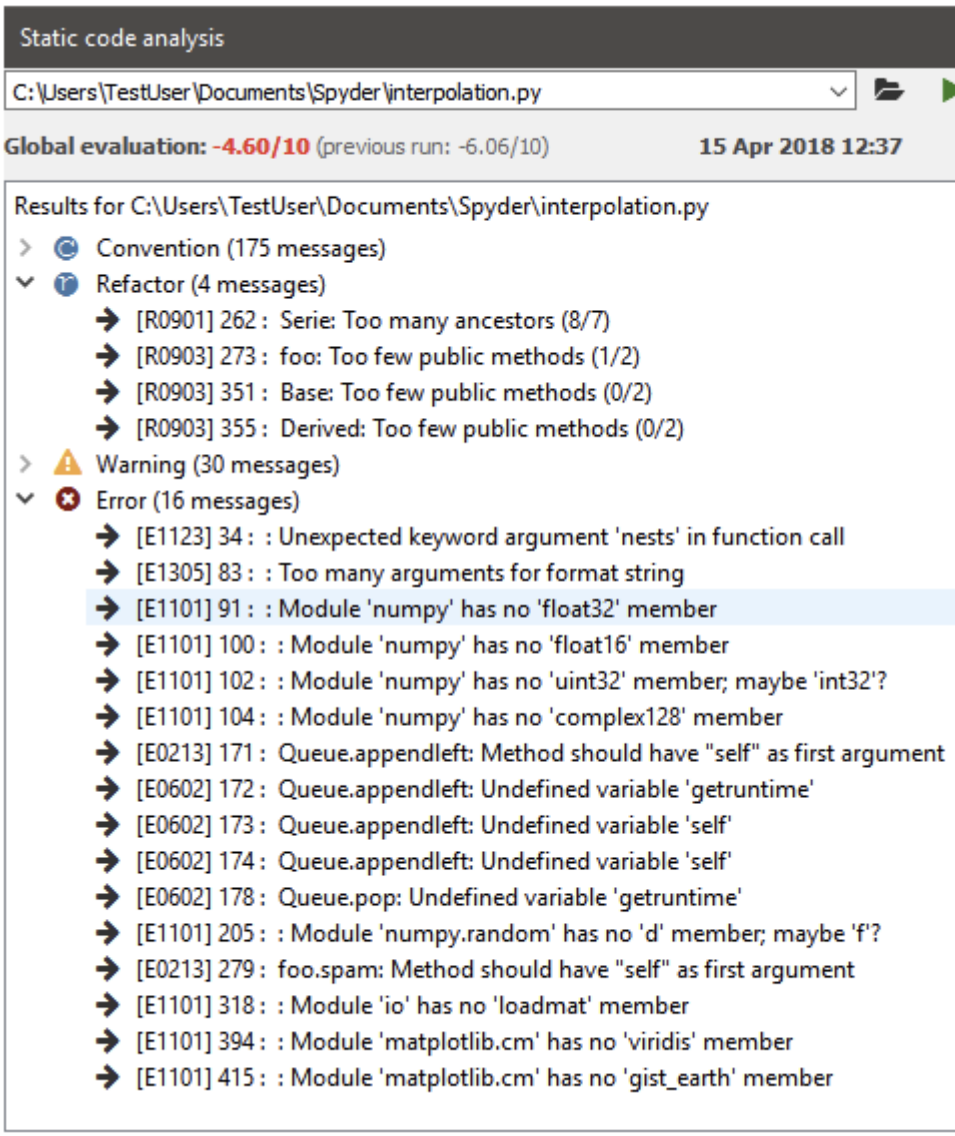
- comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators.
- Use `is not` operator rather than `not ... is`.
- Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. Any return statements where no value is returned should explicitly state this as `return None`.
- Object type comparisons should always use isinstance() instead of comparing types directly:
- Remember: sequences (strings, lists, tuples) are false if empty; this can be used:
- Don't compare boolean values to True or False using ==:
- Use ''.startswith() and".endswith() instead of string slicing to check for prefixes or suffixes.
- Context managers exist and can be useful (mainly for opening and closing files

6.6.5 1.4 Static Code Analysis and PyLint

Static code analysis detects style issues, bad practices, potential bugs, and other quality problems in your code, all without having to actually execute it. In Spyder, this is powered by the best in class Pylint back-end, which can intelligently detect an enormous and customizable range of problem signatures. It follows the style recommended by PEP 8 and also includes the following features: Checking the length of each line, checking that variable names are well-formed according to the project's coding standard, checking that declared interfaces are truly implemented.

A detailed instruction can be found here.

In brief: In the editor, select the Code Analysis pane (if not visible, go to View -> Panes -> Code Analysis) and the file you want to be analysed; hit the Analyse button.



The output will look somewhat similar to that:

There are 4 categories in the analysis output: * *convention*, * *refactor*, * *warning*, * *error* * a global score regarding code quality.

All messages have a line reference and a short description on the issue. Errors *must* be fixed, as this is a no-go for actually executing the script. Warnings and refactoring messages should be taken seriously; so should be the convention messages, even though some of the naming conventions etc. may not fit the project style. This is configurable. In general, there should be no errors and warnings left, and the overall code quality should be in the "green" range (somewhere above 5 or so).

There are advanced options to configure the type of warnings and other settings in pylint.

6.6.6 1.5 A few more best practices

Correctness

Methods and functions must return correct and verifiable results, not only under the best circumstances but in any possible context. I.e. ideally there should be unit tests exploring the full space of parameters, configuration and data states. This is often clearly a non-achievable goal, but still - we aim at it.

Tightness

- Avoid code redundancy.
- Make the program efficient, use profiling tools for detection of bottlenecks.
- Try to minimize memory consumption.
- Don't introduce new dependencies (library imports) when the desired functionality is already covered by existing dependencies.
- Stick to already supported file types.

Readability

- Write complete Python Docstrings.
- Use meaningful method and parameter names, and always annotate the data types of parameters and return values.
- No context-dependent return types! Also: Avoid `None` as return type, rather raise an Exception instead.
- Be generous with defining Exception classes.
- Comment! Comments are welcome to be redundant. And whenever there is a particular reason for the way something is done, comment on it! See below for more detail.
- For functions which implement mathematical/scientific concepts, add the actual mathematical formula as comment or to the Doctstrings. This will help maintain a high level of scientific accuracy. E.g. How is are the random walk tracks computed for tropical cyclones?

6.6.7 1.6 Pythonic Code

In Python, there are certain structures that are specific to the language, or at least the syntax of how to use them. This is usually referred to as "pythonic" code.

There is an extensive overview on on crucial "pythonic" structures and methods in the Python 101 library.

A few important examples are:

- iterables such as dictionaries, tuples, lists
- iterators and generators (a very useful construct when it comes to code performance, as the implementation of generators avoids reading into memory huge iterables at once, and allows to read them lazily on-the-go; see this blog post for more details)
- f-strings ("formatted string literals," have an f at the beginning and curly braces containing expressions that will be replaced with their values:

```
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

- decorators (a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure) something like
- type checking (Python is a dynamically typed language; also: cf. "Duck typing". Yet, as a best practice, variables should not change type once assigned)
- Do not use mutable default arguments in your functions (e.g. lists). For example, if you define a function as such:

```
def function(x, list=[]):
    default_list.append(x)
```

  Your list will be mutated for future calls of the functions too. The correct implementation would bethe following: `def func(x, list=None): list = [] if list is None`

- lambda functions (little, anonymous functions, sth like `high_ord_func(2, lambda x: x * x)`)
- list comprehensions (a short and possibly elegant syntax to create a new list in one line, sth like `newlist = [x for x in range(10) if x < 5]` returns `[0, 1, 2, 3, 4]`)

It is recommended to look up the above concepts in case not familiar with them.

2. Commenting & Documenting

6.6.8 2.1 What is what

Comments are for developers. They describe parts of the code where necessary to facilitate the understanding of programmers. They are marked by putting a # in front of every comment line (for multi-liners, wrapping them inside triple double quotes """ is basically possible, but discouraged to not mess up with docstrings). A *documentation string (docstring)* is a string that describes a module, function, class, or method definition. The docstring is a special attribute of the object (`object.__doc__`) and, for consistency, is surrounded by triple double quotes (""""). This is also where elaboration of the scientific foundation (explanation of used formulae, etc.) should be documented.

A few general rules:

- Have a look at this blog-post on commenting basics
- Comments should be D.R.Y ("Don't Repeat Yourself.")
- Obvious naming conventions can avoid unnecessary comments (cf. `families_by_city[city]` vs. `my_dict[p]`)
- comments should rarely be longer than the code they support
- All public methods need a doc-string. See below for details on the convention used within the climada project.
- Non-public methods that are not immediately obvious to the reader should at least have a short comment after the def line:

```
def complicated_function(s):
    # This function does something complicated
```

6.6.9 2.2 Numpy-style docstrings

Full reference can be found here. The standards are such that they use re-structured text (reST) syntax and are rendered using Sphinx.

There are several sections in a docstring, with headings underlined by hyphens (`---`). The sections of a function's docstring are:

1. *Short summary*: A one-line summary that does not use variable names or the function name
```
def add(a, b):
    """The sum of two numbers.

    """
```
2. *Deprecation warning* (use if applicable): to warn users that the object is deprecated, including version the object that was deprecated, and when it will be removed, reason for deprecation, new recommended way of obtaining the same functionality. Use the `deprecated` Sphinx directive:
```
.. deprecated:: 1.6.0
           `ndobj_old` will be removed in NumPy 2.0.0, it is replaced by
           `ndobj_new` because the latter works also with array subclasses.
```
3. *Extended Summary*: A few sentences giving an extended description to clarify functionality, not to discuss implementation detail or background theory (see `Notes` section below!)
4. *Parameters*:
```
Parameters
----------
x : type
    Description of parameter `x`.
```
Description of the function arguments, keywords and their respective types. Enclose variables in single backticks in the description. The colon must be preceded by a space, or omitted if the type is absent. For the parameter types, be as precise as possible. If it is not necessary to specify a keyword argument, use `optional` after the type specification: e.g. `x: int, optional`. Default values of optional parameters can also be detailed in the description. (e.g. `... description of parameter ... (default is -1)`)
5. *Returns*: Explanation of the returned values and their types. Similar to the Parameters section, except the name of each return value is optional, type isn't. If both the name and type are specified, the Returns section takes the same form as the Parameters section.
```
Returns
-------
err_code : int
    Non-zero value indicates error code, or zero on success.
err_msg : str or None
    Human readable error message, or None on success.
```

There is a range of other sections that can be included, if sensible and applicable, such as `Yield` (for generator functions only), `Raises` (which errors get raised and under what conditions), `See also` (refer to related code), `Notes` (additional information about the code, possibly including a discussion of the algorithm; may include mathematical equations, written in LaTeX format), `References`, `Examples`(to illustrate usage).

3. Exception Handling and Logging

Exception handling and logging are two important components of programming, in particular for debugging purposes. Detailled technical guides are available online (e.g., Loggin, Error and Exceptions). Here we only repeat a few key points and list a few guidelines for CLIMADA.

6.6.10 3.1 Exception handling

1. Catch specific exceptions if possible, i.e, if not needed do not catch all exceptions.
2. Do not catch exception if you do not handle them.
3. Make a clear explanatory message when you raise an error (similarly to when you use the logger to inform the user). Think of future users and how it helps them understanding the error and debugging their code.
4. Catch an exception when it arises.
5. When you can an exception and raise an error, it is in often (but not always) a good habit to not throw away the first caught exception as it may contain useful information for debugging. (use `raise Error from`)

[1]:
```
#Bad (1)
x = 1
try:
    l = len(events)
    if l < 1:
        print("l is too short")
except:
    pass
```

[ ]:
```
#Still bad (2)
try:
    l = len(events)
    if l < 1:
        print("l is too short")
except TypeError:
    pass
```

[ ]:
```
#Better, but still unsufficient (3)
try:
    l = len(events)
    if l < 1:
        raise ValueError("To compute an impact there must be at least one event.")
except TypeError:
    raise TypeError("The provided variable events is not a list")
```

[ ]:
```
#Even better (4)
try:
    l = len(events)
except TypeError:
    raise TypeError("The provided variable events is not a list")
if l < 1:
    raise ValueError("To compute an impact there must be at least one event.")
```

[ ]:
```
#Even better (5)
try:
    l = len(events)
except TypeError as tper:
    raise TypeError("The provided variable events is not a list") from tper
if l < 1:
    raise ValueError("To compute an impact there must be at least one event.")
```

Why do we bother to handle exceptions?

- The most essential benefit is to inform the user of the error, while still allowing the program to proceed.

6.6.11 3.2 Logging

- In CLIMADA, you cannot use printing. Any output must go into the LOGGER.
- For any logging messages, always think about the audience. What would a user or developer need for information? This also implies to carefully think about the correct LOGGER level. For instance, some information is for debugging, then use the debug level. In this case, make sure that the message actually helps the debudding process! Some message might just to inform the user about certain default parameters, then use the inform level. See below for more details about logger levels.
- Do not overuse the LOGGER. Think about which level of logging. Logging errors must be useful for debugging.

You can set the level of the LOGGER using `climada.util.config.LOGGER.setLevel(logging.XXX)`. This way you can for instance 'turn-off' info messages when you are making an application. For example, setting the logger to the "ERROR" level, use:

[ ]:
```
import logging
from climada.util.config import LOGGER
LOGGER.setLevel(logging.ERROR)
```

What levels to use in CLIMADA?

- Debug: what you would print while developing/debugging
- Info: information for example in the check instance
- Warning: whenever CLIMADA fills in values, makes an extrapolation, computes something that might potentially lead to unwanted results (e.g., the 250year damages extrapolated from data over 20 years)

No known use case:

- Error: instead, raise an Error and add the message (raise ValueError("Error message"))
- Critical: ...

"Logging is a means of tracking events that happen when some software runs."

When to use logging

"Logging provides a set of convenience functions for simple logging usage. These are debug(), info(), warning(), error() and critical(). To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it."

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

Logger level

"The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):"

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

4. Importing

General remarks

- Imports should be grouped in the following order:
  - Standard library imports (such as `re`, `math`, `datetime`, cf. here)
  - Related third party imports (such as `numpy`)
  - Local application/library specific imports (such as `climada.hazard.base`)
- You should put a blank line between each group of imports.

- Don't introduce new dependencies (library imports) when the desired functionality is already covered by existing dependencies.

Avoid circular importing!! Circular imports are a form of circular dependencies that are created with the import statement in Python; e.g. module A loads a method in module B, which in turn requires loading module A. This can generate problems such as tight coupling between modules, reduced code reusability, more difficult maintainance. Circular dependencies can be the source of potential failures, such as infinite recursions, memory leaks, and cascade effects. Generally, they can be resolved with better code design. Have a look here for tips to identify and resolve such imports.

Varia * there are absolute imports (uses the full path starting from the project's root folder) and relative imports (uses the path starting from the current module to the desired module; usually in the for `from .<module/package> import X`; dots `.` indicate how many directories upwards to traverse. A single dot corresponds to the current directory; two dots indicate one folder up; etc.) * generally try to avoid star imports (e.g. `from packagename import *`)

Importing utility functions

When importing CLIMADA utility functions (from climada.util), the convention is to import the function as "u_name_of_function", e.g.:

```
from climada.util import coordinates as u_coord
u_coord.make_map()
```

5. How to structure a method or function

To clarify ahead: The questions of how to structure an entire module, or even "just" a class, are not treated here. For this, please get in contact with the repository admins to help you go devise a plan.

The following few principles should be adhered to when designing a function or method (which is simply the term for a function inside a class):

- have a look at this blog-post summarizing a few important points to define your function (key-words *abstraction, reusability, modularity*)
- separate algorithmic computations and data curation
- adhere to a maximum method length (rule of thumb: if it doesn't fit your screen, it's probably an indicator that you should refactor into sub-functions)
- divide functions into single purpose pieces (one function, one goal)

6. Debugging

When writing code, you will encounter bugs and hence go through (more or less painful) debugging. Depending on the IDE you use, there are different debugging tools that will make your life much easier. They offer functionalities such as stopping the execution of the function just before the bug occurs (via breakpoints), allowing to explore the state of defined variables at this moment of time.

For spyder specifically, have a look at the instructions on how to use ipdb

6.7 Python performance tips and best practice for CLIMADA developers

This guide covers the following recommendations:

Use profiling tools to find and assess performance bottlenecks. **Replace for-loops** by built-in functions and efficient external implementations. **Consider algorithmic performance**, not only implementation performance. **Get familiar with NumPy:** vectorized functions, slicing, masks and broadcasting. **Miscellaneous:** sparse arrays, Numba, parallelization, huge files (xarray), memory. **Don't over-optimize** at the expense of readability and usability.

Table of Contents

6.7.1 1 Profiling

Python comes with powerful packages for the **performance assessment** of your code. Within IPython and notebooks, there are several magic commands for this task:

- `%time`: Time the execution of a single statement
- `%timeit`: Time repeated execution of a single statement for more accuracy
- `%%timeit` Does the same as `%timeit` for a whole cell
- `%prun`: Run code with the profiler
- `%lprun`: Run code with the line-by-line profiler
- `%memit`: Measure the memory use of a single statement
- `%mprun`: Run code with the line-by-line memory profiler

More information on profiling in the Python Data Science Handbook.

Also useful: unofficial Jupyter extension Execute Time.

While it's easy to assess how fast or slow parts of your code are, including finding the bottlenecks, **generating an improved version of it is much harder**. This guide is about **simple best practices** that everyone should know who works with Python, especially when models are performance-critical.

In the following, we will **focus on arithmetic operations** because they play an important role in CLIMADA. Operations on non-numeric objects like strings, graphs, databases, file or network IO might be just as relevant inside and outside of the CLIMADA context. Some of the tips presented here do also apply to other contexts, but **it's always worth looking for context-specific performance guides**.

6.7.2 2 General considerations

This section will be concerned with:

for-loops and built-ins **external implementations** and converting data structures **algorithmic efficiency** **memory usage**

As this section's toy example, let's assume we want to sum up all the numbers in a list:

[ ]:
```
list_of_numbers = list(range(10000))
```

2.1 for-loops

A developer with a background in C++ would probably loop over the entries of the list:

[2]:
```
%%timeit
result = 0
for i in list_of_numbers:
    result += i
```

```
332 µs ± 65.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The built-in function `sum` is much faster:

[3]:
```
%timeit sum(list_of_numbers)
```

```
54.9 µs ± 5.63 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The timing improves by a factor of 5-6 and this is not a coincidence: **for-loops generally tend to get prohibitively expensive** when the number of iterations increases.

When you have a for-loop with many iterations in your code, check for built-in functions or efficient external implementations of your programming task.

A special case worth noting are `append` operations on lists which can often be replaced by more efficient *list comprehensions*.

2.2 Converting data structures

When you find an external library that solves your task efficiently, always consider that it might be necessary to convert your data structure which takes time.

For arithmetic operations, NumPy is a great library, but if your data comes as a Python list, NumPy will spend quite some time converting it to a NumPy array:

[4]:
```
import numpy as np
%timeit np.sum(list_of_numbers)
```

```
572 µs ± 80 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

This operation is even slower than the for-loop!

However, if you can somehow obtain your data in the form of **NumPy arrays from the start**, or if you perform many operations that might compensate for the conversion time, the gain in performance can be considerable:

[5]:
```
# do the conversion outside of the `%timeit`
ndarray_of_numbers = np.array(list_of_numbers)
%timeit np.sum(ndarray_of_numbers)
```

```
10.6 µs ± 1.56 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Indeed, this is 5-6 times faster than the built-in `sum` and 20-30 times faster than the for-loop.

2.3 Always consider several implementations

Even for such a basic task as summing, there exist several implementations whose performance can vary more than you might expect:

[6]:
```
%timeit ndarray_of_numbers.sum()
%timeit np.einsum("i->", ndarray_of_numbers)
```

```
9.07 µs ± 1.39 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.55 µs ± 383 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

This is up to 50 times faster than the for-loop. More information about the `einsum` function will be given in the NumPy section of this guide.

2.4 Efficient algorithms

Consider algorithmic performance, not only implementation performance.

All of the examples above do exactly the same thing, algorithmically. However, often the largest performance improvements can be obtained from **algorithmical changes**. This is the case when your model or your data contain symmetries or more complex structure that allows you to skip or boil down arithmetic operations.

In our example, we are summing the numbers from 1 to 10,000 and it's a well known mathematical theorem that this can be done using only two multiplications and an increment:

[7]:
```
n = max(list_of_numbers)
%timeit 0.5 * n * (n + 1)
```

```
83.1 ns ± 2.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Not surprisingly, This is almost 100 times faster than even the fastest implementation of the 10,000 summing operations listed above.

You don't need a degree in maths to find algorithmical improvements. Other algorithmical improvements that are often easy to detect are: * **Filter your data set as much as possible** to perform operations only on those entries that are really relevant. **Example:** When computing a physical hazard (e.g. extreme wind) with CLIMADA, restrict to Centroids on land unless you know that some of your exposure is off shore. * Make sure to **detect inconsistent or trivial input parameters early on**, before starting any operations. **Example:** If your code does some complicated stuff and applies a user-provided normalization factor at the very end, make sure to check that the factor is not 0 before you start applying those complicated operations.

In general: Before starting to code, take pen and paper and write down what you want to do from an algorithmic perspective.

2.5 Memory usage

Be careful with deep copies of large data sets and only load portions of large files into memory as needed.

Write your code in such a way that you **handle large amounts of data chunk by chunk** so that Python does not need to load everything into memory before performing any operations. When you do, Python's generators might help you with the implementation.

Allocating unnecessary amounts of memory might slow down your code substantially due to swapping.

6.7.3 3 NumPy-related tips and best practice

As mentioned above, arithmetic operations in Python can profit a lot from NumPy's capabilities. In this section, we collect some tips how to make use of NumPy's capabilities when performance is an issue.

3.1 Vectorized functions

We mentioned above that Python's **for-loops are really slow**. This is even more important when looping over the entries in a NumPy array. Fortunately, NumPy's masks, slicing notation and vectorization capabilities help to avoid for-loops in almost every possible situation:

[8]:
```
# TASK: compute the column-sum of a 2-dimensional array
input_arr = np.random.rand(100, 3)
```

[9]:
```
%%timeit
# SLOW: summing over columns using loops
output = np.zeros(100)
for row_i in range(input_arr.shape[0]):
    for col_i in range(input_arr.shape[1]):
        output[row_i] += input_arr[row_i, col_i]
```

```
145 µs ± 5.47 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[10]:
```
# FASTER: using NumPy's vectorized `sum` function with `axis` attribute
%timeit output = input_arr.sum(axis=1)
```

```
4.23 µs ± 216 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In the special case of multiplications and sums (linear operations) over the axes of two multi-dimensional arrays, NumPy's ``**einsum**`` is even faster:

[11]:
```
%timeit output = np.einsum("ij->i", input_arr)
```
```
2.38 µs ± 214 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Another `einsum` example: **Euclidean norms**

[12]:
```
many_vectors = np.random.rand(1000, 3)
%timeit np.sqrt((many_vectors**2).sum(axis=1))
%timeit np.linalg.norm(many_vectors, axis=1)
%timeit np.sqrt(np.einsum("...j,...j->...", many_vectors, many_vectors))
```
```
24.4 µs ± 2.18 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
26.5 µs ± 2.44 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
9.5 µs ± 91.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

For more information about the capabilities of NumPy's `einsum` function, refer to the official NumPy documentation. However, note that future releases of NumPy will eventually improve the performance of core functions, so that `einsum` will become an example of over-optimization (see above) at some point. Whenever you use `einsum`, consider adding a comment that explains what it does for users that are not familiar with `einsum`'s syntax.

Not only `sum`, but many NumPy functions come with similar vectorization capabilities. You can take minima, maxima, means or standard deviations along selected axes. But did you know that the same is true for the `diff` and `argmin` functions?

[13]:
```
arr = np.random.randint(low=0, high=10, size=(4, 3))
arr
```
[13]:
```
array([[4, 2, 6],
       [2, 3, 4],
       [3, 3, 3],
       [3, 2, 4]])
```

[14]:
```
arr.argmin(axis=1)
```
[14]:
```
array([1, 0, 0, 1])
```

3.2 Broadcasting

When operations are performed on several arrays, possibly of differing shapes, be sure to use NumPy's **broadcasting** capabilities. This will save you a lot of memory and time when performing arithmetic operations.

Example: We want to multiply the columns of a two-dimensional array by values stored in a one-dimensional array. There are two naive approaches to this:

[15]:
```
input_arr = np.random.rand(100, 3)
col_factors = np.random.rand(3)
```

[16]:
```
# SLOW: stack/tile the one-dimensional array to be two-dimensional
%timeit output = np.tile(col_factors, (input_arr.shape[0], 1)) * input_arr
```
```
5.67 µs ± 718 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[17]:
```
%%timeit
# SLOW: loop over columns and factors
```

(continues on next page)

(continued from previous page)

```
output = input_arr.copy()
for i, factor in enumerate(col_factors):
    output[:, i] *= factor
```

```
9.63 µs ± 95.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The idea of *broadcasting* is that NumPy **automatically matches axes from right to left and implicitly repeats data along missing axes** if necessary:

[18]:
```
%timeit output = col_factors * input_arr
```

```
1.41 µs ± 51.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

For automatic broadcasting, the *trailing* dimensions of two arrays have to match. NumPy is matching the shapes of the arrays *from right to left*. If you happen to have arrays where other dimensions match, **you have to tell NumPy which dimensions to add by adding an axis of length 1 for each missing dimension**:

[19]:
```
input_arr = np.random.rand(3, 100)
row_factors = np.random.rand(3)
output = row_factors.reshape(3, 1) * input_arr
```

Because this concept is so important, there is a short-hand notation for adding an axis of length 1. In the slicing notation, **add ``None`` in those positions where broadcasting should take place**.

[20]:
```
input_arr = np.random.rand(3, 100)
row_factors = np.random.rand(3)
output = row_factors[:, None] * input_arr
```

[21]:
```
input_arr = np.random.rand(7, 3, 5, 4, 6)
factors = np.random.rand(7, 3, 4)
output = factors[:, :, None, :, None] * input_arr
```

3.3 A note on in-place operations

While **in-place operations** are generally faster than long and explicit expressions, they shouldn't be over-estimated when looking for performance bottlenecks. Often, the loss in code readability is not justified because NumPy's memory management is really fast.

Don't over-optimize!

[22]:
```
shape = (1200, 1700)
arr_a = np.random.rand(*shape)
arr_b = np.random.rand(*shape)
arr_c = np.random.rand(*shape)
```

[23]:
```
# long expression in one line
%timeit arr_d = arr_c * (arr_a + arr_b) - arr_a + arr_c
```

```
17.3 ms ± 820 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[24]:
```
%%timeit
# almost same performance: in-place operations
arr_d = arr_a + arr_b
```

(continues on next page)

(continued from previous page)

```
arr_d *= arr_c
arr_d -= arr_a
arr_d += arr_c
```

```
17.4 ms ± 618 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[25]:
```
%load_ext memory_profiler
```

[26]:
```
# long expression in one line
%memit arr_d = arr_c * (arr_a + arr_b) - arr_a + arr_c
```

```
peak memory: 156.68 MiB, increment: 31.20 MiB
```

[27]:
```
%%memit
# almost same memory usage: in-place operations
arr_d = arr_a + arr_b
arr_d *= arr_c
arr_d -= arr_a
arr_d += arr_c
```

```
peak memory: 157.27 MiB, increment: 0.00 MiB
```

6.7.4 4 Miscellaneous

4.1 Sparse matrices

In many contexts, we deal with sparse matrices or sparse data structures, i.e. two-dimensional arrays where most of the entries are 0. In CLIMADA, this is especially the case for the `intensity` attributes of `Hazard` objects. This kind of data is usually handled using SciPy's submodule scipy.sparse.

When dealing with sparse matrices make sure that you always understand exactly which of your variables are sparse and which are dense and only switch from sparse to dense when absolutely necessary.

Multiplications (``multiply``) and matrix multiplications (``dot``) are often faster than operations that involve masks or indexing.

As an example for the last rule, consider the problem of multiplying certain rows of a sparse array by a scalar:

[28]:
```
import scipy.sparse as sparse

array = np.tile(np.array([0, 0, 0, 2, 0, 0, 0, 1, 0], dtype=np.float64), (100, 80))
row_mask = np.tile(np.array([False, False, True, False, True], dtype=bool), (20,))
```

In the following cells, note that the code in the first line after the `%%timeit` statement is not timed, it's the setup line.

[29]:
```
%%timeit sparse_array = sparse.csr_matrix(array)
sparse_array[row_mask, :] *= 5
```

```
/home/tovogt/.local/share/miniconda3/envs/tc/lib/python3.7/site-packages/scipy/sparse/
↪data.py:55: RuntimeWarning: overflow encountered in multiply
  self.data *= other
```

```
1.52 ms ± 155 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[30]:
```
%%timeit sparse_array = sparse.csr_matrix(array)
sparse_array.multiply(np.where(row_mask, 5, 1)[:, None]).tocsr()
```

```
340 µs ± 7.32 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[31]:
```
%%timeit sparse_array = sparse.csr_matrix(array)
sparse.diags(np.where(row_mask, 5, 1)).dot(sparse_array)
```

```
400 µs ± 6.43 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

4.2 Fast for-loops using Numba

As a last resort, if there's no way to avoid a for-loop even with NumPy's vectorization capabilities, you can use the `@njit` decorator provided by the Numba package:

[32]:
```
from numba import njit

@njit
def sum_array(arr):
    result = 0.0
    for i in range(arr.shape[0]):
        result += arr[i]
    return result
```

In fact, the Numba function is more than 100 times faster than without the decorator:

[33]:
```
input_arr = np.float64(np.random.randint(low=0, high=10, size=(10000,)))
```

[34]:
```
%timeit sum_array(input_arr)
```

```
10.9 µs ± 444 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[35]:
```
# Call the function without the @njit
%timeit sum_array.py_func(input_arr)
```

```
1.84 ms ± 65.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

However, whenever available, **NumPy's own vectorized functions will usually be faster than Numba**.

[36]:
```
%timeit np.sum(input_arr)
%timeit input_arr.sum()
%timeit np.einsum("i->", input_arr)
```

```
7.6 µs ± 687 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.27 µs ± 411 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.89 µs ± 499 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Make sure you understand the basic idea behind Numba before using it, read the Numba docs**.**

Don't use ``@jit``, but use ``@njit`` which is an alias for ``@jit(nopython=True)``.

When you know what you are doing, the `fastmath` and `parallel` options can boost performance even further: read more about this in the Numba docs.

4.3 Parallelizing tasks

Depending on your hardware setup, parallelizing tasks using pathos and Numba's automatic parallelization feature can improve the performance of your implementation.

Expensive hardware is no excuse for inefficient code.

Many tasks in CLIMADA could profit from GPU implementations. However, currently there are **no plans to include GPU support in CLIMADA** because of the considerable development and maintenance workload that would come with it. If you want to change this, contact the core team of developers, open an issue or mention it in the bi-weekly meetings.

4.4 Read NetCDF datasets with `xarray`

When dealing with NetCDF datasets, memory is often an issue, because even if the file is only a few megabytes in size, the uncompressed raw arrays contained within can be several gigabytes large (especially when data is sparse or similarly structured). One way of dealing with this situation is to open the dataset with `xarray`.

``xarray`` allows to read the shape and type of variables contained in the dataset without loading any of the actual data into memory.

Furthermore, when loading slices and arithmetically aggregating variables, memory is allocated not more than necessary, but values are obtained on-the-fly from the file.

6.7.5 5 Take-home messages

We conclude by repeating the gist of this guide:

Use profiling tools to find and assess performance bottlenecks. **Replace for-loops** by built-in functions and efficient external implementations. **Consider algorithmic performance**, not only implementation performance. **Get familiar with NumPy:** vectorized functions, slicing, masks and broadcasting. **Miscellaneous:** sparse arrays, Numba, parallelization, huge files (xarray), memory. **Don't over-optimize** at the expense of readability and usability.

6.8 Miscellaneous CLIMADA conventions

Table of Contents

6.8.1 Dependencies (python packages)

Python is extremely powerful thanks to the large amount of available libraries, packages and modules. However, maintaining a code with a large number of such packages creates dependencies which is very care intensive. Indeed, each package developer can and does update and develop continuously. This means that certain code can become obsolete over time, stop working altogether, or become incompatible with other packages. Hence, it is crucial to keep the philosophie:

As many packages as needed, as few as possible.

Thus, when you are coding, follow these priorities:

1. Python standard library
2. Funktions and methods already implemented in CLIMADA (do NOT introduce circulary imports though)
3. Packages already included in CLIMADA
4. Before adding a new dependency:
    - Contact a repository admin to get permission
    - Open an issue

Hence, first try to solve your problem with the standard library and function/methods already implemented in CLIMADA (see in particular the *util functions*) then use the packages included in CLIMADA, and if this is not enough, propose the addition of a new package. Do not hesitate to propose new packages if this is needed for your work!

6.8.2 Class inheritance

In Python, a class can inherit from other classes, which is a very useful mechanism in certain circumstantce. However, it is wise to think about inheritance before implementing it. Very important, is that CLIMADA classes do not inherit from external library classes. For example, Exposure directly inherited from Geopandas. This caused problems in CLIMADA when the package Geopandas was updated.

CLIMADA classes shall NOT inherit classes from external modules

6.8.3 Does it belong into CLIMADA?

When developing for CLIMADA, it is important to distinguish between core content and particular applications. Core content is meant to be included into the climada_python repository and will be subject to a code review. Any new addition should first be discussed with one of the repository admins. The purpose of this discussion is to see

- How does the planed module fit into CLIMADA?
- What is an optimal architecture for the new module?
- What parts might already exist in other parts of the code?

Applications made with CLIMADA, such as an ECA study can be stored in the paper repository once they have been published. For other types of work, consider making a separate repository that imports CLIMADA as an external package.

6.8.4 Paper repository

Applications made with CLIMADA which are published in the form of a paper or a report are very much encouraged to be submited to the climada/paper repository. You can either:

- Prepare a well-commented jupyter notebook with the code necessary to reproduce your results and upload it to the climada/paper repository. Note however that the repository cannot be used for storing data files.
- Upload the code necessary to reproduce your results to a separate repository of your own. Then, add a link to your repository and to your publication to the readme file on the climada/paper repository.

Notes about DOI

Some journals requires you to provide a DOI to the code and data used for your publication. In this case, we encourage to create a separate repository for your code and create a DOI using Zenodo or any specific service from your institution (e.g. ETH Zürich).

The CLIMADA releases are also identified with a DOI.

6.8.5 Utility function

In CLIMADA, there is a set of utility functions defined in `climada.util`. A few examples are:

- convert large monetary numbers into thousands, millions or billions together with the correct unit name
- compute distances
- load hdf5 files
- convert iso country numbers between formats
- ...

Whenever you develop a module or make a code review, be attentive to see whether a given functionnality has already been implemented as a utility function. In addition, think carefully whether a given function/method does belong in its module or is actually independent of any particular module and should be defined as a utility function.

It is very important to not reinvent the wheel and to avoid unnecessary redundancies in the code. This makes maintenance and debugging very tedious.

6.8.6 Impact function renaming - if to impf

In the original CLIMADA code, the impact function is often referred to as `if` or `if_`. This is easy to confuse with the conditional operator *if*. Hence, in future a transition from

`if` ———> `impf`

will be performed. Once the change is active, known developers will be notified and this message updated.

6.8.7 Data dependencies

6.8.8 Web APIs

CLIMADA relies on open data available through web APIs such as those of the World Bank, Natural Earth, NASA and NOAA. You might execute the test `climada_python-x.y.z/test_data_api.py` to check that all the APIs used are active. If any is out of service (temporarily or permanently), the test will indicate which one.

6.8.9 Manual download

As indicated in the software and tutorials, other data might need to be downloaded manually by the user. The following table shows these last data sources, their version used, its current availabilty and where they are used within CLIMADA:

Availability	Name	Version	Link	CLIMADA class	CLIMADA version	CLIMADA tutorial reference
OK	Fire Information for Resource Management System		**`FIRMS <https://firms.modaps.eosdis.nasa.gov/download/ >`__**	BushFire	>V1.2.5	climada_hazard_BushFire.ipynb
OK	Gridded Population of the World (GPW)	v4.11	GPW4.11	LitPop	> v1.2.3	climada_entity_LitPop.ipynb
FAILED	Gridded Population of the World (GPW)	v4.10	GPW1.10	LitPop	>= v1.2.0	climada_entity_LitPop.ipynb

6.8.10 Side Note on Parameters

Don't use *args and **kwargs parameters without a very good reason.

There *are* valid use cases for this kind of parameter notation.

In particular `*args` comes in handy when there is an unknown number of equal typed arguments to be passed. E.g., the `pathlib.Path` constructor.

But if the parameters are expected to be structured in any way, it is just a bad idea.

[4]:
```
def f(x, y, z):
    return x + y + z

# bad in most cases
def g(*args, **kwargs):
    x = args[0]
    y = kwargs['y']
    s = f(*args, **kwargs)
    print(x, y, s)

g(1,y=2,z=3)
```

```
1 2 6
```

[ ]:
```
# usually just fine
def g(x, y, z):
    s = f(x, y, z)
    print(x, y, s)

g(1,y=2,z=3)
```

Decrease the number of parameters.

Though CLIMADA's pylint configuration .pylintrc allows 7 arguments for any method or function before it complains, it is advisable to aim for less. It is quite likely that a function with so many parameters has an inherent design flaw.

There are very well designed command line tools with innumerable optional arguments, e.g., rsync - but these are command line tools. There are also methods like pandas.DataFrame.plot() with countless optional arguments and it makes perfectly sense.

But within the climada package it probably doesn't. divide et impera!

Whenever a method has more than 5 parameters, it is more than likely that it can be refactored pretty easily into two or more methods with less parameters and less complexity:

[2]:
```
def f(a, b, c, d, e, f, g, h):
    print(f'f does many things with a lot of arguments: {a, b, c, d, e, f, g, h}')
    return sum([a, b, c, d, e, f, g, h])

f(1, 2, 3, 4, 5, 6, 7, 8)
```

```
f does many things with a lot of arguments: (1, 2, 3, 4, 5, 6, 7, 8)
```

[2]:
```
36
```

[3]:
```
def f1(a, b, c, d):
    print(f'f1 does less things with fewer arguments: {a, b, c, d}')
    return sum([a, b, c, d])

def f2(e, f, g, h):
    print(f'f2 dito: {e, f, g, h}')
    return sum([e, f, g, h])

def f3(x, y):
    print(f'f3 dito, but on a higher level: {x, y}')
    return sum([x, y])

f3(f1(1, 2, 3, 4), f2(5, 6, 7, 8))
```

```
f1 does less things with fewer arguments: (1, 2, 3, 4)
f2 dito: (5, 6, 7, 8)
f3 dito, but on a higher level: (10, 26)
```

[3]:
```
36
```

This of course pleads the case on a strictly formal level. No real complexities have been reduced during the making of this example.

Nevertheless there is the benefit of reduced test case requirements. And in real life real complexity *will* be reduced.

CHAPTER

SEVEN

SOFTWARE DOCUMENTATION

Documents functions, classes and methods:

7.1 Software documentation per package

7.1.1 climada.engine package

climada.engine.unsequa package

climada.engine.unsequa.calc_base module

class climada.engine.unsequa.calc_base.**Calc**

Bases: object

Base class for uncertainty quantification

Contains the generic sampling and sensitivity methods. For computing the uncertainty distribution for specific CLIMADA outputs see the subclass CalcImpact and CalcCostBenefit.

__init__()

Empty constructor to be overwritten by subclasses

check_distr()

Log warning if input parameters repeated among input variables

Return type True.

property input_vars

Uncertainty variables

Returns All uncertainty variables associated with the calculation

Return type tuple(UncVar)

property distr_dict

Dictionary of the input variable distribution

Probabilitiy density distribution of all the parameters of all the uncertainty variables listed in self.InputVars

Returns **distr_dict** – Dictionary of all probability density distributions.

Return type dict(sp.stats objects)

est_comp_time(*n_samples*, *time_one_run*, *pool=None*)

Estimate the computation time

Parameters

- **n_samples** (*int/float*) – The total number of samples
- **time_one_run** (*int/float*) – Estimated computation time for one parameter set in seconds
- **pool** (*pathos.pool, optional*) – pool that would be used for parallel computation. The default is None.

Return type Estimated computation time in secs.

make_sample(*N*, *sampling_method='saltelli'*, *sampling_kwargs=None*)

Make samples of the input variables

For all input parameters, sample from their respective distributions using the chosen sampling_method from SALib. https://salib.readthedocs.io/en/latest/api.html

This sets the attributes unc_output.samples_df, unc_output.sampling_method, unc_output.sampling_kwargs.

Parameters

- **N** (*int*) – Number of samples as used in the sampling method from SALib
- **sampling_method** (*str, optional*) – The sampling method as defined in SALib. Possible choices: 'saltelli', 'fast_sampler', 'latin', 'morris', 'dgsm', 'ff' https://salib.readthedocs.io/en/latest/api.html The default is 'saltelli'.
- **sampling_kwargs** (*kwargs, optional*) – Optional keyword arguments passed on to the SALib sampling_method. The default is None.

Returns **unc_output** – Uncertainty data object with the samples

Return type climada.engine.uncertainty.unc_output.UncOutput()

See also:

SALib.sample sampling methods from SALib SALib.sample

https //salib.readthedocs.io/en/latest/api.html

sensitivity(*unc_output*, *sensitivity_method='sobol'*, *sensitivity_kwargs=None*)

Compute the sensitivity indices using SALib.

Prior to doing the sensitivity analysis, one must compute the uncertainty (distribution) of the output values (with self.uncertainty()) for all the samples (rows of self.samples_df).

According to Wikipedia, sensitivity analysis is "the study of how the uncertainty in the output of a mathematical model or system (numerical or otherwise) can be apportioned to different sources of uncertainty in its inputs." The sensitivity of each input is often represented by a numeric value, called the sensitivity index. Sensitivity indices come in several forms.

This sets the attributes: sens_output.sensistivity_method sens_output.sensitivity_kwargs sens_output.xxx_sens_df for each metric unc_output.xxx_unc_df

Parameters

- **unc_output** (*climada.engine.uncertainty.unc_output.UncOutput()*) – Uncertainty data object in which to store the sensitivity indices
- **sensitivity_method** (*str*) – sensitivity analysis method from SALib.analyse Possible choices:

  'fast', 'rbd_fact', 'morris', 'sobol', 'delta', 'ff'

  The default is 'sobol'. Note that in Salib, sampling methods and sensitivity analysis methods should be used in specific pairs. https://salib.readthedocs.io/en/latest/api.html

- **sensitivity_kwargs** (*dict(), optional*) – Keyword arguments of the chosen SALib analyse method. The default is to use SALib's default arguments.

Returns **sens_output** – Uncertainty data object with all the sensitivity indices, and all the uncertainty data copied over from unc_output.

Return type climada.engine.uncertainty.unc_output.UncOutput()

climada.engine.unsequa.calc_cost_benefit module

class climada.engine.unsequa.calc_cost_benefit.**CalcCostBenefit**(*haz_input_var*, *ent_input_var*, *haz_fut_input_var=None*, *ent_fut_input_var=None*)

Bases: *climada.engine.unsequa.calc_base.Calc*

Cost Benefit uncertainty analysis class

This is the base class to perform uncertainty analysis on the outputs of climada.engine.costbenefit.CostBenefit().

metric_names
Names of the cost benefit output metris ('tot_climate_risk', 'benefit', 'cost_ben_ratio',

'imp_meas_present', 'imp_meas_future')

Type tuple(str)

value_unit
Unit of the exposures value

Type str

input_var_names
Names of the required uncertainty variables ('haz_input_var', 'ent_input_var', 'haz_fut_input_var', 'ent_fut_input_var')

Type tuple(str)

haz_input_var
Present Hazard uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

ent_input_var
Present Entity uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

haz_unc_fut_Var
Future Hazard uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

ent_fut_input_var
Future Entity uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

__init__(*haz_input_var*, *ent_input_var*, *haz_fut_input_var=None*, *ent_fut_input_var=None*)
Initialize UncCalcCostBenefit

Sets the uncertainty input variables, the cost benefit metric_names, and the units.

Parameters

- **haz_input_var** (*climada.engine.uncertainty.input_var.InputVar*) – or climada.hazard.Hazard Hazard uncertainty variable or Hazard for the present Hazard in climada.engine.CostBenefit.calc
- **ent_input_var** (*climada.engine.uncertainty.input_var.InputVar*) – or climada.entity.Entity Entity uncertainty variable or Entity for the present Entity in climada.engine.CostBenefit.calc
- **haz_fut_input_var** (*climada.engine.uncertainty.input_var.InputVar*) – or climada.hazard.Hazard, optional Hazard uncertainty variable or Hazard for the future Hazard The Default is None.
- **ent_fut_input_var** (*climada.engine.uncertainty.input_var.InputVar*) – or climada.entity.Entity, optional Entity uncertainty variable or Entity for the future Entity in climada.engine.CostBenefit.calc

uncertainty(*unc_data*, *pool=None*, ***cost_benefit_kwargs*)

Computes the cost benefit for each sample in unc_output.sample_df.

By default, imp_meas_present, imp_meas_future, tot_climate_risk, benefit, cost_ben_ratio are computed.

This sets the attributes: unc_output.imp_meas_present_unc_df, unc_output.imp_meas_future_unc_df unc_output.tot_climate_risk_unc_df unc_output.benefit_unc_df unc_output.cost_ben_ratio_unc_df unc_output.unit unc_output.cost_benefit_kwargs

Parameters

- **unc_data** (*climada.engine.uncertainty.unc_output.UncOutput*) – Uncertainty data object with the input parameters samples
- **pool** (*pathos.pools.ProcessPool, optional*) – Pool of CPUs for parralel computations. Default is None. The default is None.
- **cost_benefit_kwargs** (*keyword arguments*) – Keyword arguments passed on to climada.engine.CostBenefit.calc()

Returns **unc_output** – Uncertainty data object in with the cost benefit outputs for each sample and all the sample data copied over from unc_sample.

Return type climada.engine.uncertainty.unc_output.UncCostBenefitOutput

Raises **ValueError:** – If no sampling parameters defined, the uncertainty distribution cannot be computed.

See also:

climada.engine.cost_benefit Compute risk and adptation option cost benefits.

climada.engine.unsequa.calc_impact module

class climada.engine.unsequa.calc_impact.**CalcImpact**(*exp_input_var*, *impf_input_var*, *haz_input_var*)

Bases: *climada.engine.unsequa.calc_base.Calc*

Impact uncertainty caclulation class.

This is the class to perform uncertainty analysis on the outputs of a climada.engine.impact.Impact() object.

rp

List of the chosen return periods.

Type list(int)

calc_eai_exp
Compute eai_exp or not

Type bool

calc_at_event
Compute eai_exp or not

Type bool

metric_names
Names of the impact output metris ('aai_agg', 'freq_curve', 'at_event', 'eai_exp', 'tot_value')

Type tuple(str)

value_unit
Unit of the exposures value

Type str

input_var_names
Names of the required uncertainty input variables ('exp_input_var', 'impf_input_var', 'haz_input_var')

Type tuple(str)

exp_input_var
Exposure uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

impf_input_var
Impact function set uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

haz_input_var
Hazard uncertainty variable

Type climada.engine.uncertainty.input_var.InputVar

__init__(*exp_input_var*, *impf_input_var*, *haz_input_var*)
Initialize UncCalcImpact

Sets the uncertainty input variables, the impact metric_names, and the units.

Parameters

- **exp_input_var** (*climada.engine.uncertainty.input_var.InputVar or climada.entity.Exposure*) – Exposure uncertainty variable or Exposure
- **impf_input_var** (*climada.engine.uncertainty.input_var.InputVar or climada.entity.ImpactFuncSet*) – Impact function set uncertainty variable or Impact function set
- **haz_input_var** (*climada.engine.uncertainty.input_var.InputVar or climada.hazard.Hazard*) – Hazard uncertainty variable or Hazard

uncertainty(*unc_sample*, *rp=None*, *calc_eai_exp=False*, *calc_at_event=False*, *pool=None*)
Computes the impact for each sample in unc_data.sample_df.

By default, the aggregated average annual impact (impact.aai_agg) and the excees impact at return periods rp (imppact.calc_freq_curve(self.rp).impact) is computed. Optionally, eai_exp and at_event is computed (this may require a larger amount of memory if the number of samples and/or the number of centroids and/or exposures points is large).

This sets the attributes self.rp, self.calc_eai_exp, self.calc_at_event, self.metrics.

This sets the attributes: unc_output.aai_agg_unc_df, unc_output.freq_curve_unc_df unc_output.eai_exp_unc_df unc_output.at_event_unc_df unc_output.tot_value_unc_df unc_output.unit

Parameters

- **unc_sample** (*climada.engine.uncertainty.unc_output.UncOutput*) – Uncertainty data object with the input parameters samples
- **rp** (*list(int), optional*) – Return periods in years to be computed. The default is [5, 10, 20, 50, 100, 250].
- **calc_eai_exp** (*boolean, optional*) – Toggle computation of the impact at each centroid location. The default is False.
- **calc_at_event** (*boolean, optional*) – Toggle computation of the impact for each event. The default is False.
- **pool** (*pathos.pools.ProcessPool, optional*) – Pool of CPUs for parralel computations. The default is None.

Returns **unc_output** – Uncertainty data object with the impact outputs for each sample and all the sample data copied over from unc_sample.

Return type climada.engine.uncertainty.unc_output.UncImpactOutput

Raises **ValueError:** – If no sampling parameters defined, the distribution cannot be computed.

See also:

climada.engine.impact Compute risk.

climada.engine.unsequa.input_var module

class climada.engine.unsequa.input_var.**InputVar**(*func*, *distr_dict*)

Bases: object

Input variable for the uncertainty analysis

An uncertainty input variable requires a single or multi-parameter function. The parameters must follow a given distribution. The uncertainty input variables are the input parameters of the model.

distr_dict

Distribution of the uncertainty parameters. Keys are uncertainty parameters names and Values are probability density distribution from the scipy.stats package https://docs.scipy.org/doc/scipy/reference/stats.html

Type dict

labels

Names of the uncertainty parameters (keys of distr_dict)

Type list

func

User defined python fucntion with the uncertainty parameters as keyword arguements and which returns a climada object.

Type function

Notes

A few default Variables are defined for Hazards, Exposures, Impact Fucntions, Measures and Entities.

Examples

Categorical variable function: LitPop exposures with m,n exponents in [0,5] import scipy as sp def litpop_cat(m, n):

exp = Litpop.from_countries('CHE', exponent=[m, n]) return exp

distr_dict = { 'm': sp.stats.randint(low=0, high=5), 'n': sp.stats.randint(low=0, high=5) }

iv_cat = InputVar(func=litpop_cat, distr_dict=distr_dict)

Continuous variable function: Impact function for TC import scipy as sp def imp_fun_tc(G, v_half, vmin, k, _id=1):

imp_fun = ImpactFunc() imp_fun.haz_type = 'TC' imp_fun.id = _id imp_fun.intensity_unit = 'm/s' imp_fun.intensity = np.linspace(0, 150, num=100) imp_fun.mdd = np.repeat(1, len(imp_fun.intensity)) imp_fun.paa = np.array([sigmoid_function(v, G, v_half, vmin, k)

for v in imp_fun.intensity])

imp_fun.check() impf_set = ImpactFuncSet() impf_set.append(imp_fun) return impf_set

distr_dict = {"G": sp.stats.uniform(0.8, 1), "v_half": sp.stats.uniform(50, 100), "vmin": sp.stats.norm(loc=15, scale=30), "k": sp.stats.randint(low=1, high=9) }

iv_cont = InputVar(func=imp_fun_tc, distr_dict=distr_dict)

`__init__`(*func*, *distr_dict*)

Initialize InputVar

Parameters

- **func** (*function*) – Variable defined as a function of the uncertainty parameters
- **distr_dict** (*dict*) – Dictionary of the probability density distributions of the uncertainty parameters, with keys matching the keyword arguments (i.e. uncertainty parameters) of the func function. The distribution must be of type scipy.stats https://docs.scipy.org/doc/scipy/reference/stats.html

`evaluate`(***params*)

Return the value of uncertainty input variable.

By default, the value of the average is returned.

Parameters ****params** (*optional*) – Input parameters will be passed to self.InputVar_func.

Returns **unc_func(**params)** – Output of the uncertainty variable.

Return type climada object

`plot`(*figsize=None*)

Plot the distributions of the parameters of the uncertainty variable.

Parameters **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is derived from the total number of plots (nplots) as:

nrows, ncols = int(np.ceil(nplots / 3)), min(nplots, 3) figsize = (ncols * FIG_W, nrows * FIG_H)

Returns **axes** – The figure and axes handle of the plot.

Return type matplotlib.pyplot.figure, matplotlib.pyplot.axes

`static var_to_inputvar(var)`

Returns an uncertainty variable with no distribution if var is not an InputVar. Else, returns var.

Parameters **var** (*climada.uncertainty.InputVar or any other CLIMADA object*)

Returns var if var is InputVar, else InputVar with var and no distribution.

Return type *InputVar*

`static haz(haz, n_ev=None, bounds_int=None, bounds_freq=None)`

Helper wrapper for basic hazard uncertainty input variable

The following types of uncertainties can be added: HE: sub-sampling events from the total event set

For each sub-sample, n_ev events are sampled with replacement. HE is the value of the seed for the uniform random number generator.

HI: scale the intensity of all events (homogeneously) The instensity of all events is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_int

HF: scale the frequency of all events (homogeneously) The frequency of all events is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_freq

If a bounds is None, this parameter is assumed to have no uncertainty.

Parameters

- **haz** (*climada.hazard.Hazard*) – The base hazard
- **n_ev** (*int, optional*) – Number of events to be subsampled per sample. Can be equal or larger than haz.size. The default is None.
- **bounds_int** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous intensity scaling. The default is None.
- **bounds_freq** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous frequency scaling. The default is None.

Returns Uncertainty input variable for a hazard object.

Return type *climada.engine.unsequa.input_var.InputVar*

`static exp(exp_list, bounds_totval=None, bounds_noise=None)`

Helper wrapper for basic exposure uncertainty input variable

The following types of uncertainties can be added: ET: scale the total value (homogeneously)

The value at each exposure point is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_totvalue

EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator.

EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents.

If a bounds is None, this parameter is assumed to have no uncertainty.

Parameters

- **exp_list** (*list of climada.entity.exposures.Exposures*) – The list of base exposure. Can be one or many to uniformly sample from.
- **bounds_totval** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous total value scaling.. The default is None.
- **bounds_noise** ((*float, float), optional*) – Bounds of the uniform distribution to scale each exposure point independently. The default is None.

Returns Uncertainty input variable for an exposure object.

Return type *climada.engine.unsequa.input_var.InputVar*

`static impfset`(*impf_set*, *haz_id_dict=None*, *bounds_mdd=None*, *bounds_paa=None*, *bounds_impfi=None*)

Helper wrapper for basic impact function set uncertainty input variable.

One impact function (chosen with haz_type and fun_id) is characterized.

The following types of uncertainties can be added: MDD: scale the mdd (homogeneously)

The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd

PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa

IFi: shift the intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_int

If a bounds is None, this parameter is assumed to have no uncertainty.

Parameters

- **impf_set** (*climada.entity.impact_funcs.impact_func_set.ImpactFuncSet*) – The base impact function set.
- **bounds_mdd** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous mdd scaling. The default is None.
- **bounds_paa** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous paa scaling. The default is None.
- **bounds_impfi** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous shift of intensity. The default is None.
- **haz_id_dict** (*dict(), optional*) – Dictionary of the impact functions affected by uncertainty. Keys are hazard types (str), values are a list of impact function id (int). Default is impsf_set.get_ids() i.e. all impact functions in the set

Returns Uncertainty input variable for an impact function set object.

Return type *climada.engine.unsequa.input_var.InputVar*

`static ent`(*impf_set*, *disc_rate*, *exp_list*, *meas_set*, *haz_id_dict*, *bounds_disc=None*, *bounds_cost=None*, *bounds_totval=None*, *bounds_noise=None*, *bounds_mdd=None*, *bounds_paa=None*, *bounds_impfi=None*)

Helper wrapper for basic entity set uncertainty input variable.

Important: only the impact function defined by haz_type and fun_id will be affected by bounds_impfi, bounds_mdd, bounds_paa.

The following types of uncertainties can be added: DR: value of constant discount rate (homogeneously)

The value of the discounts in each year is sampled uniformly from a distribution with (min, max) = bounds_disc

CO: scale the cost (homogeneously) The cost of all measures is multiplied by the same number sampled uniformly from a distribution with (min, max) = bounds_cost

ET: scale the total value (homogeneously) The value at each exposure point is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_totval

EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator.

EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents.

MDD: scale the mdd (homogeneously) The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd

PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa

IFi: shift the intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_int

If a bounds is None, this parameter is assumed to have no uncertainty.

Parameters

- **bounds_disk** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous discount rate scaling. The default is None.
- **bounds_cost** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous cost of all measures scaling. The default is None.
- **bounds_totval** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous total exposure value scaling. The default is None.
- **bounds_noise** (*(float, float), optional*) – Bounds of the uniform distribution to scale each exposure point independently. The default is None.
- **bounds_mdd** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous mdd scaling. The default is None.
- **bounds_paa** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous paa scaling. The default is None.
- **bounds_impfi** (*(float, float), optional*) – Bounds of the uniform distribution for the homogeneous shift of intensity. The default is None.
- **impf_set** (*climada.engine.impact_funcs.impact_func_set.ImpactFuncSet*) – The base impact function set.
- **disc_rate** (*climada.entity.disc_rates.base.DiscRates*) – The base discount rates.
- **exp_list** (*[climada.entity.exposures.base.Exposure]*) – The list of base exposure. Can be one or many to uniformly sample from.
- **meas_set** (*climada.entity.measures.measure_set.MeasureSet*) – The base measures.

- **haz_id_dict** (*dict*) – Dictionary of the impact functions affected by uncertainty. Keys are hazard types (str), values are a list of impact function id (int).

Returns Entity uncertainty input variable

Return type *climada.engine.unsequa.input_var.InputVar*

`static entfut`(*impf_set*, *exp_list*, *meas_set*, *haz_id_dict*, *bounds_cost=None*, *bounds_eg=None*, *bounds_noise=None*, *bounds_impfi=None*, *bounds_mdd=None*, *bounds_paa=None*)

Helper wrapper for basic future entity set uncertainty input variable.

Important: only the impact function defined by haz_type and fun_id will be affected by bounds_impfi, bounds_mdd, bounds_paa.

The following types of uncertainties can be added: CO: scale the cost (homogeneously)

The cost of all measures is multiplied by the same number sampled uniformly from a distribution with (min, max) = bounds_cost

EG: scale the exposures growth (homogeneously) The value at each exposure point is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_eg

EN: mutliplicative noise (inhomogeneous) The value of each exposure point is independently multiplied by a random number sampled uniformly from a distribution with (min, max) = bounds_noise. EN is the value of the seed for the uniform random number generator.

EL: sample uniformly from exposure list From the provided list of exposure is elements are uniformly sampled. For example, LitPop instances with different exponents.

MDD: scale the mdd (homogeneously) The value of mdd at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_mdd

PAA: scale the paa (homogeneously) The value of paa at each intensity is multiplied by a number sampled uniformly from a distribution with (min, max) = bounds_paa

IFi: shift the impact function intensity (homogeneously) The value intensity are all summed with a random number sampled uniformly from a distribution with (min, max) = bounds_impfi

If a bounds is None, this parameter is assumed to have no uncertainty.

Parameters

- **bounds_cost** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous cost of all measures scaling. The default is None.
- **bounds_eg** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous total exposure growth scaling. The default is None.
- **bounds_noise** ((*float, float), optional*) – Bounds of the uniform distribution to scale each exposure point independently. The default is None.
- **bounds_mdd** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous mdd scaling. The default is None.
- **bounds_paa** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous paa scaling. The default is None.
- **bounds_impfi** ((*float, float), optional*) – Bounds of the uniform distribution for the homogeneous shift of intensity. The default is None.
- **impf_set** (*climada.engine.impact_funcs.impact_func_set.ImpactFuncSet*) – The base impact function set.

- **exp_list** (*[climada.entity.exposures.base.Exposure]*) – The list of base exposure. Can be one or many to uniformly sample from.
- **meas_set** (*climada.entity.measures.measure_set.MeasureSet*) – The base measures.
- **haz_id_dict** (*dict*) – Dictionary of the impact functions affected by uncertainty. Keys are hazard types (str), values are a list of impact function id (int).

Returns Entity uncertainty input variable

Return type *climada.engine.unsequa.input_var.InputVar*

climada.engine.unsequa.unc_output module

class climada.engine.unsequa.unc_output.**UncOutput**(*samples_df*, *unit=None*)

Bases: `object`

Class to store and plot uncertainty and sensitivity analysis output data

This is the base class to store uncertainty and sensitivity outputs of an analysis done on climada.engine.impact.Impact() or climada.engine.costbenefit.CostBenefit() object.

samples_df
Values of the sampled uncertainty parameters. It has n_samples rows and one column per uncertainty parameter.

Type pandas.DataFrame

sampling_method
Name of the sampling method from SAlib. https://salib.readthedocs.io/en/latest/api.html#

Type str

n_samples
Effective number of samples (number of rows of samples_df)

Type int

param_labels
Name of all the uncertainty parameters

Type list

distr_dict
Comon flattened dictionary of all the distr_dict of all input variables. It represents the distribution of all the uncertainty parameters.

Type dict

problem_sa
The description of the uncertainty variables and their distribution as used in SALib. https://salib.readthedocs.io/en/latest/basics.html.

Type dict

__init__(*samples_df*, *unit=None*)
Initialize Uncertainty Data object.

Parameters

- **samples_df** (*pandas.DataFrame*) – input parameters samples
- **unit** (*str, optional*) – value unit

get_samples_df()

get_unc_df(*metric_name*)

set_unc_df(*metric_name*, *unc_df*)

get_sens_df(*metric_name*)

set_sens_df(*metric_name*, *sens_df*)

check_salib(*sensitivity_method*)
Checks whether the chosen sensitivity method and the sampling method used to generated self.samples_df respect the pairing recommendation by the SALib package.

https://salib.readthedocs.io/en/latest/api.html

Parameters **sensitivity_method** (*str*) – Name of the sensitivity analysis method.

Returns True if sampling and sensitivity methods respect the recommended pairing.

Return type bool

property sampling_method
Returns the sampling method used to generate self.samples_df

Returns Sampling method name

Return type str

property sampling_kwargs
Returns the kwargs of the sampling method that generate self.samples_df

Returns Dictionary of arguments for SALib sampling method

Return type dict

property n_samples
The effective number of samples

Returns effective number of samples

Return type int

property param_labels
Labels of all uncertainty input parameters.

Returns Labels of all uncertainty input parameters.

Return type list of str

property problem_sa
The description of the uncertainty variables and their distribution as used in SALib. https://salib.readthedocs.io/en/latest/basics.html

Returns Salib problem dictionary.

Return type dict

property uncertainty_metrics
Retrieve all uncertainty output metrics names

Returns **unc_metric_list** – List of names of attributes containing metrics uncertainty values, without the trailing '_unc_df'

Return type [str]

property sensitivity_metrics
Retrieve all sensitivity output metrics names

Returns **sens_metric_list** – List of names of attributes containing metrics sensitivity values, without the trailing '_sens_df'

Return type [str]

`get_uncertainty`(*metric_list=None*)
Returns uncertainty dataframe with values for each sample

Parameters **metric_list** (*[str], optional*) – List of uncertainty metrics to consider. The default returns all uncertainty metrics at once.

Returns Joint dataframe of all uncertainty values for all metrics in the metric_list.

Return type pandas.DataFrame

See also:

uncertainty_metrics list of all available uncertainty metrics

`get_sensitivity`(*salib_si*, *metric_list=None*)
Returns sensitivity index

E.g. For the sensitivity analysis method 'sobol', the choices are ['S1', 'ST'], for 'delta' the choices are ['delta', 'S1'].

For more information see the SAlib documentation: https://salib.readthedocs.io/en/latest/basics.html

Parameters

- **salib_si** (*str*) – Sensitivity index
- **metric_list** (*[str], optional*) – List of sensitivity metrics to consider. The default returns all sensitivity indices at once.

Returns Joint dataframe of the sensitvity indices for all metrics in the metric_list

Return type pandas.DataFrame

See also:

`sensitvity_metrics` list of all available sensitivity metrics

`get_largest_si`(*salib_si*, *metric_list=None*, *threshold=0.01*)
Get largest si per metric

Parameters

- **salib_si** (*str*) – The name of the sensitivity index to plot.
- **metric_list** (*list of strings, optional*) – List of metrics to plot the sensitivity. Default is None.
- **threshold** (*float*) – The minimum value a sensitivity index must have to be considered as the largest. The default is 0.01.

Returns **max_si_df** – Dataframe with the largest si and its value per metric

Return type pandas.dataframe

`plot_sample`(*figsize=None*)
Plot the sample distributions of the uncertainty input parameters

For each uncertainty input variable, the sample distributions is shown in a separate axes.

Parameters **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is derived from the total number of plots (nplots) as:

nrows, ncols = int(np.ceil(nplots / 3)), min(nplots, 3) figsize = (ncols * FIG_W, nrows * FIG_H)

Raises `ValueError` – If no sample was computed the plot cannot be made.

Returns **axes** – The axis handle of the plot.

Return type matplotlib.pyplot.axes

`plot_uncertainty`(*metric_list=None*, *orig_list=None*, *figsize=None*, *log=False*, *axes=None*)
Plot the uncertainty distribution

For each risk metric, a separate axes is used to plot the uncertainty distribution of the output values obtained over the sampled input parameters.

Parameters

- **metric_list** (*list[str], optional*) – List of metrics to plot the distribution. The default is None.
- **orig_list** (*list[float], optional*) – List of the original (without uncertainty) values for each sub-metric of the mtrics in metric_list. The ordering is identical. The default is None.
- **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is derived from the total number of plots (nplots) as: nrows, ncols = int(np.ceil(nplots / 3)), min(nplots, 3) figsize = (ncols * FIG_W, nrows * FIG_H)
- **log** (*boolean, optional*) – Use log10 scale for x axis. Default is False.
- **axes** (*matplotlib.pyplot.axes, optional*) – Axes handles to use for the plot. The default is None.

Raises `ValueError` – If no metric distribution was computed the plot cannot be made.

Returns **axes** – The axes handle of the plot.

Return type matplotlib.pyplot.axes

See also:

uncertainty_metrics list of all available uncertainty metrics

`plot_rp_uncertainty`(*orig_list=None*, *figsize=(16, 6)*, *axes=None*)
Plot the distribution of return period uncertainty

Parameters

- **orig_list** (*list[float], optional*) – List of the original (without uncertainty) values for each sub-metric of the mtrics in metric_list. The ordering is identical. The default is None.
- **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is (8, 6)
- **axes** (*matplotlib.pyplot.axes, optional*) – Axes handles to use for the plot. The default is None.

Raises `ValueError` – If no metric distribution was computed the plot cannot be made.

Returns **ax** – The axis handle of the plot.

Return type matplotlib.pyplot.axes

plot_sensitivity(*salib_si='S1'*, *salib_si_conf='S1_conf'*, *metric_list=None*, *figsize=None*, *axes=None*, ***kwargs*)

Bar plot of a first order sensitivity index

For each metric, the sensitivity indices are plotted in a separate axes.

This requires that a senstivity analysis was already performed.

E.g. For the sensitivity analysis method 'sobol', the choices are ['S1', 'ST'], for 'delta' the choices are ['delta', 'S1'].

Note that not all sensitivity indices have a confidence interval.

For more information see the SAlib documentation: https://salib.readthedocs.io/en/latest/basics.html

Parameters

- **salib_si** (*string, optional*) – The first order (one value per metric output) sensitivity index to plot. The default is S1.
- **salib_si_conf** (*string, optional*) – The confidence value for the first order sensitivity index to plot. The default is S1_conf.
- **metric_list** (*list of strings, optional*) – List of metrics to plot the sensitivity. If a metric is not found it is ignored.
- **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is derived from the total number of plots (nplots) as:

  nrows, ncols = int(np.ceil(nplots / 3)), min(nplots, 3) figsize = (ncols * FIG_W, nrows * FIG_H)
- **axes** (*matplotlib.pyplot.axes, optional*) – Axes handles to use for the plot. The default is None.
- **kwargs** – Keyword arguments passed on to pandas.DataFrame.plot(kind='bar')

Raises **ValueError :** – If no sensitivity is available the plot cannot be made.

Returns **axes** – The axes handle of the plot.

Return type matplotlib.pyplot.axes

See also:

sensitvity_metrics list of all available sensitivity metrics

plot_sensitivity_second_order(*salib_si='S2'*, *salib_si_conf='S2_conf'*, *metric_list=None*, *figsize=None*, *axes=None*, ***kwargs*)

Plot second order sensitivity indices as matrix.

For each metric, the sensitivity indices are plotted in a separate axes.

E.g. For the sensitivity analysis method 'sobol', the choices are ['S2', 'S2_conf'].

Note that not all sensitivity indices have a confidence interval.

For more information see the SAlib documentation: https://salib.readthedocs.io/en/latest/basics.html

Parameters

- **salib_si** (*string, optional*) – The second order sensitivity indexto plot. The default is S2.
- **salib_si_conf** (*string, optional*) – The confidence value for thesensitivity index salib_si to plot. The default is S2_conf.

- **metric_list** (*list of strings, optional*) – List of metrics to plot the sensitivity. If a metric is not found it is ignored. Default is all 1D metrics.
- **figsize** (*tuple(int or float, int or float), optional*) – The figsize argument of matplotlib.pyplot.subplots() The default is derived from the total number of plots (nplots) as:

  nrows, ncols = int(np.ceil(nplots / 3)), min(nplots, 3) figsize = (ncols * 5, nrows * 5)
- **axes** (*matplotlib.pyplot.axes, optional*) – Axes handles to use for the plot. The default is None.
- **kwargs** – Keyword arguments passed on to matplotlib.pyplot.imshow()

Raises `ValueError :` – If no sensitivity is available the plot cannot be made.

Returns **axes** – The axes handle of the plot.

Return type matplotlib.pyplot.axes

See also:

`sensitvity_metrics` list of all available sensitivity metrics

`plot_sensitivity_map`(*salib_si='S1'*, ***kwargs*)

Plot a map of the largest sensitivity index in each exposure point

Requires the uncertainty distribution for eai_exp.

Parameters

- **salib_si** (*str, optional*) – The name of the sensitivity index to plot. The default is 'S1'.
- **kwargs** – Keyword arguments passed on to climada.util.plot.geo_scatter_categorical

Raises `ValueError :` – If no sensitivity data is found, raise error.

Returns **ax** – The axis handle of the plot.

Return type matplotlib.pyplot.axes

See also:

`climada.util.plot.geo_scatter_categorical` geographical plot for categorical variable

`to_hdf5`(*filename=None*)

Save output to .hdf5

Parameters **filename** (*str or pathlib.Path, optional*) – The filename with absolute or relative path. The default name is "unc_output + datetime.now() + .hdf5" and the default path is taken from climada.config

Returns **save_path** – Path to the saved file

Return type pathlib.Path

`static from_hdf5`(*filename*)

Load a uncertainty and uncertainty output data from .hdf5 file

Parameters **filename** (*str or pathlib.Path*) – The filename with absolute or relative path.

Returns **unc_output** – Uncertainty and sensitivity data loaded from .hdf5 file.

Return type climada.engine.uncertainty.unc_output.UncOutput

class climada.engine.unsequa.unc_output.**UncCostBenefitOutput**(*samples_df*, *unit*, *imp_meas_present_unc_df*, *imp_meas_future_unc_df*, *tot_climate_risk_unc_df*, *benefit_unc_df*, *cost_ben_ratio_unc_df*, *cost_benefit_kwargs*)

Bases: climada.engine.unsequa.unc_output.UncOutput

Extension of UncOutput specific for CalcCostBenefit, returned by the uncertainty() method.

__init__(*samples_df*, *unit*, *imp_meas_present_unc_df*, *imp_meas_future_unc_df*, *tot_climate_risk_unc_df*, *benefit_unc_df*, *cost_ben_ratio_unc_df*, *cost_benefit_kwargs*)

Constructor

Uncertainty output values from cost_benefit.calc for each sample

Parameters

- **samples_df** (*pandas.DataFrame*) – input parameters samples
- **unit** (*str*) – value unit
- **imp_meas_present_unc_df** (*pandas.DataFrame*) – Each row contains the values of imp_meas_present for one sample (row of samples_df)
- **imp_meas_future_unc_df** (*pandas.DataFrame*) – Each row contains the values of imp_meas_future for one sample (row of samples_df)
- **tot_climate_risk_unc_df** (*pandas.DataFrame*) – Each row contains the values of tot_cliamte_risk for one sample (row of samples_df)
- **benefit_unc_df** (*pandas.DataFrame*) – Each row contains the values of benefit for one sample (row of samples_df)
- **cost_ben_ratio_unc_df** (*pandas.DataFrame*) – Each row contains the values of cost_ben_ratio for one sample (row of samples_df)
- **cost_benefit_kwargs** (*pandas.DataFrame*) – Each row contains the value of cost_benefit for one sample (row of samples_df)

class climada.engine.unsequa.unc_output.**UncImpactOutput**(*samples_df*, *unit*, *aai_agg_unc_df*, *freq_curve_unc_df*, *eai_exp_unc_df*, *at_event_unc_df*, *tot_value_unc_df*, *coord_df*)

Bases: climada.engine.unsequa.unc_output.UncOutput

Extension of UncOutput specific for CalcImpact, returned by the uncertainty() method.

__init__(*samples_df*, *unit*, *aai_agg_unc_df*, *freq_curve_unc_df*, *eai_exp_unc_df*, *at_event_unc_df*, *tot_value_unc_df*, *coord_df*)

Constructor

Uncertainty output values from impact.calc for each sample

Parameters

- **samples_df** (*pandas.DataFrame*) – input parameters samples
- **unit** (*str*) – value unit
- **aai_agg_unc_df** (*pandas.DataFrame*) – Each row contains the value of aai_aag for one sample (row of samples_df)

- **freq_curve_unc_df** (*pandas.DataFrame*) – Each row contains the values of the impact exceedence frequency curve for one sample (row of samples_df)
- **eai_exp_unc_df** (*pandas.DataFrame*) – Each row contains the values of eai_exp for one sample (row of samples_df)
- **at_event_unc_df** (*pandas.DataFrame*) – Each row contains the values of at_event for one sample (row of samples_df)
- **tot_value_unc_df** (*pandas.DataFrame*) – Each row contains the value of tot_value for one sample (row of samples_df)
- **coord_df** (*pandas.DataFrame*) – Coordinates of the exposure

climada.engine.calibration_opt module

climada.engine.calibration_opt.**calib_instance**(*hazard*, *exposure*, *impact_func*, *df_out=Empty DataFrame Columns: [] Index: []*, *yearly_impact=False*, *return_cost='False'*)

calculate one impact instance for the calibration algorithm and write to given DataFrame

Parameters

- **hazard** (*Hazard*)
- **exposure** (*Exposure*)
- **impact_func** (*ImpactFunc*)
- **df_out** (*Dataframe, optional*) – Output DataFrame with headers of columns defined and optionally with first row (index=0) defined with values. If columns "impact", "event_id", or "year" are not included, they are created here. Data like reported impacts or impact function parameters can be given here; values are preserved.
- **yearly_impact** (*boolean, optional*) – if set True, impact is returned per year, not per event
- **return_cost** (*str, optional*) – if not 'False' but any of 'R2', 'logR2', cost is returned instead of df_out

Returns **df_out** – DataFrame with modelled impact written to rows for each year or event.

Return type DataFrame

climada.engine.calibration_opt.**init_impf**(*impf_name_or_instance, param_dict, df_out=Empty DataFrame Columns: [] Index: [0]*)

create an ImpactFunc based on the parameters in param_dict using the method specified in impf_parameterisation_name and document it in df_out.

Parameters

- **impf_name_or_instance** (*str or ImpactFunc*) – method of impact function parameterisation e.g. 'emanuel' or an instance of ImpactFunc
- **param_dict** (*dict, optional*) – dict of parameter_names and values e.g. {'v_thresh': 25.7, 'v_half': 70, 'scale': 1} or {'mdd_shift': 1.05, 'mdd_scale': 0.8, 'paa_shift': 1, paa_scale': 1}

Returns

- **imp_fun** (*ImpactFunc*) – The Impact function based on the parameterisation

- **df_out** (*DataFrame*) – Output DataFrame with headers of columns defined and with first row (index=0) defined with values. The impact function parameters from param_dict are represented here.

climada.engine.calibration_opt.**change_impf**(*impf_instance*, *param_dict*)

apply a shifting or a scaling defined in param_dict to the impact function in impf_istance and return it as a new ImpactFunc object.

Parameters

- **impf_instance** (*ImpactFunc*) – an instance of ImpactFunc
- **param_dict** (*dict*) – dict of parameter_names and values (interpreted as factors, 1 = neutral) e.g. {'mdd_shift': 1.05, 'mdd_scale': 0.8, 'paa_shift': 1, paa_scale': 1}

Returns **ImpactFunc**

Return type The Impact function based on the parameterisation

climada.engine.calibration_opt.**init_impact_data**(*hazard_type*, *region_ids*, *year_range*, *source_file*, *reference_year*, *impact_data_source='emdat'*, *yearly_impact=True*)

creates a dataframe containing the recorded impact data for one hazard type and one area (countries, country or local split)

Parameters

- **hazard_type** (*str*) – default = 'TC', type of hazard 'WS','FL' etc.
- **region_ids** (*str*) – name the region_ids or country names
- **year_range** (*list*) – list containting start and end year. e.g. [1980, 2017]
- **source_file** (*str*)
- **reference_year** (*int*) – impacts will be scaled to this year
- **impact_data_source** (*str, optional*) – default 'emdat', others maybe possible
- **yearly_impact** (*bool, optional*) – if set True, impact is returned per year, not per event

Returns **df_out** – Dataframe with recorded impact written to rows for each year or event.

Return type pd.DataFrame

climada.engine.calibration_opt.**calib_cost_calc**(*df_out*, *cost_function*)

calculate the cost function of the modelled impact impact_CLIMADA and the reported impact impact_scaled in df_out

Parameters

- **df_out** (*pd.Dataframe*) – DataFrame as created in calib_instance
- **cost_function** (*str*) – chooses the cost function e.g. 'R2' or 'logR2'

Returns **cost** – The results of the cost function when comparing modelled and reported impact

Return type float

climada.engine.calibration_opt.**calib_all**(*hazard*, *exposure*, *impf_name_or_instance*, *param_full_dict*, *impact_data_source*, *year_range*, *yearly_impact=True*)

portrait the difference between modelled and reported impacts for all impact functions described in param_full_dict and impf_name_or_instance

Parameters

- **hazard** (*list or Hazard*)
- **exposure** (*list or Exposures*) – list or instance of exposure of full countries
- **impf_name_or_instance** (*string or ImpactFunc*) – the name of a parameterisation or an instance of class ImpactFunc e.g. 'emanuel'
- **param_full_dict** (*dict*) – a dict containing keys used for f_name_or_instance and values which are iterable (lists) e.g. {'v_thresh' : [25.7, 20], 'v_half': [70], 'scale': [1, 0.8]}
- **impact_data_source** (*dict or pd.Dataframe*) – with name of impact data source and file location or dataframe
- **year_range** (*list*)
- **yearly_impact** (*bool, optional*)

Returns **df_result** – df with modelled impact written to rows for each year or event.

Return type pd.DataFrame

climada.engine.calibration_opt.**calib_optimize**(*hazard*, *exposure*, *impf_name_or_instance*, *param_dict*, *impact_data_source*, *year_range*, *yearly_impact=True*, *cost_fucntion='R2'*, *show_details=False*)

portrait the difference between modelled and reported impacts for all impact functions described in param_full_dict and impf_name_or_instance

Parameters

- **hazard** (*list or Hazard*)
- **exposure** (*list or Exposures*) – list or instance of exposure of full countries
- **impf_name_or_instance** (*string or ImpactFunc*) – the name of a parameterisation or an instance of class ImpactFunc e.g. 'emanuel'
- **param_dict** (*dict*) – a dict containing keys used for impf_name_or_instance and one set of values e.g. {'v_thresh': 25.7, 'v_half': 70, 'scale': 1}
- **impact_data_source** (*dict or pd. dataframe*) – with name of impact data source and file location or dataframe
- **year_range** (*list*)
- **yearly_impact** (*bool, optional*)
- **cost_function** (*str, optional*) – the argument for function calib_cost_calc, default 'R2'
- **show_details** (*bool, optional*) – if True, return a tuple with the parameters AND the details of the optimization like success, status, number of iterations etc

Returns **param_dict_result** – the parameters with the best calibration results (or a tuple with (1) the parameters and (2) the optimization output)

Return type dict or tuple

climada.engine.cost_benefit module

class climada.engine.cost_benefit.**CostBenefit**
Bases: object

Impact definition. Compute from an entity (exposures and impact functions) and hazard.

present_year
present reference year

Type int

future_year
future year

Type int

tot_climate_risk
total climate risk without measures

Type float

unit
unit used for impact

Type str

color_rgb
color code RGB for each measure.

Type dict

Key
measure name ('no measure' used for case without measure),

Type str

Value

Type np.array

benefit
benefit of each measure. Key: measure name, Value: float benefit

Type dict

cost_ben_ratio
cost benefit ratio of each measure. Key: measure name, Value: float cost benefit ratio

Type dict

imp_meas_future
impact of each measure at future or default. Key: measure name ('no measure' used for case without measure), Value: dict with: 'cost' (tuple): (cost measure, cost factor insurance), 'risk' (float): risk measurement, 'risk_transf' (float): annual expected risk transfer, 'efc' (ImpactFreqCurve): impact exceedance freq (optional) 'impact' (Impact): impact instance

Type dict

imp_meas_present
impact of each measure at present. Key: measure name ('no measure' used for case without measure), Value: dict with: 'cost' (tuple): (cost measure, cost factor insurance), 'risk' (float): risk measurement, 'risk_transf' (float): annual expected risk transfer, 'efc' (ImpactFreqCurve): impact exceedance freq (optional) 'impact' (Impact): impact instance

Type dict

`__init__`()
Initilization

`calc`(*hazard*, *entity*, *haz_future=None*, *ent_future=None*, *future_year=None*, *risk_func=<function risk_aai_agg>*, *imp_time_depen=None*, *save_imp=False*)
Compute cost-benefit ratio for every measure provided current and, optionally, future conditions. Present and future measures need to have the same name. The measures costs need to be discounted by the user. If future entity provided, only the costs of the measures of the future and the discount rates of the present will be used.

Parameters

- **hazard** (*climada.Hazard*)
- **entity** (*climada.entity*)
- **haz_future** (*climada.Hazard, optional*) – hazard in the future (future year provided at ent_future)
- **ent_future** (*Entity, optional*) – entity in the future. Default is None
- **future_year** (*int, optional*) – future year to consider if no ent_future. Default is None provided. The benefits are added from the entity.exposures.ref_year until ent_future.exposures.ref_year, or until future_year if no ent_future given. Default: entity.exposures.ref_year+1
- **risk_func** (*func optional*) – function describing risk measure to use to compute the annual benefit from the Impact. Default: average annual impact (aggregated).
- **imp_time_depen** (*float, optional*) – parameter which represents time evolution of impact (super- or sublinear). If None: all years count the same when there is no future hazard nor entity and 1 (linear annual change) when there is future hazard or entity. Default is None.
- **save_imp** (*bool, optional*))
- **True if Impact of each measure is saved. Default is False.**

`combine_measures`(*in_meas_names*, *new_name*, *new_color*, *disc_rates*, *imp_time_depen=None*, *risk_func=<function risk_aai_agg>*)
Compute cost-benefit of the combination of measures previously computed by calc with save_imp=True. The benefits of the measures per event are added. To combine with risk transfer options use apply_risk_transfer.

Parameters

- **in_meas_names** (*list(str)*)
- **list with names of measures to combine**
- **new_name** (*str*) – name to give to the new resulting measure new_color (np.array): color code RGB for new measure, e.g. np.array([0.1, 0.1, 0.1])
- **disc_rates** (*DiscRates*) – discount rates instance
- **imp_time_depen** (*float, optional*) – parameter which represents time evolution of impact (super- or sublinear). If None: all years count the same when there is no future hazard nor entity and 1 (linear annual change) when there is future hazard or entity. Default is None.
- **risk_func** (*func, optional*) – function describing risk measure given an Impact. Default: average annual impact (aggregated).

Return type climada.CostBenefit

apply_risk_transfer(*meas_name*, *attachment*, *cover*, *disc_rates*, *cost_fix=0*, *cost_factor=1*, *imp_time_depen=None*, *risk_func=<function risk_aai_agg>*)

Applies risk transfer to given measure computed before with saved impact and compares it to when no measure is applied. Appended to dictionaries of measures.

Parameters

- **meas_name** (*str*) – name of measure where to apply risk transfer
- **attachment** (*float*) – risk transfer values attachment (deductible)
- **cover** (*float*) – risk transfer cover
- **cost_fix** (*float*) – fixed cost of implemented innsurance, e.g. transaction costs
- **cost_factor** (*float, optional*) – factor to which to multiply the insurance layer to compute its cost. Default is 1
- **imp_time_depen** (*float, optional*) – parameter which represents time evolution of impact (super- or sublinear). If None: all years count the same when there is no future hazard nor entity and 1 (linear annual change) when there is future hazard or entity. Default is None.
- **risk_func** (*func, optional*) – function describing risk measure given an Impact. Default: average annual impact (aggregated).

remove_measure(*meas_name*)

Remove computed values of given measure

Parameters **meas_name** (*str*) – name of measure to remove

plot_cost_benefit(*cb_list=None*, *axis=None*, ***kwargs*)

Plot cost-benefit graph. Call after calc().

Parameters

- **cb_list** (*list(CostBenefit), optional*) – if other CostBenefit provided, overlay them all. Used for uncertainty visualization.
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for Rectangle matplotlib, e.g. alpha=0.5 (color is set by measures color attribute)

Return type matplotlib.axes._subplots.AxesSubplot

plot_event_view(*return_per=(10, 25, 100)*, *axis=None*, ***kwargs*)

Plot averted damages for return periods. Call after calc().

Parameters

- **return_per** (*list, optional*) – years to visualize. Default 10, 25, 100
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for bar matplotlib function, e.g. alpha=0.5 (color is set by measures color attribute)

Return type matplotlib.axes._subplots.AxesSubplot

static plot_waterfall(*hazard*, *entity*, *haz_future*, *ent_future*, *risk_func=<function risk_aai_agg>*, *axis=None*, ***kwargs*)

Plot waterfall graph at future with given risk metric. Can be called before and after calc().

Parameters

- **hazard** (*climada.Hazard*)

- **entity** (*climada.Entity*)
- **haz_future** (*Hazard*) – hazard in the future (future year provided at ent_future)
- **ent_future** (*climada.Entity*) – entity in the future
- **risk_func** (*func, optional*) – function describing risk measure given an Impact. Default: average annual impact (aggregated).
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for bar matplotlib function, e.g. alpha=0.5

Return type matplotlib.axes._subplots.AxesSubplot

`plot_arrow_averted`(*axis*, *in_meas_names=None*, *accumulate=False*, *combine=False*, *risk_func=<function risk_aai_agg>*, *disc_rates=None*, *imp_time_depen=1*, ***kwargs*)

Plot waterfall graph with accumulated values from present to future year. Call after calc() with save_imp=True.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot*) – axis from plot_waterfall or plot_waterfall_accumulated where arrow will be added to last bar
- **in_meas_names** (*list(str), optional*) – list with names of measures to represented total averted damage. Default: all measures
- **accumulate** (*bool, optional)*) – accumulated averted damage (True) or averted damage in future (False). Default: False
- **combine** (*bool, optional*) – use combine_measures to compute total averted damage (True) or just add benefits (False). Default: False
- **risk_func** (*func, optional*) – function describing risk measure given an Impact used in combine_measures. Default: average annual impact (aggregated).
- **disc_rates** (*DiscRates, optional*) – discount rates used in combine_measures
- **imp_time_depen** (*float, optional*) – parameter which represent time evolution of impact used in combine_measures. Default: 1 (linear).
- **kwargs** (*optional*) – arguments for bar matplotlib function, e.g. alpha=0.5

`plot_waterfall_accumulated`(*hazard*, *entity*, *ent_future*, *risk_func=<function risk_aai_agg>*, *imp_time_depen=1*, *axis=None*, ***kwargs*)

Plot waterfall graph with accumulated values from present to future year. Call after calc() with save_imp=True. Provide same inputs as in calc.

Parameters

- **hazard** (*climada.Hazard*)
- **entity** (*climada.Entity*)
- **ent_future** (*climada.Entity*) – entity in the future
- **risk_func** (*func, optional*) – function describing risk measure given an Impact. Default: average annual impact (aggregated).
- **imp_time_depen** (*float, optional*) – parameter which represent time evolution of impact used in combine_measures. Default: 1 (linear).
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use

- **kwargs** (*optional*) – arguments for bar matplotlib function, e.g. alpha=0.5

Return type matplotlib.axes._subplots.AxesSubplot

climada.engine.cost_benefit.**risk_aai_agg**(*impact*)
Risk measurement as average annual impact aggregated.

Parameters **impact** (*climada.engine.Impact*) – an Impact instance

Return type float

climada.engine.cost_benefit.**risk_rp_100**(*impact*)
Risk measurement as exceedance impact at 100 years return period.

Parameters **impact** (*climada.engine.Impact*) – an Impact instance

Return type float

climada.engine.cost_benefit.**risk_rp_250**(*impact*)
Risk measurement as exceedance impact at 250 years return period.

Parameters **impact** (*climada.engine.Impact*) – an Impact instance

Return type float

climada.engine.forecast module

class climada.engine.forecast.**Forecast**(*hazard_dict*, *exposure*, *impact_funcs*, *haz_model='NWP'*, *exposure_name=None*)

Bases: object

Forecast definition. Compute an impact forecast with predefined hazard originating from a forecast (like numerical weather prediction models), exposure and impact. Use the calc() method to calculate a forecasted impact. Then use the plotting methods to illustrate the forecasted impacts. By default plots are saved under in a '/forecast/plots' folder in the configurable save_dir in local_data (see climada.util.config) under a name summarizing the Hazard type, haz model name, initialization time of the forecast run, event date, exposure name and the plot title. As the class is relatively new, there might be future changes to the attributes, the methods, and the parameters used to call the methods. It was discovered at some point, that there might be a memory leak in matplotlib even when figures are closed (https://github.com/matplotlib/matplotlib/issues/8519). Due to this reason the plotting functions in this module have the flag close_fig, to close figures within the function scope, which might mitigate that problem if a script runs this plotting functions many times.

run_datetime
initialization time of the forecast model run used to create the Hazard

Type list of datetime.datetime

event_date
Date on which the Hazard event takes place

Type datetime.datetime

hazard
List of the hazard forecast with different lead times.

Type list of CLIMADA Hazard

haz_model
Short string specifying the model used to create the hazard, if possible three big letters.

Type str

`exposure`
an CLIMADA Exposures containg values at risk

Type Exposure

`exposure_name`
string specifying the exposure (e.g. 'EU'), which is used to name output files.

Type str

`vulnerability`
Set of impact functions used in the impact calculation.

Type *ImpactFuncSet*

`__init__`(*hazard_dict*, *exposure*, *impact_funcs*, *haz_model='NWP'*, *exposure_name=None*)
Initialization with hazard, exposure and vulnerability.

Parameters

- **hazard_dict** (*dict*) – Dictionary of the format {run_datetime: Hazard} with run_datetime being the initialization time of a weather forecast run and Hazard being a CLIMADA Hazard derived from that forecast for one event. A probabilistic representation of that one event is possible, as long as the attribute Hazard.date is the same for all events. Several run_datetime:Hazard combinations for the same event can be provided.
- **exposure** (*Exposure*)
- **impact_funcs** (*ImpactFuncSet*)
- **haz_model** (*str, optional*) – Short string specifying the model used to create the hazard, if possible three big letters. Default is 'NWP' for numerical weather prediction.
- **exposure_name** (*str, optional*) – string specifying the exposure (e.g. 'EU'), which is used to name output files.

`ei_exp`(*run_datetime=None*)
Expected impact per exposure

Parameters **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.

Return type float

`ai_agg`(*run_datetime=None*)
average impact aggregated over all exposures

Parameters **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.

Return type float

`haz_summary_str`(*run_datetime=None*)
provide a summary string for the hazard part of the forecast

Parameters **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.

Returns summarizing the most important information about the hazard

Return type str

`summary_str`(*run_datetime=None*)
provide a summary string for the impact forecast

Parameters **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.

Returns summarizing the most important information about the impact forecast

Return type str

`lead_time`(*run_datetime=None*)

provide the lead time for the impact forecast

Parameters **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.

Returns the difference between the initialization time of the forecast model run and the date of the event, commenly named lead time

Return type datetime.timedelta

`calc`(*force_reassign=False*)

calculate the impacts for all lead times using exposure, all hazards of all run_datetime, and ImpactFunctionSet.

Parameters **force_reassign** (*bool, optional*) – Reassign hazard centroids to the exposure for all hazards, default is false.

`plot_imp_map`(*run_datetime=None*, *save_fig=True*, *close_fig=False*, *polygon_file=None*, *polygon_file_crs='epsg:4326'*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *figsize=(9*, *13)*, *adapt_fontsize=True*)

plot a map of the impacts

Parameters

- **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.
- **save_fig** (*bool, optional*) – Figure is saved if True, folder is within your configurable save_dir and filename is derived from the method summary_str() (for more details see class docstring). Default is True.
- **close_fig** (*bool, optional*) – Figure not drawn if True. Default is False.
- **polygon_file** (*str, optional*) – Points to a .shp-file with polygons do be drawn as outlines on the plot, default is None to not draw the lines. please also specify the crs in the parameter polygon_file_crs.
- **polygon_file_crs** (*str, optional*) – String of pattern <provider>:<code> specifying the crs. has to be readable by pyproj.Proj. Default is 'epsg:4326'.
- **proj** (*ccrs*) – coordinate reference system used in coordinates The default is ccrs.PlateCarree()
- **figsize** (*tuple*) – figure size for plt.subplots, width, height in inches The default is (9, 13)
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Returns **axes**

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_hist`(*run_datetime=None*, *save_fig=True*, *close_fig=False*, *figsize=(9, 8)*)
plot histogram of the forecasted impacts all ensemble members

Parameters

- **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.
- **save_fig** (*bool, optional*) – Figure is saved if True, folder is within your configurable save_dir and filename is derived from the method summary_str() (for more details see class docstring). Default is True.
- **close_fig** (*bool, optional*) – Figure is not drawn if True. Default is False.
- **figsize** (*tuple*) – figure size for plt.subplots, width, height in inches The default is (9, 8)

Returns axes

Return type matplotlib.axes.Axes

`plot_exceedence_prob`(*threshold*, *explain_str=None*, *run_datetime=None*, *save_fig=True*, *close_fig=False*, *polygon_file=None*, *polygon_file_crs='epsg:4326'*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *figsize=(9, 13)*, *adapt_fontsize=True*)
plot exceedence map

Parameters

- **threshold** (*float*) – Threshold of impact unit for which exceedence probability should be plotted.
- **explain_str** (*str, optional*) – Short str which explains threshold, explain_str is included in the title of the figure.
- **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.
- **save_fig** (*bool, optional*) – Figure is saved if True, folder is within your configurable save_dir and filename is derived from the method summary_str() (for more details see class docstring). Default is True.
- **close_fig** (*bool, optional*) – Figure not drawn if True. Default is False.
- **polygon_file** (*str, optional*) – Points to a .shp-file with polygons do be drawn as outlines on the plot, default is None to not draw the lines. please also specify the crs in the parameter polygon_file_crs.
- **polygon_file_crs** (*str, optional*) – String of pattern <provider>:<code> specifying the crs. has to be readable by pyproj.Proj. Default is 'epsg:4326'.
- **proj** (*ccrs*) – coordinate reference system used in coordinates The default is ccrs.PlateCarree()
- **figsize** (*tuple*) – figure size for plt.subplots, width, height in inches The default is (9, 13)
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Returns axes

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_warn_map`(*polygon_file=None*, *polygon_file_crs='epsg:4326'*, *thresholds='default'*, *decision_level='exposure_point'*, *probability_aggregation=0.5*, *area_aggregation=0.5*, *title='WARNINGS'*, *explain_text='warn level based on thresholds'*, *run_datetime=None*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *figsize=(9, 13)*, *save_fig=True*, *close_fig=False*, *adapt_fontsize=True*)

plot map colored with 5 warning colors for all regions in provided shape file.

Parameters

- **polygon_file** (*str, optional*) – path to shp-file containing warning region polygons
- **polygon_file_crs** (*str, optional*) – String of pattern <provider>:<code> specifying the crs. has to be readable by pyproj.Proj. Default is 'epsg:4326'.
- **thresholds** (*list of 4 floats, optional*) – Thresholds for coloring region in second, third, forth and fifth warning color.
- **decision_level** (*str, optional*) – Either 'exposure_point' or 'polygon'. Default value is 'exposure_point'.
- **probability_aggregation** (*float or str, optional*) – Either a float between [0..1] spezifying a quantile or 'mean' or 'sum'. Default value is 0.5.
- **area_aggregation** (*float or str.*) – Either a float between [0..1] specifying a quantile or 'mean' or 'sum'. Default value is 0.5.
- **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.
- **title** (*str, optional*) – Default is 'WARNINGS'.
- **explain_text** (*str, optional*) – Defaut is 'warn level based on thresholds'.
- **proj** (*ccrs*) – coordinate reference system used in coordinates
- **figsize** (*tuple*) – figure size for plt.subplots, width, height in inches The default is (9, 13)
- **save_fig** (*bool, optional*) – Figure is saved if True, folder is within your configurable save_dir and filename is derived from the method summary_str() (for more details see class docstring). Default is True.
- **close_fig** (*bool, optional*) – Figure is not drawn if True. The default is False.
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Returns axes

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_hexbin_ei_exposure`(*run_datetime=None*, *figsize=(9, 13)*)

plot the expected impact

Parameters

- **run_datetime** (*datetime.datetime, optional*) – Select the used hazard by the run_datetime, default is first element of attribute run_datetime.
- **figsize** (*tuple*) – figure size for plt.subplots, width, height in inches The default is (9, 13)

Returns axes

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

climada.engine.impact module

class climada.engine.impact.**ImpactFreqCurve**

Bases: object

Impact exceedence frequency curve.

tag

dictionary of tags of exposures, impact functions set and hazard: {'exp': Tag(), 'impf_set': Tag(), 'haz': TagHazard()}

Type dict

return_per

return period

Type np.array

impact

impact exceeding frequency

Type np.array

unit

value unit used (given by exposures unit)

Type str

label

string describing source data

Type str

__init__()

plot(*axis=None*, *log_frequency=False*, ***kwargs*)

Plot impact frequency curve.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **log_frequency** (*boolean, optional*) – plot logarithmioc exceedance frequency on x-axis
- **kwargs** (*optional*) – arguments for plot matplotlib function, e.g. color='b'

Return type matplotlib.axes._subplots.AxesSubplot

class climada.engine.impact.**Impact**

Bases: object

Impact definition. Compute from an entity (exposures and impact functions) and hazard.

tag

dictionary of tags of exposures, impact functions set and hazard: {'exp': Tag(), 'impf_set': Tag(), 'haz': TagHazard()}

Type dict

event_id

np.array id (>0) of each hazard event

event_name
list name of each hazard event

date
date if events as integer date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1 (ordinal format of datetime library)

Type np.array

coord_exp
exposures coordinates [lat, lon] (in degrees)

Type np.array

eai_exp
expected annual impact for each exposure

Type np.array

at_event
impact for each hazard event

Type np.array

frequency
annual frequency of event

Type np.array

tot_value
total exposure value affected

Type float

aai_agg
average annual impact (aggregated)

Type float

unit
value unit used (given by exposures unit)

Type str

imp_mat
matrix num_events x num_exp with impacts. only filled if save_mat is True in calc()

Type sparse.csr_matrix

__init__()
Empty initialization.

calc_freq_curve(*return_per=None*)
Compute impact exceedance frequency curve.

Parameters **return_per** (*np.array, optional*) – return periods where to compute the exceedance impact. Use impact's frequencies if not provided

Return type *ImpactFreqCurve*

calc(*exposures*, *impact_funcs*, *hazard*, *save_mat=False*)
Compute impact of an hazard to exposures.

Parameters

- **exposures** (*climada.entity.Exposures*)

- **impact_funcs** (*climada.entity.ImpactFuncSet*) – impact functions
- **hazard** (*climada.Hazard*)
- **save_mat** (*bool*) – self impact matrix: events x exposures

Examples

Use Entity class:

```
>>> haz = Hazard.from_mat(HAZ_DEMO_MAT)  # Set hazard
>>> haz.check()
>>> ent = Entity.from_excel(ENT_TEMPLATE_XLS) # Set exposures
>>> ent.check()
>>> imp = Impact()
>>> imp.calc(ent.exposures, ent.impact_funcs, haz)
>>> imp.calc_freq_curve().plot()
```

Specify only exposures and impact functions:

```
>>> haz = Hazard.from_mat(HAZ_DEMO_MAT)  # Set hazard
>>> haz.check()
>>> funcs = ImpactFuncSet.from_excel(ENT_TEMPLATE_XLS) # Set impact functions
>>> funcs.check()
>>> exp = Exposures(pd.read_excel(ENT_TEMPLATE_XLS)) # Set exposures
>>> exp.check()
>>> imp = Impact()
>>> imp.calc(exp, funcs, haz)
>>> imp.aai_agg
```

`calc_risk_transfer`(*attachment*, *cover*)

Compute traaditional risk transfer over impact. Returns new impact with risk transfer applied and the insurance layer resulting Impact metrics.

Parameters

- **attachment** (*float*) – (deductible)
- **cover** (*float*)

Return type climada.engine.Impact

`plot_hexbin_eai_exposure`(*mask=None*, *ignore_zero=True*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *axis=None*, *adapt_fontsize=True*, ***kwargs*)

Plot hexbin expected annual impact of each exposure.

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places
- **buffer** (*float, optional*) – border to add to coordinates. Default: 1.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use

- **kwargs** (*optional*) – arguments for hexbin matplotlib function

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_scatter_eai_exposure`(*mask=None*, *ignore_zero=True*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *axis=None*, *adapt_fontsize=True*, ***kwargs*)

Plot scatter expected annual impact of each exposure.

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places
- **buffer** (*float, optional*) – border to add to coordinates. Default: 1.0.
- **extend** (*str*) – optional extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- **kwargs** (*optional*) – arguments for hexbin matplotlib function

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_raster_eai_exposure`(*res=None*, *raster_res=None*, *save_tiff=None*, *raster_f=<function Impact.<lambda>>*, *label='value (log10)'*, *axis=None*, *adapt_fontsize=True*, ***kwargs*)

Plot raster expected annual impact of each exposure.

Parameters

- **res** (*float, optional*) – resolution of current data in units of latitude and longitude, approximated if not provided.
- **raster_res** (*float, optional*) – desired resolution of the raster
- **save_tiff** (*str, optional*) – file name to save the raster in tiff format, if provided
- **raster_f** (*lambda function*) – transformation to use to data. Default: log10 adding 1.
- **label** (*str colorbar label*)
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- **kwargs** (*optional*) – arguments for imshow matplotlib function

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_basemap_eai_exposure`(*mask=None*, *ignore_zero=False*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *zoom=10*, *url='http://tile.stamen.com/terrain/tileZ/tileX/tileY.png'*, *axis=None*, ***kwargs*)

Plot basemap expected annual impact of each exposure.

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted.

- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places
- **buffer** (*float, optional*) – border to add to coordinates. Default: 0.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **zoom** (*int, optional*) – zoom coefficient used in the satellite image
- **url** (*str, optional*) – image source, e.g. ctx.sources.OSM_C
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for scatter matplotlib function, e.g. cmap='Greys'. Default: 'Wistia'

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_hexbin_impact_exposure`(*event_id=1*, *mask=None*, *ignore_zero=True*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *axis=None*, *adapt_fontsize=True*, ***kwargs*)

Plot hexbin impact of an event at each exposure. Requires attribute imp_mat.

Parameters

- **event_id** (*int, optional*) – id of the event for which to plot the impact. Default: 1.
- **mask** (*np.array, optional*) – mask to apply to impact plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places
- **buffer** (*float, optional*) – border to add to coordinates. Default: 1.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **kwargs** (*optional*) – arguments for hexbin matplotlib function
- **axis** (*matplotlib.axes._subplots.AxesSubplot*) – optional axis to use
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Return type matplotlib.figure.Figure, cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_basemap_impact_exposure`(*event_id=1*, *mask=None*, *ignore_zero=True*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *zoom=10*, *url='http://tile.stamen.com/terrain/tileZ/tileX/tileY.png'*, *axis=None*, ***kwargs*)

Plot basemap impact of an event at each exposure. Requires attribute imp_mat.

Parameters

- **event_id** (*int, optional*) – id of the event for which to plot the impact. Default: 1.
- **mask** (*np.array, optional*) – mask to apply to impact plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places

- **buffer** (*float, optional*) – border to add to coordinates. Default: 0.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **zoom** (*int, optional*) – zoom coefficient used in the satellite image
- **url** (*str, optional*) – image source, e.g. ctx.sources.OSM_C
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional axis to use*)
- **kwargs** (*optional arguments for scatter matplotlib function, e.g.*) – cmap='Greys'. Default: 'Wistia'

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`write_csv`(*file_name*)
Write data into csv file. imp_mat is not saved.

Parameters **file_name** (*str*) – absolute path of the file

`write_excel`(*file_name*)
Write data into Excel file. imp_mat is not saved.

Parameters **file_name** (*str*) – absolute path of the file

`write_sparse_csr`(*file_name*)
Write imp_mat matrix in numpy's npz format.

`calc_impact_year_set`(*all_years=True*, *year_range=None*)
Calculate yearly impact from impact data.

Parameters

- **all_years** (*boolean*) – return values for all years between first and last year with event, including years without any events.
- **year_range** (*tuple or list with integers*) – start and end year

Return type Impact year set of type numpy.ndarray with summed impact per year.

`local_exceedance_imp`(*return_periods=(25, 50, 100, 250)*)
Compute exceedance impact map for given return periods. Requires attribute imp_mat.

Parameters **return_periods** (*np.array return periods to consider*)

Return type np.array

`plot_rp_imp`(*return_periods=(25, 50, 100, 250)*, *log10_scale=True*, *smooth=True*, *axis=None*, ***kwargs*)
Compute and plot exceedance impact maps for different return periods. Calls local_exceedance_imp.

Parameters

- **return_periods** (*tuple(int), optional*) – return periods to consider
- **log10_scale** (*boolean, optional*) – plot impact as log10(impact)
- **smooth** (*bool, optional*) – smooth plot to plot.RESOLUTIONxplot.RESOLUTION
- **kwargs** (*optional*) – arguments for pcolormesh matplotlib function used in event plots

Returns

- *matplotlib.axes._subplots.AxesSubplot,*
- *np.ndarray (return_periods.size x num_centroids)*

static read_sparse_csr(*file_name*)
Read imp_mat matrix from numpy's npz format.

Parameters **file_name** (*str file name*)

Return type sparse.csr_matrix

classmethod from_csv(*file_name*)
Read csv file containing impact data generated by write_csv.

Parameters **file_name** (*str absolute path of the file*)

Returns **imp** – Impact from csv file

Return type *climada.engine.impact.Impact*

read_csv(**args*, ***kwargs*)
This function is deprecated, use Impact.from_csv instead.

classmethod from_excel(*file_name*)
Read excel file containing impact data generated by write_excel.

Parameters **file_name** (*str absolute path of the file*)

Returns **imp** – Impact from excel file

Return type *climada.engine.impact.Impact*

read_excel(**args*, ***kwargs*)
This function is deprecated, use Impact.from_excel instead.

static video_direct_impact(*exp*, *impf_set*, *haz_list*, *file_name=''*, *writer=<matplotlib.animation.PillowWriter object>*, *imp_thresh=0*, *args_exp=None*, *args_imp=None*)
Computes and generates video of accumulated impact per input events over exposure.

Parameters

- **exp** (*Exposures*) – exposures instance, constant during all video
- **impf_set** (*ImpactFuncSet*) – impact functions
- **haz_list** ((*list(Hazard)*)) – every Hazard contains an event; all hazards use the same centroids
- **file_name** (*str, optional*) – file name to save video, if provided
- **writer** (*matplotlib.animation.*, optional*) – video writer. Default: pillow with bitrate=500
- **imp_thresh** (*float*) – represent damages greater than threshold
- **args_exp** (*optional*) – arguments for scatter (points) or hexbin (raster) matplotlib function used in exposures
- **args_imp** (*optional*) – arguments for scatter (points) or hexbin (raster) matplotlib function used in impact

Return type list(*Impact*)

select(*event_ids=None*, *event_names=None*, *dates=None*, *coord_exp=None*)
Select a subset of events and/or exposure points from the impact. If multiple input variables are not None, it returns all the impacts matching at least one of the conditions.

Note: the frequencies are NOT adjusted. Method to adjust frequencies

---

and obtain correct eai_exp: 1- Select subset of impact according to your choice imp = impact.select(...) 2- Adjust manually the frequency of the subset of impact imp.frequency = [...] 3- Use select without arguments to select all events and recompute the eai_exp with the updated frequencies. imp = imp.select()

Parameters

- **event_ids** (*list[int], optional*) – Selection of events by their id. The default is None.
- **event_names** (*list[str], optional*) – Selection of events by their name. The default is None.
- **dates** (*tuple(), optional*) – (start-date, end-date), events are selected if they are >= than start-date and <= than end-date. Dates in same format as impact.date (ordinal format of datetime library) The default is None.
- **coord_exp** (*np.ndarray), optional*) – Selection of exposures coordinates [lat, lon] (in degrees) The default is None.

Raises `ValueError` – If the impact matrix is missing, the eai_exp and aai_agg cannot be updated for a selection of events and/or exposures.

Returns **imp** – A new impact object with a selection of events and/or exposures

Return type climada.engine.Impact

climada.engine.impact_data module

climada.engine.impact_data.**assign_hazard_to_emdat**(*certainty_level*, *intensity_path_haz*, *names_path_haz*, *reg_id_path_haz*, *date_path_haz*, *emdat_data*, *start_time*, *end_time*, *keep_checks=False*)

assign_hazard_to_emdat: link EMdat event to hazard

Parameters

- **certainty_level** (*str*) – 'high' or 'low'
- **intensity_path_haz** (*sparse matrix*) – with hazards as rows and grid points as cols, values only at location with impacts
- **names_path_haz** (*str*) – identifier for each hazard (i.e. IBtracID) (rows of the matrix)
- **reg_id_path_haz** (*str*) – ISO country ID of each grid point (cols of the matrix)
- **date_path_haz** (*str*) – start date of each hazard (rows of the matrix)
- **emdat_data** (*pd.DataFrame*) – dataframe with EMdat data
- **start_time** (*str*) – start date of events to be assigned 'yyyy-mm-dd'
- **end_time** (*str*) – end date of events to be assigned 'yyyy-mm-dd'
- **keep_checks** (*bool, optional*)

Return type pd.dataframe with EMdat entries linked to a hazard

climada.engine.impact_data.**hit_country_per_hazard**(*intensity_path*, *names_path*, *reg_id_path*, *date_path*)

hit_country_per_hazard: create list of hit countries from hazard set

Parameters

- **intensity_path** (*str*) – Path to file containing sparse matrix with hazards as rows and grid points as cols, values only at location with impacts

- **names_path** (*str*) – Path to file with identifier for each hazard (i.e. IBtracID) (rows of the matrix)
- **reg_id_path** (*str*) – Path to file with ISO country ID of each grid point (cols of the matrix)
- **date_path** (*str*) – Path to file with start date of each hazard (rows of the matrix)

Return type pd.DataFrame with all hit countries per hazard

climada.engine.impact_data.**create_lookup**(*emdat_data*, *start*, *end*, *disaster_subtype='Tropical cyclone'*)
create_lookup: prepare a lookup table of EMdat events to which hazards can be assigned

Parameters

- **emdat_data** (*pd.DataFrame*) – with EMdat data
- **start** (*str*) – start date of events to be assigned 'yyyy-mm-dd'
- **end** (*str*) – end date of events to be assigned 'yyyy-mm-dd'
- **disaster_subtype** (*str*) – EMdat disaster subtype

Return type pd.DataFrame

climada.engine.impact_data.**emdat_possible_hit**(*lookup*, *hit_countries*, *delta_t*)
relate EM disaster to hazard using hit countries and time

Parameters

- **lookup** (*pd.DataFrame*) – to relate EMdatID to hazard
- **delta_t** – max time difference of start of EMdat event and hazard
- **hit_countries**

Return type list with possible hits

climada.engine.impact_data.**match_em_id**(*lookup*, *poss_hit*)
function to check if EM_ID has been assigned already and combine possible hits

Parameters

- **lookup** (*pd.dataframe*) – to relate EMdatID to hazard
- **poss_hit** (*list*) – with possible hits

Returns with all possible hits per EMdat ID

Return type list

climada.engine.impact_data.**assign_track_to_em**(*lookup*, *possible_tracks_1*, *possible_tracks_2*, *level*)
function to assign a hazard to an EMdat event to get some confidene into the procedure, hazards get only assigned if there is no other hazard occuring at a bigger time interval in that country Thus a track of possible_tracks_1 gets only assigned if there are no other tracks in possible_tracks_2. The confidence can be expressed with a certainty level

Parameters

- **lookup** (*pd.DataFrame*) – to relate EMdatID to hazard
- **possible_tracks_1** (*list*) – list of possible hits with smaller time horizon
- **possible_tracks_2** (*list*) – list of possible hits with larger time horizon
- **level** (*int*) – level of confidence

Returns lookup with assigend tracks and possible hits

Return type pd.DataFrame

climada.engine.impact_data.**check_assigned_track**(*lookup*, *checkset*)
compare lookup with assigned tracks to a set with checked sets

Parameters

- **lookup** (*pd.DataFrame*) – dataframe to relate EMdatID to hazard
- **checkset** (*pd.DataFrame*) – dataframe with already checked hazards

Return type error scores

climada.engine.impact_data.**clean_emdat_df**(*emdat_file*, *countries=None*, *hazard=None*, *year_range=None*, *target_version=2020*)
Get a clean and standardized DataFrame from EM-DAT-CSV-file (1) load EM-DAT data from CSV to DataFrame and remove header/footer, (2) handle version, clean up, and add columns, and (3) filter by country, hazard type and year range (if any given)

Parameters

- **emdat_file** (*str, Path, or DataFrame*) – Either string with full path to CSV-file or pandas.DataFrame loaded from EM-DAT CSV
- **countries** (*list of str*) – country ISO3-codes or names, e.g. ['JAM', 'CUB']. countries=None for all countries (default)
- **hazard** (*list or str*) – List of Disaster (sub-)type accordung EMDAT terminology, i.e.: Animal accident, Drought, Earthquake, Epidemic, Extreme temperature, Flood, Fog, Impact, Insect infestation, Landslide, Mass movement (dry), Storm, Volcanic activity, Wildfire; Coastal Flooding, Convective Storm, Riverine Flood, Tropical cyclone, Tsunami, etc.; OR CLIMADA hazard type abbreviations, e.g. TC, BF, etc.
- **year_range** (*list or tuple*) – Year range to be extracted, e.g. (2000, 2015); (only min and max are considered)
- **target_version** (*int*) – required EM-DAT data format version (i.e. year of download), changes naming of columns/variables (default: 2020)

Returns **df_data** – DataFrame containing cleaned and filtered EM-DAT impact data

Return type pd.DataFrame

climada.engine.impact_data.**emdat_countries_by_hazard**(*emdat_file_csv*, *hazard=None*, *year_range=None*)
return list of all countries exposed to a chosen hazard type from EMDAT data as CSV.

Parameters

- **emdat_file** (*str, Path, or DataFrame*) – Either string with full path to CSV-file or pandas.DataFrame loaded from EM-DAT CSV
- **hazard** (*list or str*) – List of Disaster (sub-)type accordung EMDAT terminology, i.e.: Animal accident, Drought, Earthquake, Epidemic, Extreme temperature, Flood, Fog, Impact, Insect infestation, Landslide, Mass movement (dry), Storm, Volcanic activity, Wildfire; Coastal Flooding, Convective Storm, Riverine Flood, Tropical cyclone, Tsunami, etc.; OR CLIMADA hazard type abbreviations, e.g. TC, BF, etc.
- **year_range** (*list or tuple*) – Year range to be extracted, e.g. (2000, 2015); (only min and max are considered)

Returns

- **countries_iso3a** (*list*) – List of ISO3-codes of countries impacted by the disaster (sub-)types
- **countries_names** (*list*) – List of names of countries impacted by the disaster (sub-)types

climada.engine.impact_data.**scale_impact2refyear**(*impact_values*, *year_values*, *iso3a_values*, *reference_year=None*)

Scale give impact values proportional to GDP to the according value in a reference year (for normalization of monetary values)

Parameters

- **impact_values** (*list or array*) – Impact values to be scaled.
- **year_values** (*list or array*) – Year of each impact (same length as impact_values)
- **iso3a_values** (*list or array*) – ISO3alpha code of country for each impact (same length as impact_values)
- **Optional Parameters**
- **————————-**
- **reference_year** (*int*) – Impact is scaled proportional to GDP to the value of the reference year. No scaling for reference_year=None (default)

climada.engine.impact_data.**emdat_impact_yearlysum**(*emdat_file_csv*, *countries=None*, *hazard=None*, *year_range=None*, *reference_year=None*, *imp_str="Total Damages ('000 US$)"*, *version=2020*)

function to load EM-DAT data and sum impact per year

Parameters **emdat_file_csv** (*str or DataFrame*) – Either string with full path to CSV-file or pandas.DataFrame loaded from EM-DAT CSV

countries [list of str] country ISO3-codes or names, e.g. ['JAM', 'CUB']. countries=None for all countries (default)

hazard [list or str] List of Disaster (sub-)type accordung EMDAT terminology, i.e.: Animal accident, Drought, Earthquake, Epidemic, Extreme temperature, Flood, Fog, Impact, Insect infestation, Landslide, Mass movement (dry), Storm, Volcanic activity, Wildfire; Coastal Flooding, Convective Storm, Riverine Flood, Tropical cyclone, Tsunami, etc.; OR CLIMADA hazard type abbreviations, e.g. TC, BF, etc.

year_range [list or tuple] Year range to be extracted, e.g. (2000, 2015); (only min and max are considered)

version [int] required EM-DAT data format version (i.e. year of download), changes naming of columns/variables (default: 2020)

Returns **out** – DataFrame with summed impact and scaled impact per year and country.

Return type pd.DataFrame

climada.engine.impact_data.**emdat_impact_event**(*emdat_file_csv*, *countries=None*, *hazard=None*, *year_range=None*, *reference_year=None*, *imp_str="Total Damages ('000 US$)"*, *version=2020*)

function to load EM-DAT data return impact per event

Parameters **emdat_file_csv** (*str or DataFrame*) – Either string with full path to CSV-file or pandas.DataFrame loaded from EM-DAT CSV

countries [list of str] country ISO3-codes or names, e.g. ['JAM', 'CUB']. default: countries=None for all countries

hazard [list or str] List of Disaster (sub-)type accordung EMDAT terminology, i.e.: Animal accident, Drought, Earthquake, Epidemic, Extreme temperature, Flood, Fog, Impact, Insect infestation, Landslide, Mass move-

ment (dry), Storm, Volcanic activity, Wildfire; Coastal Flooding, Convective Storm, Riverine Flood, Tropical cyclone, Tsunami, etc.; OR CLIMADA hazard type abbreviations, e.g. TC, BF, etc.

year_range [list or tuple] Year range to be extracted, e.g. (2000, 2015); (only min and max are considered)

reference_year [int reference year of exposures. Impact is scaled] proportional to GDP to the value of the reference year. Default: No scaling for 0

imp_str [str] Column name of impact metric in EMDAT CSV, default = "Total Damages ('000 US$)"

version [int] EM-DAT version to take variable/column names from (defaul: 2020)

Returns **out** – EMDAT DataFrame with new columns "year", "region_id", and "impact" and +impact_scaled" total impact per event with same unit as chosen impact, but multiplied by 1000 if impact is given as 1000 US$ (e.g. imp_str="Total Damages ('000 US$) scaled").

Return type pd.DataFrame

climada.engine.impact_data.**emdat_to_impact**(*emdat_file_csv*, *hazard_type_climada*, *year_range=None*, *countries=None*, *hazard_type_emdat=None*, *reference_year=None*, *imp_str='Total Damages'*)

function to load EM-DAT data return impact per event

Parameters **emdat_file_csv** (*str or pd.DataFrame*) – Either string with full path to CSV-file or pandas.DataFrame loaded from EM-DAT CSV

countries [list of str] country ISO3-codes or names, e.g. ['JAM', 'CUB']. default: countries=None for all countries

hazard_type_climada [list or str] List of Disaster (sub-)type accordung EMDAT terminology, i.e.: Animal accident, Drought, Earthquake, Epidemic, Extreme temperature, Flood, Fog, Impact, Insect infestation, Landslide, Mass movement (dry), Storm, Volcanic activity, Wildfire; Coastal Flooding, Convective Storm, Riverine Flood, Tropical cyclone, Tsunami, etc.; OR CLIMADA hazard type abbreviations, e.g. TC, BF, etc.

year_range [list or tuple] Year range to be extracted, e.g. (2000, 2015); (only min and max are considered)

reference_year [int reference year of exposures. Impact is scaled] proportional to GDP to the value of the reference year. Default: No scaling for 0

imp_str [str] Column name of impact metric in EMDAT CSV, default = "Total Damages ('000 US$)"

Returns

- **impact_instance** (*instance of climada.engine.Impact*)
- *impact object of same format as output from CLIMADA*
- *impact computation.*
- *Values scaled with GDP to reference_year if reference_year is given.*
- *i.e. current US$ for imp_str="Total Damages ('000 US$) scaled" (factor 1000 is applied)*
- *impact_instance.eai_exp holds expected annual impact for each country.*
- *impact_instance.coord_exp holds rough central coordinates for each country.*
- **countries (list)** (*ISO3-codes of countries in same order as in impact_instance.eai_exp*)

7.1.2 climada.entity package

climada.entity.disc_rates package

climada.entity.disc_rates.base module

class climada.entity.disc_rates.base.**DiscRates**(*years=None*, *rates=None*, *tag=None*)
Bases: object

Defines discount rates and basic methods. Loads from files with format defined in FILE_EXT.

tag
information about the source data

Type *Tag*

years
list of years

Type np.array

rates
list of discount rates for each year (between 0 and 1)

Type np.array

__init__(*years=None*, *rates=None*, *tag=None*)
Fill discount rates with values and check consistency data

Parameters

- **years** (*numpy.ndarray(int)*) – Array of years. Default is numpy.array([]).
- **rates** (*numpy.ndarray(float)*) – Discount rates for each year in years. Default is numpy.array([]). Note: rates given in float, e.g., to set 1% rate use 0.01
- **tag** (*climate.entity.tag*) – Metadata. Default is None.

clear()
Reinitialize attributes.

check()
Check attributes consistency.

Raises **ValueError** –

select(*year_range*)
Select discount rates in given years.

Parameters

- **year_range** (*np.array(int)*) – continuous sequence of selected years.
- **Returns** (*climada.entity.DiscRates*) – The selected discrates in the year_range

append(*disc_rates*)
Check and append discount rates to current DiscRates. Overwrite discount rate if same year.

Parameters **disc_rates** (*climada.entity.DiscRates*) – DiscRates instance to append

Raises **ValueError** –

net_present_value(*ini_year*, *end_year*, *val_years*)
Compute net present value between present year and future year.

Parameters

- **ini_year** (*float*) – initial year
- **end_year** (*float*) – end year
- **val_years** (*np.array*) – cash flow at each year btw ini_year and end_year (both included)

Returns **net_present_value** – net present value between present year and future year.

Return type float

`plot`(*axis=None*, *figsize=(6, 8)*, ***kwargs*)

Plot discount rates per year.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*tuple(int, int), optional*) – size of the figure. The default is (6,8)
- **kwargs** (*optional*) – keyword arguments passed to plotting function axis.plot

Returns **axis** – axis handles of the plot

Return type matplotlib.axes._subplots.AxesSubplot

`classmethod from_mat`(*file_name*, *description=''*, *var_names={'field_name': 'discount', 'sup_field_name': 'entity', 'var_name': {'disc': 'discount_rate', 'year': 'year'}}*)

Read MATLAB file generated with previous MATLAB CLIMADA version.

Parameters

- **file_name** (*str*) – filename including path and extension
- **description** (*str, optional*) – description of the data. The default is ‘’
- **var_names** (*dict, optional*) – name of the variables in the file. The Default is DEF_VAR_MAT = {‘sup_field_name’: ‘entity’, ‘field_name’: ‘discount’,

  ‘var_name’: {‘year’: ‘year’, ‘disc’: ‘discount_rate’}}

Returns The disc rates from matlab

Return type climada.entity.DiscRates

`read_mat`(**args*, ***kwargs*)

This function is deprecated, use DiscRates.from_mats instead.

`classmethod from_excel`(*file_name*, *description=''*, *var_names={'col_name': {'disc': 'discount_rate', 'year': 'year'}, 'sheet_name': 'discount'}*)

Read excel file following template and store variables.

Parameters

- **file_name** (*str*) – filename including path and extension
- **description** (*str, optional*) – description of the data. The default is ‘’
- **var_names** (*dict, optional*) – name of the variables in the file. The Default is DEF_VAR_EXCEL = {‘sheet_name’: ‘discount’,

  ‘col_name’: {‘year’: ‘year’, ‘disc’: ‘discount_rate’}}

Returns The disc rates from excel

Return type climada.entity.DiscRates

read_excel(**args*, ***kwargs*)
This function is deprecated, use DiscRates.from_excel instead.

write_excel(*file_name*, *var_names={'col_name': {'disc': 'discount_rate', 'year': 'year'}, 'sheet_name': 'discount'}*)
Write excel file following template.

Parameters

- **file_name** (*str*) – filename including path and extension
- **var_names** (*dict, optional*) – name of the variables in the file. The Default is DEF_VAR_EXCEL = {'sheet_name': 'discount',
  'col_name': {'year': 'year', 'disc': 'discount_rate'}}

climada.entity.exposures package

climada.entity.exposures.litpop package

climada.entity.exposures.litpop.gpw_population module

climada.entity.exposures.litpop.gpw_population.**load_gpw_pop_shape**(*geometry*, *reference_year*, *gpw_version*, *data_dir=PosixPath('/home/docs/climada/data')*, *layer=0*, *verbatim=True*)
Read gridded population data from TIFF and crop to given shape(s).

Note: A (free) NASA Earthdata login is necessary to download the data. Data can be downloaded e.g. for gpw_version=11 from https://sedac.ciesin.columbia.edu/downloads/data/gpw-v4/ gpw-v4-population-count-rev11/gpw-v4-population-count-rev11_2015_30_sec_tif.zip

Parameters

- **geometry** (*shape(s) to crop data to in degree lon/lat.*) – for example shapely.geometry.(Multi)Polygon or shapefile.Shape from polygon(s) defined in a (country) shapefile.
- **reference_year** (*int*) – target year for data extraction
- **gpw_version** (*int*) – Version number of GPW population data, i.e. 11 for v4.11. The default is CONFIG.exposures.litpop.gpw_population.gpw_version.int()
- **data_dir** (*Path, optional*) – Path to data directory holding GPW data folders. The default is SYSTEM_DIR.
- **layer** (*int, optional*) – relevant data layer in input TIFF file to return. The default is 0 and should not be changed without understanding the different data layers in the given TIFF file.
- **verbatim** (*bool (optional):*) – if False, output in LOGGER is surpressed. Default is True.

Returns

- **pop_data** (*2D numpy array*) – contains extracted population count data per grid point in shape first dimension is lat, second dimension is lon.
- **meta** (*dict*) – contains meta data per array, including "transform" with meta data on coordinates.
- **global_transform** (*Affine instance*) – contains six numbers, providing transform info for global GWP grid. global_transform is required for resampling on a globally consistent grid

climada.entity.exposures.litpop.gpw_population.**get_gpw_file_path**(*gpw_version*, *reference_year*, *data_dir=PosixPath('/home/docs/climada/data')*, *verbatim=True*)

Check available GPW population data versions and year closest to *reference_year* and return full path to TIFF file.

Parameters

- **gpw_version** (*int (optional)*) – Version number of GPW population data, i.e. 11 for v4.11.
- **reference_year** (*int (optional)*) – Data year is selected as close to reference_year as possible. The default is 2020.
- **data_dir** (*pathlib.Path (optional)*) – Absolute path where files are stored. Default: SYSTEM_DIR

Raises `FileExistsError` –

Returns pathlib.Path

Return type path to input file with population data

climada.entity.exposures.litpop.litpop module

climada.entity.exposures.litpop.litpop.**GPW_VERSION = 11**

Version of Gridded Population of the World (GPW) input data. Check for updates.

class climada.entity.exposures.litpop.litpop.**LitPop**(**args*, *meta=None*, *tag=None*, *ref_year=2018*, *value_unit='USD'*, *crs=None*, ***kwargs*)

Bases: *climada.entity.exposures.base.Exposures*

Holds geopandas GeoDataFrame with metada and columns (pd.Series) defined in Attributes of Exposures class. LitPop exposure values are disaggregated proportional to a combination of nightlight intensity (NASA) and Gridded Population data (SEDAC). Total asset values can be produced capital, population count, GDP, or non-financial wealth.

Calling sequence example: country_names = ['CHE', 'Austria'] exp = LitPop.from_countries(country_names) exp.plot()

exponents

Defining powers (m, n) with which lit (nightlights) and pop (gpw) go into Lit**m * Pop**n. The default is (1,1).

Type tuple of two integers, optional

fin_mode

Socio-economic value to be used as an asset base that is disaggregated. The default is 'pc'.

Type str, optional

gpw_version

Version number of GPW population data, e.g. 11 for v4.11. The default is defined in GPW_VERSION.

Type int, optional

set_countries(**args*, ***kwargs*)

This function is deprecated, use LitPop.from_countries instead.

`classmethod from_countries`(*countries*, *res_arcsec=30*, *exponents=(1, 1)*, *fin_mode='pc'*, *total_values=None*, *admin1_calc=False*, *reference_year=2018*, *gpw_version=11*, *data_dir=PosixPath('/home/docs/climada/data')*)

Init new LitPop exposure object for a list of countries (admin 0). Sets attributes *ref_year*, *tag*, *crs*, *value*, *geometry*, *meta*, *value_unit*, *exponents*,`fin_mode`, *gpw_version*, and *admin1_calc*.

Parameters

- **countries** (*list with str or int*) – list containing country identifiers: iso3alpha (e.g. 'JPN'), iso3num (e.g. 92) or name (e.g. 'Togo')
- **res_arcsec** (*float, optional*) – Horizontal resolution in arc-sec. The default is 30 arcsec, this corresponds to roughly 1 km.
- **exponents** (*tuple of two integers, optional*) – Defining power with which lit (nightlights) and pop (gpw) go into LitPop. To get nightlights^3 without population count: (3, 0). To use population count alone: (0, 1). Default: (1, 1)
- **fin_mode** (*str, optional*) – Socio-economic value to be used as an asset base that is disaggregated to the grid points within the country
  - 'pc': produced capital (Source: World Bank), incl. manufactured or built assets such as machinery, equipment, and physical structures *pc* is in constant 2014 USD.
  - 'pop': population count (source: GPW, same as gridded population). The unit is 'people'.
  - 'gdp': gross-domestic product (Source: World Bank) [USD]
  - 'income_group': gdp multiplied by country's income group+1 [USD]. Income groups are 1 (low) to 4 (high income).
  - 'nfw': non-financial wealth (Source: Credit Suisse, of households only) [USD]
  - 'tw': total wealth (Source: Credit Suisse, of households only) [USD]
  - 'norm': normalized by country (no unit)
  - 'none': LitPop per pixel is returned unchanged (no unit)

  The default is 'pc'.
- **total_values** (*list containing numerics, same length as countries, optional*) – Total values to be disaggregated to grid in each country. The default is None. If None, the total number is extracted from other sources depending on the value of fin_mode.
- **admin1_calc** (*boolean, optional*) – If True, distribute admin1-level GDP (if available). Default: False
- **reference_year** (*int, optional*) – Reference year. Default: CONFIG.exposures.def_ref_year.
- **gpw_version** (*int, optional*) – Version number of GPW population data. The default is GPW_VERSION
- **data_dir** (*Path, optional*) – redefines path to input data directory. The default is SYSTEM_DIR.

Raises `ValueError` –

Returns **exp** – LitPop instance with exposure for given countries

Return type *LitPop*

set_nightlight_intensity(**args*, ***kwargs*)
This function is deprecated, use LitPop.from_nightlight_intensity instead.

classmethod from_nightlight_intensity(*countries=None*, *shape=None*, *res_arcsec=15*, *reference_year=2018*, *data_dir=PosixPath('/home/docs/climada/data')*)
Wrapper around *from_countries* / *from_shape*.

Initiate exposures instance with value equal to the original BlackMarble nightlight intensity resampled to the target resolution *res_arcsec*.

Provide either *countries* or *shape*.

Parameters

- **countries** (*list or str, optional*) – list containing country identifiers (name or iso3)
- **shape** (*Shape, Polygon or MultiPolygon, optional*) – geographical shape of target region, alternative to *countries*.
- **res_arcsec** (*int, optional*) – Resolution in arc seconds. The default is 15.
- **reference_year** (*int, optional*) – Reference year. The default is CONFIG.exposures.def_ref_year.
- **data_dir** (*Path, optional*) – data directory. The default is None.

Raises ValueError –

Returns **exp** – Exposure instance with values representing pure nightlight intensity from input nightlight data (BlackMarble)

Return type *LitPop*

set_population(**args*, ***kwargs*)
This function is deprecated, use LitPop.from_population instead.

classmethod from_population(*countries=None*, *shape=None*, *res_arcsec=30*, *reference_year=2018*, *gpw_version=11*, *data_dir=PosixPath('/home/docs/climada/data')*)
Wrapper around *from_countries* / *from_shape*.

Initiate exposures instance with value equal to GPW population count. Provide either *countries* or *shape*.

Parameters

- **countries** (*list or str, optional*) – list containing country identifiers (name or iso3)
- **shape** (*Shape, Polygon or MultiPolygon, optional*) – geographical shape of target region, alternative to *countries*.
- **res_arcsec** (*int, optional*) – Resolution in arc seconds. The default is 30.
- **reference_year** (*int, optional*) – Reference year (closest available GPW data year is used) The default is CONFIG.exposures.def_ref_year.
- **gpw_version** (*int, optional*) – specify GPW data verison. The default is 11.
- **data_dir** (*Path, optional*) – data directory. The default is None. Either countries or shape is required.

Raises ValueError –

Returns **exp** – Exposure instance with values representing population count according to Gridded Population of the World (GPW) input data set.

Return type *LitPop*

`set_custom_shape_from_countries`(**args*, ***kwargs*)
This function is deprecated, use LitPop.from_shape_and_countries instead.

`classmethod from_shape_and_countries`(*shape*, *countries*, *res_arcsec=30*, *exponents=(1, 1)*, *fin_mode='pc'*, *admin1_calc=False*, *reference_year=2018*, *gpw_version=11*, *data_dir=PosixPath('/home/docs/climada/data')*)
create LitPop exposure for *country* and then crop to given shape.

Parameters

- **shape** (*shapely.geometry.Polygon, MultiPolygon, shapereader.Shape,*) – or GeoSeries or list containg either Polygons or Multipolygons. Geographical shape for which LitPop Exposure is to be initiated.
- **countries** (*list with str or int*) – list containing country identifiers: iso3alpha (e.g. 'JPN'), iso3num (e.g. 92) or name (e.g. 'Togo')
- **res_arcsec** (*float, optional*) – Horizontal resolution in arc-sec. The default is 30 arcsec, this corresponds to roughly 1 km.
- **exponents** (*tuple of two integers, optional*) – Defining power with which lit (nightlights) and pop (gpw) go into LitPop. Default: (1, 1)
- **fin_mode** (*str, optional*) – Socio-economic value to be used as an asset base that is disaggregated to the grid points within the country
  - 'pc': produced capital (Source: World Bank), incl. manufactured or built assets such as machinery, equipment, and physical structures (pc is in constant 2014 USD)
  - 'pop': population count (source: GPW, same as gridded population). The unit is 'people'.
  - 'gdp': gross-domestic product (Source: World Bank) [USD]
  - 'income_group': gdp multiplied by country's income group+1 [USD] Income groups are 1 (low) to 4 (high income).
  - 'nfw': non-financial wealth (Source: Credit Suisse, of households only) [USD]
  - 'tw': total wealth (Source: Credit Suisse, of households only) [USD]
  - 'norm': normalized by country
  - 'none': LitPop per pixel is returned unchanged

  The default is 'pc'.
- **admin1_calc** (*boolean, optional*) – If True, distribute admin1-level GDP (if available). Default: False
- **reference_year** (*int, optional*) – Reference year for data sources. Default: 2020
- **gpw_version** (*int, optional*) – Version number of GPW population data. The default is GPW_VERSION
- **data_dir** (*Path, optional*) – redefines path to input data directory. The default is SYSTEM_DIR.

Raises `NotImplementedError` –

Returns **exp** – The exposure LitPop within shape

Return type *LitPop*

set_custom_shape(**args*, ***kwargs*)
This function is deprecated, use LitPop.from_shape instead.

classmethod from_shape(*shape*, *total_value*, *res_arcsec=30*, *exponents=(1, 1)*, *value_unit='USD'*, *reference_year=2018*, *gpw_version=11*, *data_dir=PosixPath('/home/docs/climada/data')*)
init LitPop exposure object for a custom shape. Requires user input regarding the total value to be disaggregated.

Sets attributes *ref_year*, *tag*, *crs*, *value*, *geometry*, *meta*, *value_unit*, *exponents*,`fin_mode`, *gpw_version*, and *admin1_calc*.

This method can be used to initiated LitPop Exposure for sub-national regions such as states, districts, cantons, cities, ... but shapes and total value need to be provided manually. If these required input parameters are not known / available, better initiate Exposure for entire country and extract shape afterwards.

Parameters

- **shape** (*shapely.geometry.Polygon or MultiPolygon or shapereader.Shape.*) – Geographical shape for which LitPop Exposure is to be initiated.
- **total_value** (*int, float or None type*) – Total value to be disaggregated to grid in shape. If None, no value is disaggregated.
- **res_arcsec** (*float, optional*) – Horizontal resolution in arc-sec. The default 30 arcsec corresponds to roughly 1 km.
- **exponents** (*tuple of two integers, optional*) – Defining power with which lit (nightlights) and pop (gpw) go into LitPop.
- **value_unit** (*str*) – Unit of exposure values. The default is USD.
- **reference_year** (*int, optional*) – Reference year for data sources. Default: CONFIG.exposures.def_ref_year
- **gpw_version** (*int, optional*) – Version number of GPW population data. The default is set in CONFIG.
- **data_dir** (*Path, optional*) – redefines path to input data directory. The default is SYSTEM_DIR.

Raises

- **NotImplementedError** –
- **ValueError** –
- **TypeError** –

Returns **exp** – The exposure LitPop within shape

Return type *LitPop*

set_country(**args*, ***kwargs*)
This function is deprecated, use LitPop.from_countries instead.

climada.entity.exposures.litpop.litpop.**get_value_unit**(*fin_mode*)
get *value_unit* depending on *fin_mode*

Parameters **fin_mode** (*Socio-economic value to be used as an asset base*)

Returns **value_unit**

Return type str

climada.entity.exposures.litpop.litpop.**reproject_input_data**(*data_array_list*, *meta_list*, *i_align=0*, *target_res_arcsec=None*, *global_origins=(- 180.0, 89.99999999999991)*, *resampling=Resampling.bilinear*, *conserve=None*)

LitPop-sepcific wrapper around u_coord.align_raster_data.

Reprojects all arrays in data_arrays to a given resolution – all based on the population data grid.

Parameters

- **data_array_list** (*list or array of numpy arrays containing numbers*) – Data to be reprojected, i.e. list containing N (min. 1) 2D-arrays. The data with the reference grid used to align the global destination grid to should be first data_array_list[i_align], e.g., pop (GPW population data) for LitPop.
- **meta_list** (*list of dicts*) – meta data dictionaries of data arrays in same order as data_array_list. Required fields in each dict are 'dtype,', 'width', 'height', 'crs', 'transform'. Example:

  {'driver': 'GTiff', 'dtype': 'float32', 'nodata': 0, 'width': 2702, 'height': 1939, 'count': 1, 'crs': CRS.from_epsg(4326), 'transform': Affine(0.00833333333333333, 0.0, -18.175000000000068,

  0.0, -0.00833333333333333, 43.79999999999993)}

  The meta data with the reference grid used to define the global destination grid should be first in the list, e.g., GPW population data for LitPop.
- **i_align** (*int, optional*) – Index/Position of meta in meta_list to which the global grid of the destination is to be aligned to (c.f. u_coord.align_raster_data) The default is 0.
- **target_res_arcsec** (*int, optional*) – target resolution in arcsec. The default is None, i.e. same resolution as reference data.
- **global_origins** (*tuple with two numbers (lat, lon), optional*) – global lon and lat origins as basis for destination grid. The default is the same as for GPW population data:

  (-180.0, 89.99999999999991)
- **resampling** (*resampling function, optional*) – The default is rasterio.warp.Resampling.bilinear
- **conserve** (*str, optional, either 'mean' or 'sum'*) – Conserve mean or sum of data? The default is None (no conservation).

Returns

- **data_array_list** (*list*) – contains reprojected data sets
- **meta_out** (*dict*) – contains meta data of new grid (same for all arrays)

climada.entity.exposures.litpop.litpop.**gridpoints_core_calc**(*data_arrays*, *offsets=None*, *exponents=None*, *total_val_rescale=None*)

Combines N dense numerical arrays by point-wise multipilcation and optionally rescales to new total value: (1) An offset (1 number per array) is added to all elements in

the corresponding data array in data_arrays (optional).

(2) Numbers in each array are taken to the power of the corresponding exponent (optional).

(3) Arrays are multiplied element-wise.

(4) if total_val_rescale is provided, results are normalized and re-scaled with total_val_rescale.

(5) One array with results is returned.

Parameters

- **data_arrays** (*list or array of numpy arrays containing numbers*) – Data to be combined, i.e. list containing N (min. 1) arrays of same shape.
- **total_val_rescale** (*float or int, optional*) – Total value for optional rescaling of resulting array. All values in result_array are skaled so that the sum is equal to total_val_rescale. The default (None) implies no rescaling.
- **offsets** (*list or array containing N numbers >= 0, optional*) – One numerical offset per array that is added (sum) to the corresponding array in data_arrays. The default (None) corresponds to np.zeros(N).
- **exponents** (*list or array containing N numbers >= 0, optional*) – One exponent per array used as power for the corresponding array. The default (None) corresponds to np.ones(N).

Raises `ValueError` – If input lists don't have the same number of elements. Or: If arrays in data_arrays do not have the same shape.

Returns **result_array** – Results from calculation described above.

Return type np.array of same shape as arrays in data_arrays

climada.entity.exposures.litpop.nightlight module

climada.entity.exposures.litpop.nightlight.**NOAA_RESOLUTION_DEG = 0.008333333333333333**
NOAA nightlights coordinates resolution in degrees.

climada.entity.exposures.litpop.nightlight.**NASA_RESOLUTION_DEG = 0.004166666666666667**
NASA nightlights coordinates resolution in degrees.

climada.entity.exposures.litpop.nightlight.**NASA_TILE_SIZE = (21600, 21600)**
NASA nightlights tile resolution.

climada.entity.exposures.litpop.nightlight.**NOAA_BORDER = (-180, -65, 180, 75)**
NOAA nightlights border (min_lon, min_lat, max_lon, max_lat)

climada.entity.exposures.litpop.nightlight.**BM_FILENAMES =**
['BlackMarble_%i_A1_geo_gray.tif', 'BlackMarble_%i_A2_geo_gray.tif',
'BlackMarble_%i_B1_geo_gray.tif', 'BlackMarble_%i_B2_geo_gray.tif',
'BlackMarble_%i_C1_geo_gray.tif', 'BlackMarble_%i_C2_geo_gray.tif',
'BlackMarble_%i_D1_geo_gray.tif', 'BlackMarble_%i_D2_geo_gray.tif']
Nightlight NASA files which generate the whole earth when put together.

climada.entity.exposures.litpop.nightlight.**load_nasa_nl_shape**(*geometry*, *year*, *data_dir=PosixPath('/home/docs/climada/data')*, *dtype='float32'*)

Read nightlight data from NASA BlackMarble tiles cropped to given shape(s) and combine arrays from each tile. 1) check and download required blackmarble files 2) read and crop data from each file required in a bounding box around

the given *geometry*.

3) **combine data from all input files into one array. this array then** contains all data in the geographic bounding box around *geometry*.

4) return array with nightlight data

Parameters

- **geometry** (*shape(s) to crop data to in degree lon/lat.*) – for example shapely.geometry.(Multi)Polygon or shapefile.Shape. from polygon defined in a shapefile. The object should have attribute 'bounds' or 'points'
- **year** (*int*) – target year for nightlight data, e.g. 2016. Closest availble year is selected.
- **data_dir** (*Path (optional)*) – Path to directory with BlackMarble data. The default is SYSTEM_DIR.
- **dtype** (*dtype*) – data type for output default 'float32', required for LitPop, choose 'int8' for integer.

Returns

- **results_array** (*numpy array*) – extracted and combined nightlight data for bounding box around shape
- **meta** (*dict*) – rasterio meta data for results_array

climada.entity.exposures.litpop.nightlight.**get_required_nl_files**(*bounds*)

Determines which of the satellite pictures are necessary for a certain bounding box (e.g. country)

Parameters **bounds** (*1x4 tuple*) – bounding box from shape (min_lon, min_lat, max_lon, max_lat).

Raises **ValueError** – invalid *bounds*

Returns **req_files** – Array indicating the required files for the current operation with a boolean value (1: file is required, 0: file is not required).

Return type numpy array

climada.entity.exposures.litpop.nightlight.**check_nl_local_file_exists**(*required_files=None*, *check_path=PosixPath('/home/docs/climada*
year=2016)

Checks if BM Satellite files are avaialbe and returns a vector denoting the missing files.

Parameters

- **required_files** (*numpy array, optional*) – boolean array of dimension (8,) with which some files can be skipped. Only files with value 1 are checked, with value zero are skipped. The default is np.ones(len(BM_FILENAMES),)
- **check_path** (*str or Path*) – absolute path where files are stored. Default: SYSTEM_DIR
- **year** (*int*) – year of the image, e.g. 2016

Returns **files_exist** – Boolean array that denotes if the required files exist.

Return type numpy array

climada.entity.exposures.litpop.nightlight.**download_nl_files**(*req_files=array([1., 1., 1., 1., 1., 1., 1., 1.])*, *files_exist=array([0., 0., 0., 0., 0., 0., 0., 0.])*, *dwnl_path=PosixPath('/home/docs/climada/data')*, *year=2016*)

Attempts to download nightlight files from NASA webpage.

Parameters

- **req_files** (*numpy array, optional*) –

  Boolean array which indicates the files required (0-> skip, 1-> download). The default is np.ones(len(BM_FILENAMES),).

- **files_exist** (*numpy array, optional*) –

  Boolean array which indicates if the files already exist locally and should not be downloaded (0-> download, 1-> skip). The default is np.zeros(len(BM_FILENAMES),).

- **dwnl_path** (*str or path, optional*) – Download directory path. The default is SYSTEM_DIR.

- **year** (*int, optional*) – Data year to be downloaded. The default is 2016.

Raises

- `ValueError` –
- `RuntimeError` –

Returns **dwnl_path** – Download directory path.

Return type str or path

climada.entity.exposures.litpop.nightlight.**load_nasa_nl_shape_single_tile**(*geometry*, *path*, *layer=0*)

Read nightlight data from single NASA BlackMarble tile and crop to given shape.

Parameters

- **geometry** (*shape or geometry object*) – shape(s) to crop data to in degree lon/lat. for example shapely.geometry.Polygon object or from polygon defined in a shapefile.
- **path** (*Path or str*) – full path to BlackMarble tif (including filename)
- **layer** (*int, optional*) – TIFF-layer to be returned. The default is 0. BlackMarble usually comes with 3 layers.

Returns

- **out_image[layer,** (*,:] : 2D numpy ndarray*) – 2d array with data cropped to bounding box of shape
- **meta** (*dict*) – rasterio meta

climada.entity.exposures.litpop.nightlight.**load_nightlight_nasa**(*bounds*, *req_files*, *year*)

Get nightlight from NASA repository that contain input boundary.

Note: Legacy for BlackMarble, not required for litpop module

Parameters

- **bounds** (*tuple*) – min_lon, min_lat, max_lon, max_lat
- **req_files** (*np.array*) – array with flags for NASA files needed

- **year** (*int*) – nightlight year

Returns

- **nightlight** (*sparse.csr_matrix*)
- **coord_nl** (*np.array*)

climada.entity.exposures.litpop.nightlight.**read_bm_file**(*bm_path*, *filename*)

Reads a single NASA BlackMarble GeoTiff and returns the data. Run all required checks first.

Note: Legacy for BlackMarble, not required for litpop module

Parameters

- **bm_path** (*str*) – absolute path where files are stored.
- **filename** (*str*) – filename of the file to be read.

Returns

- **arr1** (*array*) – Raw BM data
- **curr_file** (*gdal GeoTiff File*) – Additional info from which coordinates can be calculated.

climada.entity.exposures.litpop.nightlight.**unzip_tif_to_py**(*file_gz*)

Unzip image file, read it, flip the x axis, save values as pickle and remove tif.

Parameters **file_gz** (*str*) – file fith .gz format to unzip

Returns

- **fname** (*str*) – file_name of unzipped file
- **nightlight** (*sparse.csr_matrix*)

climada.entity.exposures.litpop.nightlight.**untar_noaa_stable_nightlight**(*f_tar_ini*)

Move input tar file to SYSTEM_DIR and extract stable light file. Returns absolute path of stable light file in format tif.gz.

Parameters **f_tar_ini** (*str*) – absolute path of file

Returns **f_tif_gz** – path of stable light file

Return type str

climada.entity.exposures.litpop.nightlight.**load_nightlight_noaa**(*ref_year=2013*, *sat_name=None*)

Get nightlight luminosites. Nightlight matrix, lat and lon ordered such that nightlight[1][0] corresponds to lat[1], lon[0] point (the image has been flipped).

Parameters

- **ref_year** (*int, optional*) – reference year. The default is 2013.
- **sat_name** (*str, optional*) – satellite provider (e.g. 'F10', 'F18', ...)

Returns

- **nightlight** (*sparse.csr_matrix*)
- **coord_nl** (*np.array*)
- **fn_light** (*str*)

climada.entity.exposures.base module

class climada.entity.exposures.base.**Exposures**(**args*, *meta=None*, *tag=None*, *ref_year=2018*, *value_unit='USD'*, *crs=None*, ***kwargs*)

Bases: object

geopandas GeoDataFrame with metada and columns (pd.Series) defined in Attributes.

tag
metada - information about the source data

Type *Tag*

ref_year
metada - reference year

Type int

value_unit
metada - unit of the exposures values

Type str

latitude
latitude

Type pd.Series

longitude
longitude

Type pd.Series

crs
CRS information inherent to GeoDataFrame.

Type dict or crs

value
a value for each exposure

Type pd.Series

impf_
e.g. impf_TC. impact functions id for hazard TC. There might be different hazards defined: impf_TC, impf_FL, … If not provided, set to default '**impf_**' with ids 1 in check().

Type pd.Series, optional

geometry
geometry of type Point of each instance. Computed in method set_geometry_points().

Type pd.Series, optional

meta
dictionary containing corresponding raster properties (if any): width, height, crs and transform must be present at least (transform needs to contain upper left corner!). Exposures might not contain all the points of the corresponding raster. Not used in internal computations.

Type dict

deductible
deductible value for each exposure

Type pd.Series, optional

cover
cover value for each exposure

Type pd.Series, optional

category_id
category id for each exposure

Type pd.Series, optional

region_id
region id for each exposure

Type pd.Series, optional

centr_
e.g. centr_TC. centroids index for hazard TC. There might be different hazards defined: centr_TC, centr_FL, ... Computed in method assign_centroids().

Type pd.Series, optional

vars_oblig = ['value', 'latitude', 'longitude']
Name of the variables needed to compute the impact.

vars_def = ['impf_', 'if_']
Name of variables that can be computed.

vars_opt = ['centr_', 'deductible', 'cover', 'category_id', 'region_id', 'geometry']
Name of the variables that aren't need to compute the impact.

property crs
Coordinate Reference System, refers to the crs attribute of the inherent GeoDataFrame

__init__(**args*, *meta=None*, *tag=None*, *ref_year=2018*, *value_unit='USD'*, *crs=None*, ***kwargs*)
Creates an Exposures object from a GeoDataFrame

Parameters

- **args** – Arguments of the GeoDataFrame constructor
- **kwargs** – Named arguments of the GeoDataFrame constructor, additionally
- **meta** (*dict, optional*) – Metadata dictionary. Default: {} (empty dictionary)
- **tag** (*climada.entity.exposures.tag.Tag, optional*) – Exposures tag. Defaults to the entry of the same name in *meta* or an empty Tag object.
- **ref_year** (*int, optional*) – Reference Year. Defaults to the entry of the same name in *meta* or 2018.
- **value_unit** (*str, optional*) – Unit of the exposed value. Defaults to the entry of the same name in *meta* or 'USD'.
- **crs** (*object, anything accepted by pyproj.CRS.from_user_input*) – Coordinate reference system. Defaults to the entry of the same name in *meta*, or to the CRS of the GeoDataFrame (if provided) or to 'epsg:4326'.

check()
Check Exposures consistency.

Reports missing columns in log messages. If no impf_* column is present in the dataframe, a default column '**impf_**' is added with default impact function id 1.

`set_crs`(*crs=None*)
Set the Coordinate Reference System. If the epxosures GeoDataFrame has a 'geometry' column it will be updated too.

Parameters **crs** (*object, optional*) – anything anything accepted by pyproj.CRS.from_user_input if the original value is None it will be set to the default CRS.

`set_gdf`(*gdf: geopandas.geodataframe.GeoDataFrame*, *crs=None*)
Set the *gdf* GeoDataFrame and update the CRS

Parameters

- **gdf** (*GeoDataFrame*)
- **crs** (*object, optional,*) – anything anything accepted by pyproj.CRS.from_user_input, by default None, then *gdf.crs* applies or - if not set - the exposure's current crs

`get_impf_column`(*haz_type=''*)
Find the best matching column name in the exposures dataframe for a given hazard type,

Parameters **haz_type** (*str or None*) – hazard type, as in the hazard's tag.haz_type which is the HAZ_TYPE constant of the hazard's module

Returns a column name, the first of the following that is present in the exposures' dataframe: - impf_[haz_type] - if_[haz_type] - **impf_** - **if_**

Return type str

Raises `ValueError` – if none of the above is found in the dataframe.

`assign_centroids`(*hazard*, *distance='euclidean'*, *threshold=100*)
Assign for each exposure coordinate closest hazard coordinate. -1 used for disatances > threshold in point distances. If raster hazard, -1 used for centroids outside raster.

Parameters

- **hazard** (*Hazard*) – Hazard to match (with raster or vector centroids).
- **distance** (*str, optional*) – Distance to use in case of vector centroids. Possible values are "euclidean", "haversine" and "approx". Default: "euclidean"
- **threshold** (*float*) – If the distance (in km) to the nearest neighbor exceeds *threshold*, the index *-1* is assigned. Set *threshold* to 0, to disable nearest neighbor matching. Default: 100 (km)

See also:

climada.util.coordinates.assign_coordinates method to associate centroids to exposure points

Notes

The default order of use is: 1. if centroid raster is defined, assign exposures points to the closest raster point. 2. if no raster, assign centroids to the nearest neighbor using euclidian metric

Both cases can introduce innacuracies for coordinates in lat/lon coordinates as distances in degrees differ from distances in meters on the Earth surface, in particular for higher latitude and distances larger than 100km. If more accuracy is needed, please use 'haversine' distance metric. This however is slower for (quasi-)gridded data, and works only for non-gridded data.

`set_geometry_points`(*scheduler=None*)
Set geometry attribute of GeoDataFrame with Points from latitude and longitude attributes.

Parameters **scheduler** (*str, optional*) – used for dask map_partitions. "threads", "synchronous" or "processes"

`set_lat_lon()`
Set latitude and longitude attributes from geometry attribute.

`set_from_raster(*args, **kwargs)`
This function is deprecated, use Exposures.from_raster instead.

`classmethod from_raster(file_name, band=1, src_crs=None, window=False, geometry=False, dst_crs=False, transform=None, width=None, height=None, resampling=Resampling.nearest)`
Read raster data and set latitude, longitude, value and meta

Parameters

- **file_name** (*str*) – file name containing values
- **band** (*int, optional*) – bands to read (starting at 1)
- **src_crs** (*crs, optional*) – source CRS. Provide it if error without it.
- **window** (*rasterio.windows.Windows, optional*) – window where data is extracted
- **geometry** (*shapely.geometry, optional*) – consider pixels only in shape
- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float*) – number of lons for transform
- **height** (*float*) – number of lats for transform
- **resampling** (*rasterio.warp,.Resampling optional*) – resampling function used for reprojection to dst_crs

Return type *Exposures*

`plot_scatter(mask=None, ignore_zero=False, pop_name=True, buffer=0.0, extend='neither', axis=None, figsize=(9, 13), adapt_fontsize=True, **kwargs)`
Plot exposures geometry's value sum scattered over Earth's map. The plot will we projected according to the current crs.

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places, by default True.
- **buffer** (*float, optional*) – border to add to coordinates. Default: 0.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*tuple, optional*) – figure size for plt.subplots
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

- **kwargs** (*optional*) – arguments for scatter matplotlib function, e.g. cmap='Greys'. Default: 'Wistia'

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_hexbin`(*mask=None*, *ignore_zero=False*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *axis=None*, *figsize=(9, 13)*, *adapt_fontsize=True*, ***kwargs*)

Plot exposures geometry's value sum binned over Earth's map. An other function for the bins can be set through the key reduce_C_function. The plot will we projected according to the current crs.

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places, by default True.
- **buffer** (*float, optional*) – border to add to coordinates. Default: 0.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*tuple*) – figure size for plt.subplots
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- **kwargs** (*optional*) – arguments for hexbin matplotlib function, e.g. reduce_C_function=np.average. Default: reduce_C_function=np.sum

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

`plot_raster`(*res=None*, *raster_res=None*, *save_tiff=None*, *raster_f=<function Exposures.<lambda>>*, *label='value (log10)'*, *scheduler=None*, *axis=None*, *figsize=(9, 13)*, *fill=True*, *adapt_fontsize=True*, ***kwargs*)

Generate raster from points geometry and plot it using log10 scale: np.log10((np.fmax(raster+1, 1))).

Parameters

- **res** (*float, optional*) – resolution of current data in units of latitude and longitude, approximated if not provided.
- **raster_res** (*float, optional*) – desired resolution of the raster
- **save_tiff** (*str, optional*) – file name to save the raster in tiff format, if provided
- **raster_f** (*lambda function*) – transformation to use to data. Default: log10 adding 1.
- **label** (*str*) – colorbar label
- **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*tuple, optional*) – figure size for plt.subplots
- **fill** (*bool, optional*) – If false, the areas with no data will be plotted in white. If True, the areas with missing values are filled as 0s. The default is True.

- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- **kwargs** (*optional*) – arguments for imshow matplotlib function

Return type matplotlib.figure.Figure, cartopy.mpl.geoaxes.GeoAxesSubplot

plot_basemap(*mask=None*, *ignore_zero=False*, *pop_name=True*, *buffer=0.0*, *extend='neither'*, *zoom=10*, *url='http://tile.stamen.com/terrain/tileZ/tileX/tileY.png'*, *axis=None*, ***kwargs*)

Scatter points over satellite image using contextily

Parameters

- **mask** (*np.array, optional*) – mask to apply to eai_exp plotted. Same size of the exposures, only the selected indexes will be plot.
- **ignore_zero** (*bool, optional*) – flag to indicate if zero and negative values are ignored in plot. Default: False
- **pop_name** (*bool, optional*) – add names of the populated places, by default True.
- **buffer** (*float, optional*) – border to add to coordinates. Default: 0.0.
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max']
- **zoom** (*int, optional*) – zoom coefficient used in the satellite image
- **url** (*str, optional*) – image source, e.g. ctx.sources.OSM_C
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for scatter matplotlib function, e.g. cmap='Greys'. Default: 'Wistia'

Return type matplotlib.figure.Figure, cartopy.mpl.geoaxes.GeoAxesSubplot

write_hdf5(*file_name*)

Write data frame and metadata in hdf5 format

Parameters **file_name** (*str*) – (path and) file name to write to.

read_hdf5(**args*, ***kwargs*)

This function is deprecated, use Exposures.from_hdf5 instead.

classmethod from_hdf5(*file_name*)

Read data frame and metadata in hdf5 format

Parameters

- **file_name** (*str*) – (path and) file name to read from.
- **additional_vars** (*list*) – list of additional variable names to read that are not in exposures.base._metadata

Return type *Exposures*

read_mat(**args*, ***kwargs*)

This function is deprecated, use Exposures.from_mat instead.

classmethod from_mat(*file_name*, *var_names=None*)

Read MATLAB file and store variables in exposures.

Parameters

- **file_name** (*str*) – absolute path file

- **var_names** (*dict, optional*) – dictionary containing the name of the MATLAB variables. Default: DEF_VAR_MAT.

Return type *Exposures*

`to_crs`(*crs=None*, *epsg=None*, *inplace=False*)
Wrapper of the GeoDataFrame.to_crs method.

Transform geometries to a new coordinate reference system. Transform all geometries in a GeoSeries to a different coordinate reference system. The crs attribute on the current GeoSeries must be set. Either crs in string or dictionary form or an EPSG code may be specified for output. This method will transform all points in all objects. It has no notion or projecting entire geometries. All segments joining points are assumed to be lines in the current projection, not geodesics. Objects crossing the dateline (or other projection boundary) will have undesirable behavior.

Parameters

- **crs** (*dict or str*) – Output projection parameters as string or in dictionary form.
- **epsg** (*int*) – EPSG code specifying output projection.
- **inplace** (*bool, optional, default: False*) – Whether to return a new GeoDataFrame or do the transformation in place.

Return type None if inplace is True else a transformed copy of the exposures object

`plot`(**args*, ***kwargs*)
Plot a GeoDataFrame.

Generate a plot of a GeoDataFrame with matplotlib. If a column is specified, the plot coloring will be based on values in that column.

Parameters

- **column** (*str, np.array, pd.Series (default None)*) – The name of the dataframe column, np.array, or pd.Series to be plotted. If np.array or pd.Series are used then it must have same length as dataframe. Values are used to color the plot. Ignored if *color* is also set.
- **kind** (*str*) –

  The kind of plots to produce:

  – ‘geo’: Map (default)

  Pandas Kinds - ‘line’ : line plot - ‘bar’ : vertical bar plot - ‘barh’ : horizontal bar plot - ‘hist’ : histogram - ‘box’ : BoxPlot - ‘kde’ : Kernel Density Estimation plot - ‘density’ : same as ‘kde’ - ‘area’ : area plot - ‘pie’ : pie plot - ‘scatter’ : scatter plot - ‘hexbin’ : hexbin plot.
- **cmap** (*str (default None)*) – The name of a colormap recognized by matplotlib.
- **color** (*str (default None)*) – If specified, all objects will be colored uniformly.
- **ax** (*matplotlib.pyplot.Artist (default None)*) – axes on which to draw the plot
- **cax** (*matplotlib.pyplot Artist (default None)*) – axes on which to draw the legend in case of color map.
- **categorical** (*bool (default False)*) – If False, cmap will reflect numerical values of the column being plotted. For non-numerical columns, this will be set to True.
- **legend** (*bool (default False)*) – Plot a legend. Ignored if no *column* is given, or if *color* is given.

- **scheme** (*str (default None)*) – Name of a choropleth classification scheme (requires mapclassify). A mapclassify.MapClassifier object will be used under the hood. Supported are all schemes provided by mapclassify (e.g. 'BoxPlot', 'EqualInterval', 'FisherJenks', 'FisherJenksSampled', 'HeadTailBreaks', 'JenksCaspall', 'JenksCaspallForced', 'JenksCaspallSampled', 'MaxP', 'MaximumBreaks', 'NaturalBreaks', 'Quantiles', 'Percentiles', 'StdMean', 'UserDefined'). Arguments can be passed in classification_kwds.
- **k** (*int (default 5)*) – Number of classes (ignored if scheme is None)
- **vmin** (*None or float (default None)*) – Minimum value of cmap. If None, the minimum data value in the column to be plotted is used.
- **vmax** (*None or float (default None)*) – Maximum value of cmap. If None, the maximum data value in the column to be plotted is used.
- **markersize** (*str or float or sequence (default None)*) – Only applies to point geometries within a frame. If a str, will use the values in the column of the frame specified by markersize to set the size of markers. Otherwise can be a value to apply to all points, or a sequence of the same length as the number of points.
- **figsize** (*tuple of integers (default None)*) – Size of the resulting matplotlib.figure.Figure. If the argument axes is given explicitly, figsize is ignored.
- **legend_kwds** (*dict (default None)*) – Keyword arguments to pass to matplotlib.pyplot.legend() or matplotlib.pyplot.colorbar(). Additional accepted keywords when *scheme* is specified:

  fmt [string] A formatting specification for the bin edges of the classes in the legend. For example, to have no decimals: `{"fmt": "{:.0f}"}`.

  labels [list-like] A list of legend labels to override the auto-generated labels. Needs to have the same number of elements as the number of classes (*k*).

  interval [boolean (default False)] An option to control brackets from mapclassify legend. If True, open/closed interval brackets are shown in the legend.
- **categories** (*list-like*) – Ordered list-like object of categories to be used for categorical plot.
- **classification_kwds** (*dict (default None)*) – Keyword arguments to pass to mapclassify
- **missing_kwds** (*dict (default None)*) – Keyword arguments specifying color options (as style_kwds) to be passed on to geometries with missing values in addition to or overwriting other style kwds. If None, geometries with missing values are not plotted.
- **aspect** (*'auto', 'equal', None or float (default 'auto')*) – Set aspect of axis. If 'auto', the default aspect for map plots is 'equal'; if however data are not projected (coordinates are long/lat), the aspect is by default set to 1/cos(df_y * pi/180) with df_y the y coordinate of the middle of the GeoDataFrame (the mean of the y range of bounding box) so that a long/lat square appears square in the middle of the plot. This implies an Equirectangular projection. If None, the aspect of *ax* won't be changed. It can also be set manually (float) as the ratio of y-unit to x-unit.
- ****style_kwds** (*dict*) – Style options to be passed on to the actual plot function, such as `edgecolor`, `facecolor`, `linewidth`, `markersize`, `alpha`.

Returns **ax**

Return type matplotlib axes instance

Examples

```
>>> df = geopandas.read_file(geopandas.datasets.get_path("naturalearth_lowres
↪"))
>>> df.head()
    pop_est      continent                      name iso_a3  gdp_md_est       ␣
↪                                  geometry
0     920938        Oceania                      Fiji    FJI      8374.0 ␣
↪MULTIPOLYGON (((180.00000 -16.06713, 180.00000...
1   53950935         Africa                  Tanzania    TZA    150600.0 ␣
↪POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...
2     603253         Africa                 W. Sahara    ESH       906.5 ␣
↪POLYGON ((-8.66559 27.65643, -8.66512 27.58948...
3   35623680  North America                    Canada    CAN   1674000.0 ␣
↪MULTIPOLYGON (((-122.84000 49.00000, -122.9742...
4  326625791  North America  United States of America    USA  18560000.0 ␣
↪MULTIPOLYGON (((-122.84000 49.00000, -120.0000...
```

```
>>> df.plot("pop_est", cmap="Blues")
```

See the User Guide page ../../user_guide/mapping for details.

copy(*deep=True*)
Make a copy of this Exposures object.

Parameters **deep (bool)** (*Make a deep copy, i.e. also copy data. Default True.*)

Return type *Exposures*

write_raster(*file_name*, *value_name='value'*, *scheduler=None*)
Write value data into raster file with GeoTiff format

Parameters **file_name** (*str*) – name output file in tif format

static concat(*exposures_list*)
Concatenates Exposures or DataFrame objectss to one Exposures object.

Parameters **exposures_list** (*list of Exposures or DataFrames*) – The list must not be empty with the first item supposed to be an Exposures object.

Returns with the metadata of the first item in the list and the dataframes concatenated.

Return type *Exposures*

climada.entity.exposures.base.**add_sea**(*exposures*, *sea_res*, *scheduler=None*)
Add sea to geometry's surroundings with given resolution. region_id set to -1 and other variables to 0.

Parameters

- **exposures** (*Exposures*) – the Exposures object without sea surroundings.
- **sea_res** (*tuple (float,float)*) – (sea_coast_km, sea_res_km), where first parameter is distance from coast to fill with water and second parameter is resolution between sea points
- **scheduler** (*str, optional*) – used for dask map_partitions. "threads", "synchronous" or "processes"

Return type *Exposures*

climada.entity.exposures.base.**INDICATOR_IMPF = 'impf_'**
Name of the column containing the impact functions id of specified hazard

climada.entity.exposures.base.**INDICATOR_CENTR = 'centr_'**
Name of the column containing the centroids id of specified hazard

climada.entity.impact_funcs package

climada.entity.impact_funcs.base module

class climada.entity.impact_funcs.base.**ImpactFunc**
Bases: object

Contains the definition of one impact function.

haz_type
hazard type acronym (e.g. 'TC')

Type str

id
id of the impact function. Exposures of the same type will refer to the same impact function id

Type int or str

name
name of the ImpactFunc

Type str

intensity_unit
unit of the intensity

Type str

intensity
intensity values

Type np.array

mdd
mean damage (impact) degree for each intensity (numbers in [0,1])

Type np.array

paa
percentage of affected assets (exposures) for each intensity (numbers in [0,1])

Type np.array

__init__()
Empty initialization.

calc_mdr(*inten*)
Interpolate impact function to a given intensity.

Parameters **inten** (*float or np.array*) – intensity, the x-coordinate of the interpolated values.

Return type np.array

plot(*axis=None*, ***kwargs*)
Plot the impact functions MDD, MDR and PAA in one graph, where MDR = PAA * MDD.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use

- **kwargs** (*optional*) – arguments for plot matplotlib function, e.g. marker='x'

Return type matplotlib.axes._subplots.AxesSubplot

`check()`
Check consistent instance data.

Raises `ValueError` –

`classmethod from_step_impf(intensity, mdd=(0, 1), paa=(1, 1), impf_id=1)`
Step function type impact function.

By default, everything is destroyed above the step. Useful for high resolution modelling.

This method modifies self (climada.entity.impact_funcs instance) by assigning an id, intensity, mdd and paa to the impact function.

Parameters

- **intensity** (*tuple(float, float, float)*) – tuple of 3-intensity numbers: (minimum, threshold, maximum)
- **mdd** (*tuple(float, float)*) – (min, max) mdd values. The default is (0, 1)
- **paa** (*tuple(float, float)*) – (min, max) paa values. The default is (1, 1)
- **impf_id** (*int, optional, default=1*) – impact function id

Returns **impf** – Step impact function

Return type climada.entity.impact_funcs.ImpactFunc

`set_step_impf(*args, **kwargs)`
This function is deprecated, use ImpactFunc.from_step_impf instead.

`classmethod from_sigmoid_impf(intensity, L, k, x0, if_id=1)`
Sigmoid type impact function hinging on three parameter.

This type of impact function is very flexible for any sort of study, hazard and resolution. The sigmoid is defined as:

$$f(x) =$$

rac{L}{1+exp^{-k(x-x0)}}

For more information: https://en.wikipedia.org/wiki/Logistic_function

This method modifies self (climada.entity.impact_funcs instance) by assining an id, intensity, mdd and paa to the impact function.

intensity: tuple(float, float, float) tuple of 3 intensity numbers along np.arange(min, max, step)

L [float] "top" of sigmoid

k [float] "slope" of sigmoid

x0 [float] intensity value where f(x)==L/2

if_id [int, optional, default=1] impact function id

impf [climada.entity.impact_funcs.ImpactFunc] Step impact function

`set_sigmoid_impf(*args, **kwargs)`
This function is deprecated, use LitPop.from_countries instead.

climada.entity.impact_funcs.impact_func_set module

class climada.entity.impact_funcs.impact_func_set.**ImpactFuncSet**

Bases: object

Contains impact functions of type ImpactFunc. Loads from files with format defined in FILE_EXT.

tag
information about the source data

Type *Tag*

_data
contains ImpactFunc classes. It's not suppossed to be directly accessed. Use the class methods instead.

Type dict

__init__()
Empty initialization.

Examples

Fill impact functions with values and check consistency data:

```
>>> fun_1 = ImpactFunc()
>>> fun_1.haz_type = 'TC'
>>> fun_1.id = 3
>>> fun_1.intensity = np.array([0, 20])
>>> fun_1.paa = np.array([0, 1])
>>> fun_1.mdd = np.array([0, 0.5])
>>> imp_fun = ImpactFuncSet()
>>> imp_fun.append(fun_1)
>>> imp_fun.check()
```

Read impact functions from file and checks consistency data.

```
>>> imp_fun = ImpactFuncSet()
>>> imp_fun.read(ENT_TEMPLATE_XLS)
```

clear()
Reinitialize attributes.

append(*func*)
Append a ImpactFunc. Overwrite existing if same id and haz_type.

Parameters **func** (*ImpactFunc*) – ImpactFunc instance

Raises **ValueError** –

remove_func(*haz_type=None*, *fun_id=None*)
Remove impact function(s) with provided hazard type and/or id. If no input provided, all impact functions are removed.

Parameters

- **haz_type** (*str, optional*) – all impact functions with this hazard
- **fun_id** (*int, optional*) – all impact functions with this id

`get_func`(*haz_type=None*, *fun_id=None*)
Get ImpactFunc(s) of input hazard type and/or id. If no input provided, all impact functions are returned.

Parameters

- **haz_type** (*str, optional*) – hazard type
- **fun_id** (*int, optional*) – ImpactFunc id

Returns

- *ImpactFunc (if haz_type and fun_id),*
- *list(ImpactFunc) (if haz_type or fun_id),*
- **{ImpactFunc.haz_type** (*{ImpactFunc.id : ImpactFunc}} (if None)*)

`get_hazard_types`(*fun_id=None*)
Get impact functions hazard types contained for the id provided. Return all hazard types if no input id.

Parameters **fun_id** (*int, optional*) – id of an impact function

Return type list(str)

`get_ids`(*haz_type=None*)
Get impact functions ids contained for the hazard type provided. Return all ids for each hazard type if no input hazard type.

Parameters **haz_type** (*str, optional*) – hazard type from which to obtain the ids

Returns

- *list(ImpactFunc.id) (if haz_type provided),*
- **{ImpactFunc.haz_type** (*list(ImpactFunc.id)} (if no haz_type)*)

`size`(*haz_type=None*, *fun_id=None*)
Get number of impact functions contained with input hazard type and /or id. If no input provided, get total number of impact functions.

Parameters

- **haz_type** (*str, optional*) – hazard type
- **fun_id** (*int, optional*) – ImpactFunc id

Return type int

`check`()
Check instance attributes.

Raises `ValueError` –

`extend`(*impact_funcs*)
Append impact functions of input ImpactFuncSet to current ImpactFuncSet. Overwrite ImpactFunc if same id and haz_type.

Parameters **impact_funcs** (*ImpactFuncSet*) – ImpactFuncSet instance to extend

Raises `ValueError` –

`plot`(*haz_type=None*, *fun_id=None*, *axis=None*, ***kwargs*)
Plot impact functions of selected hazard (all if not provided) and selected function id (all if not provided).

Parameters

- **haz_type** (*str, optional*) – hazard type

- **fun_id** (*int, optional*) – id of the function

Return type matplotlib.axes._subplots.AxesSubplot

classmethod from_excel(*file_name*, *description=''*, *var_names={'col_name': {'func_id': 'impact_fun_id', 'inten': 'intensity', 'mdd': 'mdd', 'name': 'name', 'paa': 'paa', 'peril': 'peril_id', 'unit': 'intensity_unit'}, 'sheet_name': 'impact_functions'}*)

Read excel file following template and store variables.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names** (*dict, optional*) – name of the variables in the file

Return type ImpFuncSet

read_excel(**args*, ***kwargs*)

This function is deprecated, use ImpactFuncSet.from_excel instead.

classmethod from_mat(*file_name*, *description=''*, *var_names={'field_name': 'damagefunctions', 'sup_field_name': 'entity', 'var_name': {'fun_id': 'DamageFunID', 'inten': 'Intensity', 'mdd': 'MDD', 'name': 'name', 'paa': 'PAA', 'peril': 'peril_ID', 'unit': 'Intensity_unit'}}*)

Read MATLAB file generated with previous MATLAB CLIMADA version.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names** (*dict, optional*) – name of the variables in the file

Returns **impf_set** – Impact func set as defined in matlab file.

Return type climada.entity.impact_func_set.ImpactFuncSet

read_mat(**args*, ***kwargs*)

This function is deprecated, use ImpactFuncSet.from_mat instead.

write_excel(*file_name*, *var_names={'col_name': {'func_id': 'impact_fun_id', 'inten': 'intensity', 'mdd': 'mdd', 'name': 'name', 'paa': 'paa', 'peril': 'peril_id', 'unit': 'intensity_unit'}, 'sheet_name': 'impact_functions'}*)

Write excel file following template.

Parameters

- **file_name** (*str*) – absolute file name to write
- **var_names** (*dict, optional*) – name of the variables in the file

climada.entity.impact_funcs.storm_europe module

class climada.entity.impact_funcs.storm_europe.**ImpfStormEurope**
Bases: *climada.entity.impact_funcs.base.ImpactFunc*

Impact functions for tropical cyclones.

__init__()
Empty initialization.

classmethod from_schwierz(*impf_id=1*)
Generate the impact function of Schwierz et al. 2010, doi:10.1007/s10584-009-9712-1

Returns **impf** – impact function for asset damages due to storm defined in Schwierz et al. 2010

Return type climada.entity.impact_funcs.storm_europe.ImpfStormEurope:

classmethod from_welker(*impf_id=1*)
Return the impact function of Welker et al. 2021, doi:10.5194/nhess-21-279-2021 It is the Schwierz function, calibrated with a simple multiplicative factor to minimize RMSE between modelled damages and reported damages.

Returns **impf** – impact function for asset damages due to storm defined in Welker et al. 2021

Return type climada.entity.impact_funcs.storm_europe.ImpfStormEurope:

set_schwierz(*impf_id=1*)
This function is deprecated, use ImpfStormEurope.from_schwierz instead.

set_welker(*impf_id=1*)
This function is deprecated, use ImpfStormEurope.from_welker instead.

climada.entity.impact_funcs.trop_cyclone module

class climada.entity.impact_funcs.trop_cyclone.**ImpfTropCyclone**
Bases: *climada.entity.impact_funcs.base.ImpactFunc*

Impact functions for tropical cyclones.

__init__()
Empty initialization.

set_emanuel_usa(**args*, ***kwargs*)
This function is deprecated, use from_emanuel_usa() instead.

classmethod from_emanuel_usa(*impf_id=1*, *intensity=array([0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120])*, *v_thresh=25.7*, *v_half=74.7*, *scale=1.0*)
Init TC impact function using the formula of Kerry Emanuel, 2011: 'Global Warming Effects on U.S. Hurricane Damage', https://doi.org/10.1175/WCAS-D-11-00007.1

Parameters

- **impf_id** (*int, optional*) – impact function id. Default: 1
- **intensity** (*np.array, optional*) – intensity array in m/s. Default: 5 m/s step array from 0 to 120m/s
- **v_thresh** (*float, optional*) – first shape parameter, wind speed in m/s below which there is no damage. Default: 25.7(Emanuel 2011)

- **v_half** (*float, optional*) – second shape parameter, wind speed in m/s at which 50% of max. damage is expected. Default: v_threshold + 49 m/s (mean value of Sealy & Strobl 2017)
- **scale** (*float, optional*) – scale parameter, linear scaling of MDD. 0<=scale<=1. Default: 1.0

Raises `ValueError` –

Returns **impf** – TC impact function instance based on formula by Emanuel (2011)

Return type *ImpfTropCyclone*

class climada.entity.impact_funcs.trop_cyclone.**ImpfSetTropCyclone**

Bases: *climada.entity.impact_funcs.impact_func_set.ImpactFuncSet*

Impact function set (ImpfS) for tropical cyclones.

__init__()

Empty initialization.

Examples

Fill impact functions with values and check consistency data:

```
>>> fun_1 = ImpactFunc()
>>> fun_1.haz_type = 'TC'
>>> fun_1.id = 3
>>> fun_1.intensity = np.array([0, 20])
>>> fun_1.paa = np.array([0, 1])
>>> fun_1.mdd = np.array([0, 0.5])
>>> imp_fun = ImpactFuncSet()
>>> imp_fun.append(fun_1)
>>> imp_fun.check()
```

Read impact functions from file and checks consistency data.

```
>>> imp_fun = ImpactFuncSet()
>>> imp_fun.read(ENT_TEMPLATE_XLS)
```

set_calibrated_regional_ImpfSet(**args*, ***kwargs*)

This function is deprecated, use from_calibrated_regional_ImpfSet() instead.

classmethod from_calibrated_regional_ImpfSet(*calibration_approach='TDR'*, *q=0.5*, *input_file_path=None*, *version=1*)

initiate TC wind impact functions based on Eberenz et al. 2021: https://doi.org/10.5194/nhess-21-393-2021

Parameters

- **calibration_approach** (*str*) –

  'TDR' (default): Total damage ratio (TDR) optimization with TDR=1.0 (simulated damage = reported damage from EM-DAT)

  'TDR1.5' [Total damage ratio (TDR) optimization with] TDR=1.5 (simulated damage = 1.5*reported damage from EM-DAT)

  'RMSF': Root-mean-squared fraction (RMSF) optimization 'EDR': quantile from individually fitted v_half per event,

i.e. v_half fitted to get EDR=1.0 for each event

- **q** (*float*) – quantile between 0 and 1.0 to select (EDR only, default=0.5, i.e. median v_half)
- **input_file_path** (*str or DataFrame*) – full path to calibration result file to be used instead of default file in repository (expert users only)

Returns **v_half** – Impf slope parameter v_half per region

Return type dict

Returns **impf_set** – TC Impact Function Set based on Eberenz et al, 2021.

Return type *ImpfSetTropCyclone*

`static calibrated_regional_vhalf`(*calibration_approach='TDR'*, *q=0.5*, *input_file_path=None*, *version=1*)

return calibrated TC wind impact function slope parameter v_half per region based on Eberenz et al., 2021: https://doi.org/10.5194/nhess-21-393-2021

Parameters

- **calibration_approach** (*str*) –

  'TDR' (default): Total damage ratio (TDR) optimization with TDR=1.0 (simulated damage = reported damage from EM-DAT)

  'TDR1.5' [Total damage ratio (TDR) optimization with] TDR=1.5 (simulated damage = 1.5*reported damage from EM-DAT)

  'RMSF': Root-mean-squared fraction (RMSF) optimization 'EDR': quantile from individually fitted v_half per event,

  i.e. v_half fitted to get EDR=1.0 for each event
- **q** (*float*) – quantile between 0 and 1.0 to select (EDR only, default=0.5, i.e. median v_half)
- **input_file_path** (*str or DataFrame*) – full path to calibration result file to be used instead of default file in repository (expert users only)

Raises `ValueError` –

Returns **v_half** – TC impact function slope parameter v_half per region

Return type dict

`static get_countries_per_region`(*region=None*)

Returns dictionaries with numerical and alphabetical ISO3 codes of all countries associated to a calibration region. Only contains countries that were affected by tropical cyclones between 1980 and 2017 according to EM-DAT.

Parameters **region** (*str*) – regional abbreviation (default='all'), either 'NA1', 'NA2', 'NI', 'OC', 'SI', 'WP1', 'WP2', 'WP3', 'WP4', or 'all'.

Returns

- **region_name** (*dict or str*) – long name per region
- **impf_id** (*dict or int*) – impact function ID per region
- **iso3n** (*dict or list*) – numerical ISO3codes (=region_id) per region
- **iso3a** (*dict or list*) – numerical ISO3codes (=region_id) per region

climada.entity.measures package

climada.entity.measures.base module

class climada.entity.measures.base.**Measure**
Bases: object

Contains the definition of one measure.

name
name of the measure

Type str

haz_type
related hazard type (peril), e.g. TC

Type str

color_rgb
integer array of size 3. Color code of this measure in RGB

Type np.array

cost
discounted cost (in same units as assets)

Type float

hazard_set
file name of hazard to use (in h5 format)

Type str

hazard_freq_cutoff
hazard frequency cutoff

Type float

exposures_set
file name of exposure to use (in h5 format) or Exposure instance

Type str or climada.entity.Exposure

imp_fun_map
change of impact function id of exposures, e.g. '1to3'

Type str

hazard_inten_imp
parameter a and b of hazard intensity change

Type tuple(float, float)

mdd_impact
parameter a and b of the impact over the mean damage degree

Type tuple(float, float)

paa_impact
parameter a and b of the impact over the percentage of affected assets

Type tuple(float, float)

`exp_region_id`
region id of the selected exposures to consider ALL the previous parameters

Type int

`risk_transf_attach`
risk transfer attachment

Type float

`risk_transf_cover`
risk transfer cover

Type float

`risk_transf_cost_factor`
factor to multiply to resulting insurance layer to get the total cost of risk transfer

Type float

`__init__()`
Empty initialization.

`check()`
Check consistent instance data.

Raises `ValueError` –

`calc_impact(exposures, imp_fun_set, hazard)`
Apply measure and compute impact and risk transfer of measure implemented over inputs.

Parameters

- **exposures** (*climada.entity.Exposures*) – exposures instance
- **imp_fun_set** (*climada.entity.ImpactFuncSet*) – impact function set instance
- **hazard** (*climada.hazard.Hazard*) – hazard instance

Returns

- *climada.engine.Impact*
- *resulting impact and risk transfer of measure*

`apply(exposures, imp_fun_set, hazard)`
Implement measure with all its defined parameters.

Parameters

- **exposures** (*climada.entity.Exposures*) – exposures instance
- **imp_fun_set** (*climada.entity.ImpactFuncSet*) – impact function set instance
- **hazard** (*climada.hazard.Hazard*) – hazard instance

Returns

new_exp, new_ifs, new_haz – climada.entity.ImpactFuncSet, climada.hazard.Hazard

Exposure, impact function set with implemented measure with all defined parameters.

Return type climada.entity.Exposure,

climada.entity.measures.measure_set module

class climada.entity.measures.measure_set.**MeasureSet**

Bases: object

Contains measures of type Measure. Loads from files with format defined in FILE_EXT.

tag
information about the source data

Type *Tag*

_data
cotains Measure classes. It's not suppossed to be directly accessed. Use the class methods instead.

Type dict

__init__()
Empty initialization.

Examples

Fill MeasureSet with values and check consistency data:

```
>>> act_1 = Measure()
>>> act_1.name = 'Seawall'
>>> act_1.color_rgb = np.array([0.1529, 0.2510, 0.5451])
>>> act_1.hazard_intensity = (1, 0)
>>> act_1.mdd_impact = (1, 0)
>>> act_1.paa_impact = (1, 0)
>>> meas = MeasureSet()
>>> meas.append(act_1)
>>> meas.tag.description = "my dummy MeasureSet."
>>> meas.check()
```

Read measures from file and checks consistency data:

```
>>> meas = MeasureSet.from_excel(ENT_TEMPLATE_XLS)
```

clear()
Reinitialize attributes.

append(*meas*)
Append an Measure. Override if same name and haz_type.

Parameters **meas** (*Measure*) – Measure instance

Raises **ValueError** –

remove_measure(*haz_type=None*, *name=None*)
Remove impact function(s) with provided hazard type and/or id. If no input provided, all impact functions are removed.

Parameters

- **haz_type** (*str, optional*) – all impact functions with this hazard
- **name** (*str, optional*) – measure name

`get_measure`(*haz_type=None*, *name=None*)
Get ImpactFunc(s) of input hazard type and/or id. If no input provided, all impact functions are returned.

Parameters

- **haz_type** (*str, optional*) – hazard type
- **name** (*str, optional*) – measure name

Returns

- *Measure (if haz_type and name),*
- *list(Measure) (if haz_type or name),*
- **{Measure.haz_type** (*{Measure.name : Measure}} (if None)*)

`get_hazard_types`(*meas=None*)
Get measures hazard types contained for the name provided. Return all hazard types if no input name.

Parameters **name** (*str, optional*) – measure name

Return type list(str)

`get_names`(*haz_type=None*)
Get measures names contained for the hazard type provided. Return all names for each hazard type if no input hazard type.

Parameters **haz_type** (*str, optional*) – hazard type from which to obtain the names

Returns

- *list(Measure.name) (if haz_type provided),*
- **{Measure.haz_type** (*list(Measure.name)} (if no haz_type)*)

`size`(*haz_type=None*, *name=None*)
Get number of measures contained with input hazard type and /or id. If no input provided, get total number of impact functions.

Parameters

- **haz_type** (*str, optional*) – hazard type
- **name** (*str, optional*) – measure name

Return type int

`check`()
Check instance attributes.

Raises `ValueError` –

`extend`(*meas_set*)
Extend measures of input MeasureSet to current MeasureSet. Overwrite Measure if same name and haz_type.

Parameters **impact_funcs** (*MeasureSet*) – ImpactFuncSet instance to extend

Raises `ValueError` –

classmethod from_mat(*file_name*, *description=''*, *var_names={'field_name': 'measures', 'sup_field_name': 'entity', 'var_name': {'color': 'color', 'cost': 'cost', 'exp_reg': 'Region_ID', 'exp_set': 'assets_file', 'fun_map': 'damagefunctions_map', 'haz': 'peril_ID', 'haz_frq': 'hazard_high_frequency_cutoff', 'haz_int_a': 'hazard_intensity_impact_a', 'haz_int_b': 'hazard_intensity_impact_b', 'haz_set': 'hazard_event_set', 'mdd_a': 'MDD_impact_a', 'mdd_b': 'MDD_impact_b', 'name': 'name', 'paa_a': 'PAA_impact_a', 'paa_b': 'PAA_impact_b', 'risk_att': 'risk_transfer_attachement', 'risk_cov': 'risk_transfer_cover'}}*)

Read MATLAB file generated with previous MATLAB CLIMADA version.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names** (*dict, optional*) – name of the variables in the file

Returns **meas_set** – Measure Set from matlab file

Return type climada.entity.MeasureSet()

read_mat(**args*, ***kwargs*)

This function is deprecated, use MeasureSet.from_mat instead.

classmethod from_excel(*file_name*, *description=''*, *var_names={'col_name': {'color': 'color', 'cost': 'cost', 'exp_reg': 'Region_ID', 'exp_set': 'assets file', 'fun_map': 'damagefunctions map', 'haz': 'peril_ID', 'haz_frq': 'hazard high frequency cutoff', 'haz_int_a': 'hazard intensity impact a', 'haz_int_b': 'hazard intensity impact b', 'haz_set': 'hazard event set', 'mdd_a': 'MDD impact a', 'mdd_b': 'MDD impact b', 'name': 'name', 'paa_a': 'PAA impact a', 'paa_b': 'PAA impact b', 'risk_att': 'risk transfer attachement', 'risk_cov': 'risk transfer cover', 'risk_fact': 'risk transfer cost factor'}, 'sheet_name': 'measures'}*)

Read excel file following template and store variables.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names** (*dict, optional*) – name of the variables in the file

Returns **meas_set** – Measures set from Excel

Return type climada.entity.MeasureSet

read_excel(**args*, ***kwargs*)

This function is deprecated, use MeasureSet.from_excel instead.

write_excel(*file_name*, *var_names={'col_name': {'color': 'color', 'cost': 'cost', 'exp_reg': 'Region_ID', 'exp_set': 'assets file', 'fun_map': 'damagefunctions map', 'haz': 'peril_ID', 'haz_frq': 'hazard high frequency cutoff', 'haz_int_a': 'hazard intensity impact a', 'haz_int_b': 'hazard intensity impact b', 'haz_set': 'hazard event set', 'mdd_a': 'MDD impact a', 'mdd_b': 'MDD impact b', 'name': 'name', 'paa_a': 'PAA impact a', 'paa_b': 'PAA impact b', 'risk_att': 'risk transfer attachement', 'risk_cov': 'risk transfer cover', 'risk_fact': 'risk transfer cost factor'}, 'sheet_name': 'measures'}*)

Write excel file following template.

Parameters

- **file_name** (*str*) – absolute file name to write

- **var_names** (*dict, optional*) – name of the variables in the file

climada.entity.entity_def module

class climada.entity.entity_def.**Entity**(*exposures=None*, *disc_rates=None*, *impact_func_set=None*, *measure_set=None*)

Bases: object

Collects exposures, impact functions, measures and discount rates. Default values set when empty constructor.

exposures
exposures

Type *Exposures*

impact_funcs
impact functions

Type ImpactFucs

measures
measures

Type *MeasureSet*

disc_rates
discount rates

Type *DiscRates*

def_file
Default file from configuration file

Type str

__init__(*exposures=None*, *disc_rates=None*, *impact_func_set=None*, *measure_set=None*)
Initialize entity

Parameters

- **exposures** (*climada.entity.Exposures, optional*) – Exposures of the entity. The default is None (empty Exposures()).
- **disc_rates** (*climada.entity.DiscRates, optional*) – Disc rates of the entity. The default is None (empty DiscRates()).
- **impact_func_set** (*climada.entity.ImpactFuncSet, optional*) – The impact function set. The default is None (empty ImpactFuncSet()).
- **measure_set** (*climada.entity.Measures, optional*) – The measures. The default is None (empty MeasuresSet().

classmethod from_mat(*file_name*, *description=''*)
Read MATLAB file of climada.

Parameters

- **file_name** (*str, optional*) – file name(s) or folder name containing the files to read
- **description** (*str or list(str), optional*) – one description of the data or a description of each data file

Returns **ent** – The entity from matlab file

Return type climada.entity.Entity

read_mat(**args*, ***kwargs*)
This function is deprecated, use Entity.from_mat instead.

classmethod from_excel(*file_name*, *description=''*)
Read csv or xls or xlsx file following climada's template.

Parameters

- **file_name** (*str, optional*) – file name(s) or folder name containing the files to read
- **description** (*str or list(str), optional*) – one description of the data or a description of each data file

Returns **ent** – The entity from excel file

Return type climada.entity.Entity

read_excel(**args*, ***kwargs*)
This function is deprecated, use Entity.from_excel instead.

write_excel(*file_name*)
Write excel file following template.

check()
Check instance attributes.

Raises **ValueError** –

climada.entity.tag module

class climada.entity.tag.**Tag**(*file_name=''*, *description=''*)
Bases: object

Source data tag for Exposures, DiscRates, ImpactFuncSet, MeasureSet.

file_name
name of the source file

Type str

description
description of the data

Type str

__init__(*file_name=''*, *description=''*)
Initialize values.

Parameters

- **file_name** (*str, optional*) – file name to read
- **description** (*str, optional*) – description of the data

append(*tag*)
Append input Tag instance information to current Tag.

7.1.3 climada.hazard package

climada.hazard.centroids package

climada.hazard.centroids.centr module

class climada.hazard.centroids.centr.**Centroids**

Bases: object

Contains raster or vector centroids.

meta
rasterio meta dictionary containing raster properties: width, height, crs and transform must be present at least. The affine ransformation needs to be shearless (only stretching) and have positive x- and negative y-orientation.

Type dict, optional

lat
latitude of size size

Type np.array, optional

lon
longitude of size size

Type np.array, optional

geometry
contains lat and lon crs. Might contain geometry points for lat and lon

Type gpd.GeoSeries, optional

area_pixel
area of size size

Type np.array, optional

dist_coast
distance to coast of size size

Type np.array, optional

on_land
on land (True) and on sea (False) of size size

Type np.array, optional

region_id
country region code of size size

Type np.array, optional

elevation
elevation of size size

Type np.array, optional

vars_check = {'area_pixel', 'dist_coast', 'elevation', 'geometry', 'lat', 'lon', 'on_land', 'region_id'}
Variables whose size will be checked

__init__()
Initialize to None raster and vector. Default crs=DEF_CRS

check()
Check integrity of stored information.

Checks that either *meta* attribute is set, or *lat*, *lon* and *geometry.crs*. Checks sizes of (optional) data attributes.

equal(*centr*)
Return True if two centroids equal, False otherwise

Parameters **centr** (*Centroids*) – centroids to compare

Returns **eq**

Return type bool

static from_base_grid(*land=False*, *res_as=360*, *base_file=None*)
Initialize from base grid data provided with CLIMADA

Parameters

- **land** (*bool, optional*) – If True, restrict to grid points on land. Default: False.
- **res_as** (*int, optional*) – Base grid resolution in arc-seconds (one of 150, 360). Default: 360.
- **base_file** (*str, optional*) – If set, read this file instead of one provided with climada.

classmethod from_geodataframe(*gdf*, *geometry_alias='geom'*)
Create Centroids instance from GeoDataFrame.

The geometry, lat, and lon attributes are set from the GeoDataFrame.geometry attribute, while the columns are copied as attributes to the Centroids object in the form of numpy.ndarrays using pandas.Series.to_numpy. The Series dtype will thus be respected.

Columns named lat or lon are ignored, as they would overwrite the coordinates extracted from the point features. If the geometry attribute bears an alias, it can be dropped by setting the geometry_alias parameter.

If the GDF includes a region_id column, but no on_land column, then on_land=True is inferred for those centroids that have a set region_id.

Example

```
>>> gdf = geopandas.read_file('centroids.shp')
>>> gdf.region_id = gdf.region_id.astype(int)  # type coercion
>>> centroids = Centroids.from_geodataframe(gdf)
```

Parameters

- **gdf** (*GeoDataFrame*) – Where the geometry column needs to consist of point features. See above for details on processing.
- **geometry_alias** (*str, opt*) – Alternate name for the geometry column; dropped to avoid duplicate assignment.

Returns **centr** – Centroids with data from given GeoDataFrame

Return type *Centroids*

set_raster_from_pix_bounds(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_pix_bounds instead.

classmethod from_pix_bounds(*xf_lat*, *xo_lon*, *d_lat*, *d_lon*, *n_lat*, *n_lon*, *crs='EPSG:4326'*)
Create Centroids object with meta attribute according to pixel border data.

Parameters

- **xf_lat** (*float*) – upper latitude (top)
- **xo_lon** (*float*) – left longitude
- **d_lat** (*float*) – latitude step (negative)
- **d_lon** (*float*) – longitude step (positive)
- **n_lat** (*int*) – number of latitude points
- **n_lon** (*int*) – number of longitude points
- **crs** (*dict() or rasterio.crs.CRS, optional*) – CRS. Default: DEF_CRS

Returns **centr** – Centroids with meta according to given pixel border data.

Return type *Centroids*

set_raster_from_pnt_bounds(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_pnt_bounds instead.

classmethod from_pnt_bounds(*points_bounds*, *res*, *crs='EPSG:4326'*)
Create Centroids object with meta attribute according to points border data.

raster border = point border + res/2

Parameters

- **points_bounds** (*tuple*) – points' lon_min, lat_min, lon_max, lat_max
- **res** (*float*) – desired resolution in same units as points_bounds
- **crs** (*dict() or rasterio.crs.CRS, optional*) – CRS. Default: DEF_CRS

Returns **centr** – Centroids with meta according to given points border data.

Return type *Centroids*

set_lat_lon(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_lat_lon instead.

classmethod from_lat_lon(*lat*, *lon*, *crs='EPSG:4326'*)
Create Centroids object from given latitude, longitude and CRS.

Parameters

- **lat** (*np.array*) – latitude
- **lon** (*np.array*) – longitude
- **crs** (*dict() or rasterio.crs.CRS, optional*) – CRS. Default: DEF_CRS

Returns **centr** – Centroids with points according to given coordinates

Return type *Centroids*

set_raster_file(*file_name*, *band=None*, ***kwargs*)
This function is deprecated, use Centroids.from_raster_file and Centroids.values_from_raster_files instead.

`classmethod from_raster_file`(*file_name*, *src_crs=None*, *window=False*, *geometry=False*, *dst_crs=False*, *transform=None*, *width=None*, *height=None*, *resampling=Resampling.nearest*)

Create a new Centroids object from a raster file

Select region using window or geometry. Reproject input by providing dst_crs and/or (transform, width, height).

Parameters

- **file_name** (*str*) – path of the file
- **src_crs** (*crs, optional*) – source CRS. Provide it if error without it.
- **window** (*rasterio.windows.Window, optional*) – window to read
- **geometry** (*shapely.geometry, optional*) – consider pixels only in shape
- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float*) – number of lons for transform
- **height** (*float*) – number of lats for transform
- **resampling** (*rasterio.warp,.Resampling optional*) – resampling function used for reprojection to dst_crs

Returns **centr** – Centroids with meta attribute according to the given raster file

Return type *Centroids*

`values_from_raster_files`(*file_names*, *band=None*, *src_crs=None*, *window=False*, *geometry=False*, *dst_crs=False*, *transform=None*, *width=None*, *height=None*, *resampling=Resampling.nearest*)

Read raster of bands and set 0 values to the masked ones.

Each band is an event. Select region using window or geometry. Reproject input by proving dst_crs and/or (transform, width, height).

Parameters

- **file_names** (*str*) – path of the file
- **band** (*list(int), optional*) – band number to read. Default: [1]
- **src_crs** (*crs, optional*) – source CRS. Provide it if error without it.
- **window** (*rasterio.windows.Window, optional*) – window to read
- **geometry** (*shapely.geometry, optional*) – consider pixels only in shape
- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float*) – number of lons for transform
- **height** (*float*) – number of lats for transform
- **resampling** (*rasterio.warp,.Resampling optional*) – resampling function used for reprojection to dst_crs

Raises `ValueError` –

Returns **inten** – Each row is an event.

Return type scipy.sparse.csr_matrix

`set_vector_file`(*file_name*, *inten_name=None*, ***kwargs*)
This function is deprecated, use Centroids.from_vector_file and Centroids.values_from_vector_files instead.

`classmethod from_vector_file`(*file_name*, *dst_crs=None*)
Create Centroids object from vector file (any format supported by fiona).

Parameters

- **file_name** (*str*) – vector file with format supported by fiona and 'geometry' field.
- **dst_crs** (*crs, optional*) – reproject to given crs

Returns **centr** – Centroids with points according to the given vector file

Return type *Centroids*

`values_from_vector_files`(*file_names*, *val_names=None*, *dst_crs=None*)
Read intensity or other data from vector files, making sure that geometry is compatible.

If the geometry of the shapes in any of the given files does not agree with the geometry of this Centroids instance, a ValueError is raised.

Parameters

- **file_names** (*list(str)*) – vector files with format supported by fiona and 'geometry' field.
- **val_names** (*list(str), optional*) – list of names of the columns of the values. Default: ['intensity']
- **dst_crs** (*crs, optional*) – reproject to given crs

Raises `ValueError` –

Returns **values** – Sparse array of shape (len(val_name), len(geometry)).

Return type scipy.sparse.csr_matrix

`read_mat`(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_mat instead.

`classmethod from_mat`(*file_name*, *var_names=None*)
Read centroids from CLIMADA's MATLAB version.

Parameters

- **file_name** (*str*) – absolute or relative file name
- **var_names** (*dict, optional*) – name of the variables

Raises `KeyError` –

Returns **centr** – Centroids with data from the given file

Return type *Centroids*

`read_excel`(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_excel instead.

`classmethod from_excel`(*file_name*, *var_names=None*)
Generate a new centroids object from an excel file with column names in var_names.

Parameters

- **file_name** (*str*) – absolute or relative file name

- **var_names** (*dict, default*) – name of the variables

Raises `KeyError` –

Returns **centr** – Centroids with data from the given file

Return type *Centroids*

`clear()`
Clear vector and raster data.

`append(centr)`
Append centroids points.

If centr or self are rasters they are converted to points first using Centroids.set_meta_to_lat_lon. Note that self is modified in-place, and meta is set to {}. Thus, raster information in self is lost.

Note: this is a wrapper for centroids.union.

Parameters **centr** (*Centroids*) – Centroids to append. The centroids need to have the same CRS.

See also:

`union` Union of Centroid objects.

`union(*others)`
Create the union of centroids from the inputs.

The centroids are combined together point by point. Rasters are converted to points and raster information is lost in the output. All centroids must have the same CRS.

In any case, the attribute .geometry is computed for all centroids. This requires a CRS to be defined. If Centroids.crs is None, the default DEF_CRS is set for all centroids (self and others).

When at least one centroids has one of the following property defined, it is also computed for all others. .area_pixel, .dist_coast, .on_land, .region_id, .elevetaion'

!Caution!: the input objects (self and others) are modified in place. Missing properties are added, existing ones are not overwritten.

Parameters **others** (*any number of climada.hazard.Centroids()*) – Centroids to form the union with

Returns **centroids** – Centroids containing the union of the centroids in others.

Return type *Centroids*

Raises `ValueError` –

`get_closest_point(x_lon, y_lat, scheduler=None)`
Returns closest centroid and its index to a given point.

Parameters

- **x_lon** (*float*) – x coord (lon)
- **y_lat** (*float*) – y coord (lat)
- **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

Returns

- **x_close** (*float*) – x-coordinate (longitude) of closest centroid.

- **y_close** (*float*) – y-coordinate (latitude) of closest centroids.
- **idx_close** (*int*) – Index of centroid in internal ordering of centroids.

`set_region_id`(*scheduler=None*)
Set region_id as country ISO numeric code attribute for every pixel or point.

Parameters **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

`set_area_pixel`(*min_resol=1e-08*, *scheduler=None*)
Set *area_pixel* attribute for every pixel or point (area in m*m).

Parameters

- **min_resol** (*float, optional*) – if centroids are points, use this minimum resolution in lat and lon. Default: 1.0e-8
- **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

`set_area_approx`(*min_resol=1e-08*)
Set *area_pixel* attribute for every pixel or point (approximate area in m*m).

Values are differentiated per latitude. Faster than *set_area_pixel*.

Parameters **min_resol** (*float, optional*) – if centroids are points, use this minimum resolution in lat and lon. Default: 1.0e-8

`set_elevation`(*topo_path*)
Set elevation attribute for every pixel or point in meters.

Parameters **topo_path** (*str*) – Path to a raster file containing gridded elevation data.

`set_dist_coast`(*signed=False*, *precomputed=False*, *scheduler=None*)
Set dist_coast attribute for every pixel or point in meters.

Parameters

- **signed** (*bool*) – If True, use signed distances (positive off shore and negative on land). Default: False.
- **precomputed** (*bool*) – If True, use precomputed distances (from NASA). Default: False.
- **scheduler** (*str*) – Used for dask map_partitions. "threads", "synchronous" or "processes"

`set_on_land`(*scheduler=None*)
Set on_land attribute for every pixel or point.

Parameters **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

`remove_duplicate_points`(*scheduler=None*)
Return Centroids with removed duplicated points

Parameters **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

Returns **cen** – Sub-selection of this object.

Return type *Centroids*

`select`(*reg_id=None*, *extent=None*, *sel_cen=None*)
Return Centroids with points in the given reg_id or within mask

Parameters

- **reg_id** (*int*) – region to filter according to region_id values
- **extent** (*tuple*) – Format (min_lon, max_lon, min_lat, max_lat) tuple. If min_lon > lon_max, the extend crosses the antimeridian and is [lon_max, 180] + [-180, lon_min] Borders are inclusive.
- **sel_cen** (*np.array*) – 1-dim mask, overrides reg_id and extent

Returns **cen** – Sub-selection of this object

Return type *Centroids*

`select_mask`(*reg_id=None*, *extent=None*)

Make mask of selected centroids

Parameters

- **reg_id** (*int*) – region to filter according to region_id values
- **extent** (*tuple*) – Format (min_lon, max_lon, min_lat, max_lat) tuple. If min_lon > lon_max, the extend crosses the antimeridian and is [lon_max, 180] + [-180, lon_min] Borders are inclusive.

Returns **sel_cen** – 1d mask of selected centroids

Return type 1d array of booleans

`set_lat_lon_to_meta`(*min_resol=1e-08*)

Compute meta from lat and lon values.

Parameters **min_resol** (*float, optional*) – Minimum centroids resolution to use in the raster. Default: 1.0e-8.

`set_meta_to_lat_lon`()

Compute lat and lon of every pixel center from meta raster.

`plot`(*axis=None*, *figsize=(9, 13)*, ***kwargs*)

Plot centroids scatter points over earth.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** ((*float, float), optional*) – figure size for plt.subplots The default is (9, 13)
- **kwargs** (*optional*) – arguments for scatter matplotlib function

Returns **axis**

Return type matplotlib.axes._subplots.AxesSubplot

`calc_pixels_polygons`(*scheduler=None*)

Return a gpd.GeoSeries with a polygon for every pixel

Parameters **scheduler** (*str*) – used for dask map_partitions. “threads”, “synchronous” or “processes”

Returns **geo**

Return type gpd.GeoSeries

`empty_geometry_points`()

Removes all points in geometry.

Useful when centroids is used in multiprocessing function.

write_hdf5(*file_data*)
Write centroids attributes into hdf5 format.

Parameters **file_data** (*str or h5*) – If string, path to write data. If h5 object, the datasets will be generated there.

read_hdf5(**args*, ***kwargs*)
This function is deprecated, use Centroids.from_hdf5 instead.

classmethod from_hdf5(*file_data*)
Create a centroids object from a HDF5 file.

Parameters **file_data** (*str or h5*) – If string, path to read data. If h5 object, the datasets will be read from there.

Returns **centr** – Centroids with data from the given file

Return type *Centroids*

property crs
Get CRS of raster or vector.

property size
Get number of pixels or points.

property shape
Get shape of rastered data.

property total_bounds
Get total bounds (left, bottom, right, top).

property coord
Get [lat, lon] array.

set_geometry_points(*scheduler=None*)
Set *geometry* attribute with Points from *lat*/*lon* attributes.

Parameters **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

climada.hazard.base module

class climada.hazard.base.**Hazard**(*haz_type=''*, *pool=None*)
Bases: object

Contains events of some hazard type defined at centroids. Loads from files with format defined in FILE_EXT.

tag
information about the source

Type TagHazard

units
units of the intensity

Type str

centroids
centroids of the events

Type *Centroids*

event_id
id (>0) of each event

Type np.array

event_name
name of each event (default: event_id)

Type list(str)

date
integer date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1 (ordinal format of datetime library)

Type np.array

orig
flags indicating historical events (True) or probabilistic (False)

Type np.array

frequency
frequency of each event in years

Type np.array

intensity
intensity of the events at centroids

Type sparse.csr_matrix

fraction
fraction of affected exposures for each event at each centroid

Type sparse.csr_matrix

intensity_thres = 10
Intensity threshold per hazard used to filter lower intensities. To be set for every hazard type

vars_oblig = {'centroids', 'event_id', 'fraction', 'frequency', 'intensity', 'tag', 'units'}
scalar, str, list, 1dim np.array of size num_events, scipy.sparse matrix of shape num_events x num_centroids, Centroids and Tag.

Type Name of the variables needed to compute the impact. Types

vars_def = {'date', 'event_name', 'orig'}
Name of the variables used in impact calculation whose value is descriptive and can therefore be set with default values. Types: scalar, string, list, 1dim np.array of size num_events.

vars_opt = {}
Name of the variables that aren't need to compute the impact. Types: scalar, string, list, 1dim np.array of size num_events.

__init__(*haz_type=''*, *pool=None*)
Initialize values.

Parameters

- **haz_type** (*str, optional*) – acronym of the hazard type (e.g. 'TC').
- **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. Default: None

Examples

Fill hazard values by hand:

```
>>> haz = Hazard('TC')
>>> haz.intensity = sparse.csr_matrix(np.zeros((2, 2)))
>>> ...
```

Take hazard values from file:

```
>>> haz = Hazard.from_mat(HAZ_DEMO_MAT, 'demo')
```

clear()
Reinitialize attributes (except the process Pool).

check()
Check dimension of attributes.

Raises **ValueError** –

classmethod from_raster(*files_intensity*, *files_fraction=None*, *attrs=None*, *band=None*, *haz_type=None*, *pool=None*, *src_crs=None*, *window=False*, *geometry=False*, *dst_crs=False*, *transform=None*, *width=None*, *height=None*, *resampling=Resampling.nearest*)
Create Hazard with intensity and fraction values from raster files

If raster files are masked, the masked values are set to 0.

Files can be partially read using either window or geometry. Additionally, the data is reprojected when custom dst_crs and/or transform, width and height are specified.

Parameters

- **files_intensity** (*list(str)*) – file names containing intensity
- **files_fraction** (*list(str)*) – file names containing fraction
- **attrs** (*dict, optional*) – name of Hazard attributes and their values
- **band** (*list(int), optional*) – bands to read (starting at 1), default [1]
- **haz_type** (*str, optional*) – acronym of the hazard type (e.g. 'TC'). Default: None, which will use the class default (" for vanilla *Hazard* objects, and hard coded in some subclasses)
- **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. Default: None
- **src_crs** (*crs, optional*) – source CRS. Provide it if error without it.
- **window** (*rasterio.windows.Windows, optional*) – window where data is extracted
- **geometry** (*shapely.geometry, optional*) – consider pixels only in shape
- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float, optional*) – number of lons for transform
- **height** (*float, optional*) – number of lats for transform
- **resampling** (*rasterio.warp.Resampling, optional*) – resampling function used for reprojection to dst_crs

Return type *Hazard*

`set_raster`(**args*, ***kwargs*)
This function is deprecated, use Hazard.from_raster.

`set_vector`(**args*, ***kwargs*)
This function is deprecated, use Hazard.from_vector.

`classmethod from_vector`(*files_intensity*, *files_fraction=None*, *attrs=None*, *inten_name=None*, *frac_name=None*, *dst_crs=None*, *haz_type=None*)
Read vector files format supported by fiona. Each intensity name is considered an event.

Parameters

- **files_intensity** (*list(str)*) – file names containing intensity, default: ['intensity']
- **files_fraction** ((*list(str)*)) – file names containing fraction, default: ['fraction']
- **attrs** (*dict, optional*) – name of Hazard attributes and their values
- **inten_name** (*list(str), optional*) – name of variables containing the intensities of each event
- **frac_name** (*list(str), optional*) – name of variables containing the fractions of each event
- **dst_crs** (*crs, optional*) – reproject to given crs
- **haz_type** (*str, optional*) – acronym of the hazard type (e.g. 'TC'). default: None, which will use the class default (" for vanilla *Hazard* objects, hard coded in some subclasses)

Returns **haz** – Hazard from vector file

Return type climada.hazard.Hazard

`reproject_raster`(*dst_crs=False*, *transform=None*, *width=None*, *height=None*, *resampl_inten=Resampling.nearest*, *resampl_fract=Resampling.nearest*)
Change current raster data to other CRS and/or transformation

Parameters

- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float*) – number of lons for transform
- **height** (*float*) – number of lats for transform
- **resampl_inten** (*rasterio.warp,.Resampling optional*) – resampling function used for reprojection to dst_crs for intensity
- **resampl_fract** (*rasterio.warp,.Resampling optional*) – resampling function used for reprojection to dst_crs for fraction

`reproject_vector`(*dst_crs*, *scheduler=None*)
Change current point data to a a given projection

Parameters

- **dst_crs** (*crs*) – reproject to given crs
- **scheduler** (*str, optional*) – used for dask map_partitions. "threads", "synchronous" or "processes"

raster_to_vector()
Change current raster to points (center of the pixels)

vector_to_raster(*scheduler=None*)
Change current point data to a raster with same resolution

Parameters **scheduler** (*str, optional*) – used for dask map_partitions. "threads", "synchronous" or "processes"

read_mat(**args*, ***kwargs*)
This function is deprecated, use Hazard.from_mat.

classmethod from_mat(*file_name*, *description=''*, *var_names=None*)
Read climada hazard generate with the MATLAB code in .mat format.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names** (*dict, optional*) – name of the variables in the file, default: DEF_VAR_MAT constant

Returns **haz** – Hazard object from the provided MATLAB file

Return type climada.hazard.Hazard

Raises **KeyError** –

read_excel(**args*, ***kwargs*)
This function is deprecated, use Hazard.from_excel.

classmethod from_excel(*file_name*, *description=''*, *var_names=None*, *haz_type=None*)
Read climada hazard generated with the MATLAB code in Excel format.

Parameters

- **file_name** (*str*) – absolute file name
- **description** (*str, optional*) – description of the data
- **var_names (dict, default)** (*name of the variables in the file,*) – default: DEF_VAR_EXCEL constant
- **haz_type** (*str, optional*) – acronym of the hazard type (e.g. 'TC'). Default: None, which will use the class default ('' for vanilla *Hazard* objects, and hard coded in some subclasses)

Returns **haz** – Hazard object from the provided Excel file

Return type climada.hazard.Hazard

Raises **KeyError** –

select(*event_names=None*, *date=None*, *orig=None*, *reg_id=None*, *extent=None*, *reset_frequency=False*)
Select events matching provided criteria

The frequency of events may need to be recomputed (see *reset_frequency*)!

Parameters

- **event_names** (*list of str, optional*) – Names of events.
- **date** (*array-like of length 2 containing str or int, optional*) – (initial date, final date) in string ISO format ('2011-01-02') or datetime ordinal integer.

- **orig** (*bool, optional*) – Select only historical (True) or only synthetic (False) events.
- **reg_id** (*int, optional*) – Region identifier of the centroids' region_id attibute.
- **extent** (*tuple(float, float, float, float), optional*) – Extent of centroids as (min_lon, max_lon, min_lat, max_lat). The default is None.
- **reset_frequency** (*bool, optional*) – Change frequency of events proportional to difference between first and last year (old and new). Default: False.

Returns **haz** – If no event matching the specified criteria is found, None is returned.

Return type *Hazard* or None

`select_tight`(*buffer=0.8999280057595392*, *val='intensity'*)

Reduce hazard to those centroids spanning a minimal box which contains all non-zero intensity or fraction points.

Parameters

- **buffer** (*float, optional*) – Buffer of box in the units of the centroids. The default is approximately equal to the default threshold from the assign_centroids method (works if centroids in lat/lon)
- **val** (*string, optional*) – Select tight by non-zero 'intensity' or 'fraction'. The default is 'intensity'.

Returns Copy of the Hazard with centroids reduced to minimal box. All other hazard properties are carried over without changes.

Return type *Hazard*

See also:

`self.select` Method to select centroids by lat/lon extent

`util.coordinates.assign_coordinates` algorithm to match centroids.

`local_exceedance_inten`(*return_periods=(25, 50, 100, 250)*)

Compute exceedance intensity map for given return periods.

Parameters **return_periods** (*np.array*) – return periods to consider

Returns **inten_stats**

Return type np.array

`plot_rp_intensity`(*return_periods=(25, 50, 100, 250)*, *smooth=True*, *axis=None*, *figsize=(9, 13)*, *adapt_fontsize=True*, ***kwargs*)

Compute and plot hazard exceedance intensity maps for different return periods. Calls local_exceedance_inten.

Parameters

- **return_periods** (*tuple(int), optional*) – return periods to consider
- **smooth** (*bool, optional*) – smooth plot to plot.RESOLUTIONxplot.RESOLUTION
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*tuple, optional*) – figure size for plt.subplots
- **kwargs** (*optional*) – arguments for pcolormesh matplotlib function used in event plots

Returns **axis, inten_stats** – intenstats is return_periods.size x num_centroids

Return type matplotlib.axes._subplots.AxesSubplot, np.ndarray

`plot_intensity`(*event=None*, *centr=None*, *smooth=True*, *axis=None*, *adapt_fontsize=True*, ***kwargs*)
Plot intensity values for a selected event or centroid.

Parameters

- **event** (*int or str, optional*) – If event > 0, plot intensities of event with id = event. If event = 0, plot maximum intensity in each centroid. If event < 0, plot abs(event)-largest event. If event is string, plot events with that name.
- **centr** (*int or tuple, optional*) – If centr > 0, plot intensity of all events at centroid with id = centr. If centr = 0, plot maximum intensity of each event. If centr < 0, plot abs(centr)-largest centroid where higher intensities are reached. If tuple with (lat, lon) plot intensity of nearest centroid.
- **smooth** (*bool, optional*) – Rescale data to RESOLUTIONxRESOLUTION pixels (see constant in module *climada.util.plot*)
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for pcolormesh matplotlib function used in event plots or for plot function used in centroids plots

Return type matplotlib.axes._subplots.AxesSubplot

Raises `ValueError` –

`plot_fraction`(*event=None*, *centr=None*, *smooth=True*, *axis=None*, ***kwargs*)
Plot fraction values for a selected event or centroid.

Parameters

- **event** (*int or str, optional*) – If event > 0, plot fraction of event with id = event. If event = 0, plot maximum fraction in each centroid. If event < 0, plot abs(event)-largest event. If event is string, plot events with that name.
- **centr** (*int or tuple, optional*) – If centr > 0, plot fraction of all events at centroid with id = centr. If centr = 0, plot maximum fraction of each event. If centr < 0, plot abs(centr)-largest centroid where highest fractions are reached. If tuple with (lat, lon) plot fraction of nearest centroid.
- **smooth** (*bool, optional*) – Rescale data to RESOLUTIONxRESOLUTION pixels (see constant in module *climada.util.plot*)
- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **kwargs** (*optional*) – arguments for pcolormesh matplotlib function used in event plots or for plot function used in centroids plots

Return type matplotlib.axes._subplots.AxesSubplot

Raises `ValueError` –

`sanitize_event_ids`()
Make sure that event ids are unique

`get_event_id`(*event_name*)
Get an event id from its name. Several events might have the same name.

Parameters **event_name** (*str*) – Event name

Returns **list_id**

Return type np.array(int)

`get_event_name`(*event_id*)
Get the name of an event id.

Parameters **event_id** (*int*) – id of the event

Return type str

Raises `ValueError` –

`get_event_date`(*event=None*)
Return list of date strings for given event or for all events, if no event provided.

Parameters **event** (*str or int, optional*) – event name or id.

Returns **l_dates**

Return type list(str)

`calc_year_set`()
From the dates of the original events, get number yearly events.

Returns **orig_yearset** – key are years, values array with event_ids of that year

Return type dict

`remove_duplicates`()
Remove duplicate events (events with same name and date).

`set_frequency`(*yearrange=None*)
Set hazard frequency from yearrange or intensity matrix.

Parameters **yearrange** (*tuple or list, optional*) – year range to be used to compute frequency per event. If yearrange is not given (None), the year range is derived from self.date

`property size`
Return number of events.

`write_raster`(*file_name*, *intensity=True*)
Write intensity or fraction as GeoTIFF file. Each band is an event

Parameters

- **file_name** (*str*) – file name to write in tif format
- **intensity** (*bool*) – if True, write intensity, otherwise write fraction

`write_hdf5`(*file_name*, *todense=False*)
Write hazard in hdf5 format.

Parameters **file_name** (*str*) – file name to write, with h5 format

`read_hdf5`(**args*, ***kwargs*)
This function is deprecated, use Hazard.from_hdf5.

`classmethod from_hdf5`(*file_name*)
Read hazard in hdf5 format.

Parameters **file_name** (*str*) – file name to read, with h5 format

Returns **haz** – Hazard object from the provided MATLAB file

Return type climada.hazard.Hazard

`append`(**others*)
Append the events and centroids to this hazard object.

All of the given hazards must be of the same type and use the same units as self. The centroids of all hazards must have the same CRS.

The following kinds of object attributes are processed:

- All centroids are combined together using *Centroids.union*.
- Lists, 1-dimensional arrays (NumPy) and sparse CSR matrices (SciPy) are concatenated.

Sparse matrices are concatenated along the first (vertical) axis.

- All *tag* attributes are appended to *self.tag*.

For any other type of attribute: A ValueError is raised if an attribute of that name is not defined in all of the non-empty hazards at least. However, there is no check that the attribute value is identical among the given hazard objects. The initial attribute value of *self* will not be modified.

Note: Each of the hazard's *centroids* attributes might be modified in place in the sense that missing properties are added, but existing ones are not overwritten. In case of raster centroids, conversion to point centroids is applied so that raster information (meta) is lost. For more information, see *Centroids.union*.

Parameters **others** (*one or more climada.hazard.Hazard objects*) – Hazard instances to append to self

Raises `TypeError, ValueError` –

See also:

`Hazard.concat` concatenate 2 or more hazards

`Centroids.union` combine centroids

`classmethod concat`(*haz_list*)

Concatenate events of several hazards of same type.

This function creates a new hazard of the same class as the first hazard in the given list and then applies the *append* method. Please refer to the docs of *Hazard.append* for caveats and limitations of the concatenation procedure.

For centroids, tags, lists, arrays and sparse matrices, the remarks in *Hazard.append* apply. All other attributes are copied from the first object in *haz_list*.

Note that *Hazard.concat* can be used to concatenate hazards of a subclass. The result's type will be the subclass. However, calling *concat([])* (with an empty list) is equivalent to instantiation without init parameters. So, *Hazard.concat([])* is equivalent to *Hazard()*. If *HazardB* is a subclass of *Hazard*, then *HazardB.concat([])* is equivalent to *HazardB()* (unless *HazardB* overrides the *concat* method).

Parameters **haz_list** (*list of climada.hazard.Hazard objects*) – Hazard instances of the same hazard type (subclass).

Returns **haz_concat** – This will be of the same type (subclass) as all the hazards in *haz_list*.

Return type instance of climada.hazard.Hazard

See also:

`Hazard.append` append hazards to a hazard in place

`Centroids.union` combine centroids

`change_centroids`(*centroids*, *threshold=100*)

Assign (new) centroids to hazard.

Centoids of the hazard not in centroids are mapped onto the nearest point. Fails if a point is further than threshold from the closest centroid.

The centroids must have the same CRS as self.centroids.

Parameters

- **haz** (*Hazard*) – Hazard instance
- **centroids** (*Centroids*) – Centroids instance on which to map the hazard.
- **threshold** (*int or float*) – Threshold (in km) for mapping haz.centroids not in centroids. Argument is passed to climada.util.coordinates.assign_coordinates. Default: 100 (km)

Returns **haz_new_cent** – Hazard projected onto centroids

Return type *Hazard*

Raises `ValueError` –

See also:

`util.coordinates.assign_coordinates` algorithm to match centroids.

climada.hazard.isimip_data module

climada.hazard.storm_europe module

`class` climada.hazard.storm_europe.`StormEurope`

Bases: *climada.hazard.base.Hazard*

A hazard set containing european winter storm events. Historic storm events can be downloaded at http://wisc.climate.copernicus.eu/ and read with *from_footprints*. Weather forecasts can be automatically downloaded from https://opendata.dwd.de/ and read with from_icon_grib(). Weather forecast from the COSMO-Consortium http://www.cosmo-model.org/ can be read with from_cosmoe_file().

`ssi_wisc`
Storm Severity Index (SSI) as recorded in the footprint files; apparently not reproducible from the footprint values only.

Type np.array, float

`ssi`
SSI as set by set_ssi; uses the Dawkins definition by default.

Type np.array, float

`intensity_thres = 14.7`
Intensity threshold for storage in m/s; same as used by WISC SSI calculations.

`vars_opt = {'ssi', 'ssi_full_area', 'ssi_wisc'}`
Name of the variables that aren't need to compute the impact.

`__init__`()
Calls the Hazard init dunder. Sets unit to 'm/s'.

`read_footprints`(**args*, ***kwargs*)
This function is deprecated, use StormEurope.from_footprints instead.

`classmethod from_footprints`(*path*, *description=None*, *ref_raster=None*, *centroids=None*, *files_omit='fp_era20c_1990012515_701_0.nc'*, *combine_threshold=None*, *intensity_thres=None*)
Create new StormEurope object from WISC footprints.

Assumes that all footprints have the same coordinates as the first file listed/first file in dir.

Parameters

- **path** (*str, list(str)*) – A location in the filesystem. Either a path to a single netCDF WISC footprint, or a folder containing only footprints, or a globbing pattern to one or more footprints.
- **description** (*str, optional*) – description of the events, defaults to 'WISC historical hazard set'
- **ref_raster** (*str, optional*) – Reference netCDF file from which to construct a new bare-bones Centroids instance. Defaults to the first file in path.
- **centroids** (*Centroids, optional*) – A Centroids struct, overriding ref_raster
- **files_omit** (*str, list(str), optional*) – List of files to omit; defaults to one duplicate storm present in the WISC set as of 2018-09-10.
- **combine_threshold** (*int, optional*) – threshold for combining events in number of days. if the difference of the dates (self.date) of two events is smaller or equal to this threshold, the two events are combined into one. Default is None, Advised for WISC is 2
- **intensity_thres** (*float, optional*) – Intensity threshold for storage in m/s. Default: class attribute StormEurope.intensity_thres (same as used by WISC SSI calculations)

Returns **haz** – StormEurope object with data from WISC footprints.

Return type *StormEurope*

`read_cosmoe_file`(**args*, ***kwargs*)

This function is deprecated, use StormEurope.from_cosmoe_file instead.

`classmethod from_cosmoe_file`(*fp_file*, *run_datetime*, *event_date=None*, *model_name='COSMO-2E'*, *description=None*, *intensity_thres=None*)

Create a new StormEurope object with gust footprint from weather forecast.

The funciton is designed for the COSMO ensemble model used by the COSMO Consortium http://www.cosmo-model.org/ and postprocessed to an netcdf file using fieldextra. One event is one full day in UTC. Works for MeteoSwiss model output of COSMO-1E (11 members, resolution 1.1 km, forecast period 33-45 hours) COSMO-2E (21 members, resolution 2.2 km, forecast period 5 days)

The frequency of each event is informed by their probability in the ensemble forecast and is equal to 1/11 or 1/21 for COSMO-1E or COSMO-2E, respectively.

Parameters

- **fp_file** (*str*) – string directing to one netcdf file
- **run_datetime** (*datetime*) – The starting timepoint of the forecast run of the cosmo model
- **event_date** (*datetime, optional*) – one day within the forecast period, only this day (00H-24H) will be included in the hazard
- **model_name** (*str,optional*) – provide the name of the COSMO model, for the description (e.g., 'COSMO-1E', 'COSMO-2E')
- **description** (*str, optional*) – description of the events, defaults to a combination of model_name and run_datetime
- **intensity_thres** (*float, optional*) – Intensity threshold for storage in m/s. Default: class attribute StormEurope.intensity_thres (same as used by WISC SSI calculations)

Returns **haz** – StormEurope object with data from COSMO ensemble file.

Return type *StormEurope*

read_icon_grib(**args*, ***kwargs*)
This function is deprecated, use StormEurope.from_icon_grib instead.

classmethod from_icon_grib(*run_datetime*, *event_date=None*, *model_name='icon-eu-eps'*, *description=None*, *grib_dir=None*, *delete_raw_data=True*, *intensity_thres=None*)
Create new StormEurope object from DWD icon weather forecast footprints.

New files are available for 24 hours on https://opendata.dwd.de, old files can be processed if they are already stored in grib_dir. One event is one full day in UTC. Current setup works for runs starting at 00H and 12H. Otherwise the aggregation is inaccurate, because of the given file structure with 1-hour, 3-hour and 6-hour maxima provided.

The frequency for one event is 1/(number of ensemble members)

Parameters

- **run_datetime** (*datetime*) – The starting timepoint of the forecast run of the icon model
- **event_date** (*datetime, optional*) – one day within the forecast period, only this day (00H-24H) will be included in the hazard
- **model_name** (*str,optional*) – select the name of the icon model to be downloaded. Must match the url on https://opendata.dwd.de (see download_icon_grib for further info)
- **description** (*str, optional*) – description of the events, defaults to a combination of model_name and run_datetime
- **grib_dir** (*str, optional*) – path to folder, where grib files are or should be stored
- **delete_raw_data** (*bool,optional*) – select if downloaded raw data in .grib.bz2 file format should be stored on the computer or removed
- **intensity_thres** (*float, optional*) – Intensity threshold for storage in m/s. Default: class attribute StormEurope.intensity_thres (same as used by WISC SSI calculations)

Returns **haz** – StormEurope object with data from DWD icon weather forecast footprints.

Return type *StormEurope*

calc_ssi(*method='dawkins'*, *intensity=None*, *on_land=True*, *threshold=None*, *sel_cen=None*)
Calculate the SSI, method must either be 'dawkins' or 'wisc_gust'.

'dawkins', after Dawkins et al. (2016), doi:10.5194/nhess-16-1999-2016, matches the MATLAB version. ssi = sum_i(area_cell_i * intensity_cell_i^3)

'wisc_gust', according to the WISC Tier 1 definition found at https://wisc.climate.copernicus.eu/wisc/#/help/products#tier1_section ssi = sum(area_on_land) * mean(intensity)^3

In both definitions, only raster cells that are above the threshold are used in the computation. Note that this method does not reproduce self.ssi_wisc, presumably because the footprint only contains the maximum wind gusts instead of the sustained wind speeds over the 72 hour window. The deviation may also be due to differing definitions of what lies on land (i.e. Syria, Russia, Northern Africa and Greenland are exempt).

Parameters

- **method** (*str*) – Either 'dawkins' or 'wisc_gust'
- **intensity** (*scipy.sparse.csr*) – Intensity matrix; defaults to self.intensity

- **on_land** (*bool*) – Only calculate the SSI for areas on land, ignoring the intensities at sea. Defaults to true, whereas the MATLAB version did not.
- **threshold** (*float, optional*) – Intensity threshold used in index definition. Cannot be lower than the read-in value.
- **sel_cen** (*np.array, bool*) – A boolean vector selecting centroids. Takes precendence over on_land.

`self.ssi_dawkins`
SSI per event

Type np.array

`set_ssi`(***kwargs*)
Wrapper around calc_ssi for setting the self.ssi attribute.

Parameters **kwargs** – passed on to calc_ssi

`ssi`
SSI per event

Type np.array

`plot_ssi`(*full_area=False*)

Plot the distribution of SSIs versus their cumulative exceedance frequencies, highlighting historical storms in red.

Returns

- **fig** (*matplotlib.figure.Figure*)
- **ax** (*matplotlib.axes._subplots.AxesSubplot*)

`generate_prob_storms`(*reg_id=528*, *spatial_shift=4*, *ssi_args=None*, ***kwargs*)
Generates a new hazard set with one original and 29 probabilistic storms per historic storm. This represents a partial implementation of the Monte-Carlo method described in section 2.2 of Schwierz et al. (2010), doi:10.1007/s10584-009-9712-1. It omits the rotation of the storm footprints, as well as the pseudo- random alterations to the intensity.

In a first step, the original intensity and five additional intensities are saved to an array. In a second step, those 6 possible intensity levels are shifted by n raster pixels into each direction (N/S/E/W).

Caveats:

- Memory safety is an issue; trial with the entire dataset resulted in 60GB of swap memory being used…
- Can only use numeric region_id for country selection
- Drops event names as provided by WISC

Parameters

- **region_id** (*int, list of ints, or None*) – iso_n3 code of the countries we want the generated hazard set to be returned for.
- **spatial_shift** (*int*) – amount of raster pixels to shift by
- **ssi_args** (*dict*) – A dictionary of arguments passed to calc_ssi
- **kwargs** – keyword arguments passed on to self._hist2prob()

Returns **new_haz** – A new hazard set for the given country. Centroid attributes are preserved. self.orig attribute is set to True for original storms (event_id ending in 00). Also contains a ssi_prob attribute,

Return type *StormEurope*

climada.hazard.tag module

class climada.hazard.tag.**Tag**(*haz_type=''*, *file_name=''*, *description=''*)

Bases: object

Contain information used to tag a Hazard.

file_name
name of the source file(s)

Type str or list(str)

haz_type
acronym defining the hazard type (e.g. 'TC')

Type str

description
description(s) of the data

Type str or list(str)

__init__(*haz_type=''*, *file_name=''*, *description=''*)
Initialize values.

Parameters

- **haz_type** (*str, optional*) – acronym of the hazard type (e.g. 'TC').
- **file_name** (*str or list(str), optional*) – file name(s) to read
- **description** (*str or list(str), optional*) – description of the data

append(*tag*)
Append input Tag instance information to current Tag.

join_file_names()
Get a string with the joined file names.

join_descriptions()
Get a string with the joined descriptions.

climada.hazard.tc_clim_change module

climada.hazard.tc_clim_change.**TOT_RADIATIVE_FORCE = PosixPath('/home/docs/climada/data/rcp_db.xls')**
//www.iiasa.ac.at/web-apps/tnt/RcpDb. generated: 2018-07-04 10:47:59.

Type © RCP Database (Version 2.0.5) http

climada.hazard.tc_clim_change.**get_knutson_criterion**()
Fill changes in TCs according to Knutson et al. 2015 Global projections of intense tropical cyclone activity for the late twenty-first century from dynamical downscaling of CMIP5/RCP4.5 scenarios.

Returns **criterion** – list of the criterion dictionary for frequency and intensity change per basin, per category taken from the Table 3 in Knutson et al. 2015. with items 'basin' (str), 'category' (list(int)), 'year' (int), 'change' (float), 'variable' ('intensity' or 'frequency')

Return type list(dict)

climada.hazard.tc_clim_change.**calc_scale_knutson**(*ref_year=2050*, *rcp_scenario=45*)

Comparison 2081-2100 (i.e., late twenty-first century) and 2001-20 (i.e., present day). Late twenty-first century effects on intensity and frequency per Saffir-Simpson-category and ocean basin is scaled to target year and target RCP proportional to total radiative forcing of the respective RCP and year.

Parameters

- **ref_year** (*int, optional*) – year between 2000 ad 2100. Default: 2050
- **rcp_scenario** (*int, optional*) – 26 for RCP 2.6, 45 for RCP 4.5. The default is 45 60 for RCP 6.0 and 85 for RCP 8.5.

Returns **factor** – factor to scale Knutson parameters to the give RCP and year

Return type float

climada.hazard.tc_tracks module

climada.hazard.tc_tracks.**CAT_NAMES = {-1: 'Tropical Depression', 0: 'Tropical Storm', 1: 'Hurricane Cat. 1', 2: 'Hurricane Cat. 2', 3: 'Hurricane Cat. 3', 4: 'Hurricane Cat. 4', 5: 'Hurricane Cat. 5'}**

Saffir-Simpson category names.

climada.hazard.tc_tracks.**SAFFIR_SIM_CAT = [34, 64, 83, 96, 113, 137, 1000]**

Saffir-Simpson Hurricane Wind Scale in kn based on NOAA

class climada.hazard.tc_tracks.**TCTracks**(*pool=None*)

Bases: `object`

Contains tropical cyclone tracks.

data

List of tropical cyclone tracks. Each track contains following attributes:

- time (coords)
- lat (coords)
- lon (coords)
- time_step (in hours)
- radius_max_wind (in nautical miles)
- radius_oci (in nautical miles)
- max_sustained_wind (in knots)
- central_pressure (in hPa/mbar)
- environmental_pressure (in hPa/mbar)
- basin (for each track position)
- max_sustained_wind_unit (attrs)
- central_pressure_unit (attrs)

- name (attrs)
- sid (attrs)
- orig_event_flag (attrs)
- data_provider (attrs)
- id_no (attrs)
- category (attrs)

Computed during processing:

- on_land (bool for each track position)
- dist_since_lf (in km)

Type list(xarray.Dataset)

`__init__`(*pool=None*)
Create new (empty) TCTracks instance.

Parameters **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. Default: None

`append`(*tracks*)
Append tracks to current.

Parameters **tracks** (*xarray.Dataset or list(xarray.Dataset)*) – tracks to append.

`get_track`(*track_name=None*)
Get track with provided name.

Returns the first matching track based on the assumption that no other track with the same name or sid exists in the set.

Parameters **track_name** (*str, optional*) – Name or sid (ibtracsID for IBTrACS) of track. If None (default), return all tracks.

Returns **result** – Usually, a single track is returned. If no track with the specified name is found, an empty list *[]* is returned. If called with *track_name=None*, the list of all tracks is returned.

Return type xarray.Dataset or list of xarray.Dataset

`subset`(*filterdict*)
Subset tracks based on track attributes.

Select all tracks matching exactly the given attribute values.

Parameters **filterdict** (*dict or OrderedDict*) – Keys are attribute names, values are the corresponding attribute values to match. In case of an ordered dict, the filters are applied in the given order.

Returns **tc_tracks** – A new instance of TCTracks containing only the matching tracks.

Return type *TCTracks*

`tracks_in_exp`(*exposure*, *buffer=1.0*)
Select only the tracks that are in the vicinity (buffer) of an exposure.

Each exposure point/geometry is extended to a disc of radius *buffer*. Each track is converted to a line and extended by a radius *buffer*.

Parameters

- **exposure** (*Exposure*) – Exposure used to select tracks.
- **buffer** (*float, optional*) – Size of buffer around exposure geometries (in the units of *exposure.crs*), see *geopandas.distance*. Default: 1.0

Returns **filtered_tracks** – TCTracks object with tracks from tc_tracks intersecting the exposure whitin a buffer distance.

Return type *TCTracks*

`read_ibtracs_netcdf`(**args*, ***kwargs*)

This function is deprecated, use TCTracks.from_ibtracs_netcdf instead.

`classmethod from_ibtracs_netcdf`(*provider=None*, *rescale_windspeeds=True*, *storm_id=None*, *year_range=None*, *basin=None*, *genesis_basin=None*, *interpolate_missing=True*, *estimate_missing=False*, *correct_pres=False*, *discard_single_points=True*, *file_name='IBTrACS.ALL.v04r00.nc'*)

Create new TCTracks object from IBTrACS databse.

When using data from IBTrACS, make sure to be familiar with the scope and limitations of IBTrACS, e.g. by reading the official documentation (https://www.ncdc.noaa.gov/ibtracs/pdf/IBTrACS_version4_Technical_Details.pdf). Reading the CLIMADA documentation can't replace a thorough understanding of the underlying data. This function only provides a (hopefully useful) interface for the data input, but cannot provide any guidance or make recommendations about if and how to use IBTrACS data for your particular project.

Resulting tracks are required to have both pressure and wind speed information at all time steps. Therefore, all track positions where one of wind speed or pressure are missing are discarded unless one of *interpolate_missing* or *estimate_missing* are active.

Some corrections are automatically applied, such as: *environmental_pressure* is enforced to be larger than *central_pressure*.

Note that the tracks returned by this function might contain irregular time steps since that is often the case for the original IBTrACS records. Apply the *equal_timestep* function afterwards to enforce regular time steps.

Parameters

- **provider** (*str or list of str, optional*) – Either specify an agency, such as "usa", "newdelhi", "bom", "cma", "tokyo", or the special values "official" and "official_3h":
  - "official" means using the (usually 6-hourly) officially reported values of the officially responsible agencies.
  - "official_3h" means to include (inofficial) 3-hourly data of the officially responsible agencies (whenever available).

  If you want to restrict to the officially reported values by the officially responsible agencies (*provider="official"*) without any modifications to the original official data, make sure to also set *estimate_missing=False* and *interpolate_missing=False*. Otherwise, gaps in the official reporting will be filled using interpolation and/or statistical estimation procedures (see below). If a list is given, the following logic is applied: For each storm, the variables that are not reported by the first agency for this storm are taken from the next agency in the list that did report this variable for this storm. For different storms, the same variable might be taken from different agencies. Default: ['official_3h', 'usa', 'tokyo', 'newdelhi', 'reunion', 'bom', 'nadi', 'wellington', 'cma', 'hko', 'ds824', 'td9636', 'td9635', 'neumann', 'mlc']

- **rescale_windspeeds** (*bool, optional*) – If True, all wind speeds are linearly rescaled to 1-minute sustained winds. Note however that the IBTrACS documentation (Section 5.2, https://www.ncdc.noaa.gov/ibtracs/pdf/IBTrACS_version4_Technical_Details.pdf) includes a warning about this kind of conversion: "While a multiplicative factor can describe the numerical differences, there are procedural and observational differences between agencies that can change through time, which confounds the simple multiplicative factor." Default: True
- **storm_id** (*str or list of str, optional*) – IBTrACS ID of the storm, e.g. 1988234N13299, [1988234N13299, 1989260N11316].
- **year_range** (*tuple (min_year, max_year), optional*) – Year range to filter track selection. Default: None.
- **basin** (*str, optional*) – If given, select storms that have at least one position in the specified basin. This allows analysis of a given basin, but also means that basin-specific track sets should not be combined across basins since some storms will be in more than one set. If you would like to select storms by their (unique) genesis basin instead, use the parameter *genesis_basin*. For possible values (basin abbreviations), see the parameter *genesis_basin*. If None, this filter is not applied. Default: None.
- **genesis_basin** (*str, optional*) – The basin where a TC is formed is not defined in IBTrACS. However, this filter option allows to restrict to storms whose first valid eye position is in the specified basin, which simulates the genesis location. Note that the resulting genesis basin of a particular track may depend on the selected *provider* and on *estimate_missing* because only the first *valid* eye position is considered. Possible values are 'NA' (North Atlantic), 'SA' (South Atlantic), 'EP' (Eastern North Pacific, which includes the Central Pacific region), 'WP' (Western North Pacific), 'SP' (South Pacific), 'SI' (South Indian), 'NI' (North Indian). If None, this filter is not applied. Default: None.
- **interpolate_missing** (*bool, optional*) – If True, interpolate temporal reporting gaps within a variable (such as pressure, wind speed, or radius) linearly if possible. Temporal interpolation is with respect to the time steps defined in IBTrACS for a particular storm. No new time steps are added that are not originally defined in IBTrACS. For each time step with a missing value, this procedure is only able to fill in that value if there are other time steps before and after this time step for which values have been reported. This procedure will be applied before the statistical estimations referred to by *estimate_missing*. It is applied to all variables (eye position, wind speed, environmental and central pressure, storm radius and radius of maximum winds). Default: True
- **estimate_missing** (*bool, optional*) – For each fixed time step, estimate missing pressure, wind speed and radius using other variables that are available at that time step. The relationships between the variables are purely statistical. In comparison to *interpolate_missing*, this procedure is able to estimate values for variables that haven't been reported by any agency at any time step, as long as other variables are available. A typical example are storms before 1950, for which there are often no reported values for pressure, but for wind speed. In this case, a rough statistical pressure-wind relationship is applied to estimate the missing pressure values from the available wind-speed values. Make sure to set *rescale_windspeeds=True* when using this option because the statistical relationships are calibrated using rescaled wind speeds. Default: False
- **correct_pres** (*bool, optional*) – For backwards compatibility, alias for *estimate_missing*. This is deprecated, use *estimate_missing* instead!
- **discard_single_points** (*bool, optional*) – Whether to discard tracks that consists of a single point. Recommended for full compatiblity with other functions such as

equal_timesteps. Default: True.

- **file_name** (*str, optional*) – Name of NetCDF file to be dowloaded or located at climada/data/system. Default: 'IBTrACS.ALL.v04r00.nc'

Returns **tracks** – TCTracks with data from IBTrACS

Return type *TCTracks*

`read_processed_ibtracs_csv`(**args*, ***kwargs*)
This function is deprecated, use TCTracks.from_processed_ibtracs_csv instead.

`classmethod from_processed_ibtracs_csv`(*file_names*)
Create TCTracks object from processed ibtracs CSV file(s).

Parameters **file_names** (*str or list of str*) – Absolute file name(s) or folder name containing the files to read.

Returns **tracks** – TCTracks with data from the processed ibtracs CSV file.

Return type *TCTracks*

`read_simulations_emanuel`(**args*, ***kwargs*)
This function is deprecated, use TCTracks.from_simulations_emanuel instead.

`classmethod from_simulations_emanuel`(*file_names*, *hemisphere=None*)
Create new TCTracks object from Kerry Emanuel's tracks.

Parameters

- **file_names** (*str or list of str*) – Absolute file name(s) or folder name containing the files to read.
- **hemisphere** (*str or None, optional*) – For global data sets, restrict to northern ('N') or southern ('S') hemisphere. Default: None (no restriction)

Returns **tracks** – TCTracks with data from Kerry Emanuel's simulations.

Return type *TCTracks*

`read_one_gettelman`(*nc_data*, *i_track*)
This function is deprecated, use TCTracks.from_gettelman instead.

`classmethod from_gettelman`(*path*)
Create new TCTracks object from Andrew Gettelman's tracks.

Parameters **path** (*str or Path*) – Path to one of Andrew Gettelman's NetCDF files.

Returns **tracks** – TCTracks with data from Andrew Gettelman's simulations.

Return type *TCTracks*

`read_simulations_chaz`(**args*, ***kwargs*)
This function is deprecated, use TCTracks.from_simulations_chaz instead.

`classmethod from_simulations_chaz`(*file_names*, *year_range=None*, *ensemble_nums=None*)
Create new TCTracks object from CHAZ simulations

Lee, C.-Y., Tippett, M.K., Sobel, A.H., Camargo, S.J. (2018): An Environmentally Forced Tropical Cyclone Hazard Model. J Adv Model Earth Sy 10(1): 223–241.

Parameters

- **file_names** (*str or list of str*) – Absolute file name(s) or folder name containing the files to read.

- **year_range** (*tuple (min_year, max_year), optional*) – Filter by year, if given.
- **ensemble_nums** (*list, optional*) – Filter by ensembleNum, if given.

Returns **tracks** – TCTracks with data from the CHAZ simulations.

Return type *TCTracks*

read_simulations_storm(**args*, ***kwargs*)
This function is deprecated, use TCTracks.from_simulations_storm instead.

classmethod from_simulations_storm(*path*, *years=None*)
Create new TCTracks object from STORM simulations

Bloemendaal et al. (2020): Generation of a global synthetic tropical cyclone hazard dataset using STORM. Scientific Data 7(1): 40.

Track data available for download from

https://doi.org/10.4121/uuid:82c1dc0d-5485-43d8-901a-ce7f26cda35d

Wind speeds are converted to 1-minute sustained winds through division by 0.88 (this value is taken from Bloemendaal et al. (2020), cited above).

Parameters

- **path** (*str*) – Full path to a txt-file as contained in the *data.zip* archive from the official source linked above.
- **years** (*list of int, optional*) – If given, only read the specified "years" from the txt-File. Note that a "year" refers to one ensemble of tracks in the data set that represents one sample year.

Returns **tracks** – TCTracks with data from the STORM simulations.

Return type *TCTracks*

equal_timestep(*time_step_h=1*, *land_params=False*, *pool=None*)
Generate interpolated track values to time steps of time_step_h.

Parameters

- **time_step_h** (*float or int, optional*) – Temporal resolution in hours (positive, may be non-integer-valued). Default: 1.
- **land_params** (*bool, optional*) – If True, recompute *on_land* and *dist_since_lf* at each node. Default: False.
- **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. If not given, the pool attribute of *self* will be used. Default: None

calc_random_walk(***kwargs*)
Deprecated. Use *TCTracks.calc_perturbed_trajectories* instead.

calc_perturbed_trajectories(***kwargs*)
See function in *climada.hazard.tc_tracks_synth*.

property size
Get longitude from coord array.

get_bounds(*deg_buffer=0.1*)
Get bounds as (lon_min, lat_min, lon_max, lat_max) tuple.

Parameters **deg_buffer** (*float*) – A buffer to add around the bounding box

Returns **bounds**

Return type tuple (lon_min, lat_min, lon_max, lat_max)

`property bounds`
Exact bounds of trackset as tuple, no buffer.

`get_extent`(*deg_buffer=0.1*)
Get extent as (lon_min, lon_max, lat_min, lat_max) tuple.

Parameters **deg_buffer** (*float*) – A buffer to add around the bounding box

Returns **extent**

Return type tuple (lon_min, lon_max, lat_min, lat_max)

`property extent`
Exact extent of trackset as tuple, no buffer.

`generate_centroids`(*res_deg*, *buffer_deg*)
Generate gridded centroids within padded bounds of tracks

Parameters

- **res_deg** (*float*) – Resolution in degrees.
- **buffer_deg** (*float*) – Buffer around tracks in degrees.

Returns **centroids** – Centroids instance.

Return type *Centroids*

`plot`(*axis=None*, *figsize=(9, 13)*, *legend=True*, *adapt_fontsize=True*, ***kwargs*)
Track over earth. Historical events are blue, probabilistic black.

Parameters

- **axis** (*matplotlib.axes._subplots.AxesSubplot, optional*) – axis to use
- **figsize** (*(float, float), optional*) – figure size for plt.subplots The default is (9, 13)
- **legend** (*bool, optional*) – whether to display a legend of Tropical Cyclone categories. Default: True.
- **kwargs** (*optional*) – arguments for LineCollection matplotlib, e.g. alpha=0.5
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Returns **axis**

Return type matplotlib.axes._subplots.AxesSubplot

`write_netcdf`(*folder_name*)
Write a netcdf file per track with track.sid name in given folder.

Parameters **folder_name** (*str*) – Folder name where to write files.

`read_netcdf`(**args*, ***kwargs*)
This function is deprecated, use TCTracks.from_netcdf instead.

`classmethod from_netcdf`(*folder_name*)
Create new TCTracks object from NetCDF files contained in a given folder

Parameters **folder_name** (*str*) – Folder name from where to read files.

Returns **tracks** – TCTracks with data from the given directory of NetCDF files.

Return type *TCTracks*

write_hdf5(*file_name*, *complevel=5*)
Write TC tracks in NetCDF4-compliant HDF5 format.

Parameters

- **file_name** (*str or Path*) – Path to a new HDF5 file. If it exists already, the file is overwritten.
- **complevel** (*int*) – Specifies a compression level (0-9) for the zlib compression of the data. A value of 0 or None disables compression. Default: 5

classmethod from_hdf5(*file_name*)
Create new TCTracks object from a NetCDF4-compliant HDF5 file

Parameters **file_name** (*str or Path*) – Path to a file that has been generated with *TCTracks.write_hdf*.

Returns **tracks** – TCTracks with data from the given HDF5 file.

Return type *TCTracks*

to_geodataframe(*as_points=False*, *split_lines_antimeridian=True*)
Transform this TCTracks instance into a GeoDataFrame.

Parameters

- **as_points** (*bool, optional*) – If False (default), one feature (row) per track with a LineString or MultiLineString as geometry (or Point geometry for tracks of length one) and all track attributes (sid, name, orig_event_flag, etc) as dataframe columns. If True, one feature (row) per track time step, with variable values per time step (radius_max_wind, max_sustained_wind, etc) as columns in addition to attributes.
- **split_lines_antimeridian** (*bool, optional*) – If True, tracks that cross the antimeridian are split into multiple Lines as a MultiLineString, with each Line on either side of the meridian. This ensures all Lines are within (-180, +180) degrees longitude. Note that lines might be split at more locations than strictly necessary, due to the underlying splitting algorithm (https://github.com/Toblerity/Shapely/issues/572).

Returns **gdf**

Return type GeoDataFrame

climada.hazard.tc_tracks.**set_category**(*max_sus_wind*, *wind_unit='kn'*, *saffir_scale=None*)
Add storm category according to Saffir-Simpson hurricane scale.

Parameters

- **max_sus_wind** (*np.array*) – Maximum sustained wind speed records for a single track.
- **wind_unit** (*str, optional*) – Units of wind speed. Default: 'kn'.
- **saffir_scale** (*list, optional*) – Saffir-Simpson scale in same units as wind (default scale valid for knots).

Returns

category –

Intensity of given track according to the Saffir-Simpson hurricane scale:

- -1 : tropical depression
- 0 : tropical storm
- 1 : Hurricane category 1

- 2 : Hurricane category 2
- 3 : Hurricane category 3
- 4 : Hurricane category 4
- 5 : Hurricane category 5

Return type int

climada.hazard.tc_tracks_synth module

climada.hazard.tc_tracks_synth.**LANDFALL_DECAY_V = {-1: 0.00012859077693295416, 0: 0.0017226346292718126, 1: 0.002309772914350468, 2: 0.0025968221565522698, 3: 0.002626252944053856, 4: 0.002550639312763181, 5: 0.003788695795963695}**

Global landfall decay parameters for wind speed by TC category. Keys are TC categories with -1='TD', 0='TS', 1='Cat 1', ..., 5='Cat 5'. It is v_rel as derived from: tracks = TCTracks.from_ibtracs_netcdf(year_range=(1980,2019), estimate_missing=True) extent = tracks.get_extent() land_geom = climada.util.coordinates.get_land_geometry(

extent=extent, resolution=10

) v_rel, p_rel = _calc_land_decay(tracks.data, land_geom, pool=tracks.pool)

climada.hazard.tc_tracks_synth.**LANDFALL_DECAY_P = {-1: (1.0088807492745373, 0.002117478217863062), 0: (1.0192813768091684, 0.003068578025845065), 1: (1.0362982218631644, 0.003620816186262243), 2: (1.0468630800617038, 0.004067381088015585), 3: (1.0639055205005432, 0.003708174876364079), 4: (1.0828373148889825, 0.003997492773076179), 5: (1.1088615145002092, 0.005224331234796362)}**

Global landfall decay parameters for pressure by TC category. Keys are TC categories with -1='TD', 0='TS', 1='Cat 1', ..., 5='Cat 5'. It is p_rel as derived from: tracks = TCTracks.from_ibtracs_netcdf(year_range=(1980,2019), estimate_missing=True) extent = tracks.get_extent() land_geom = climada.util.coordinates.get_land_geometry(

extent=extent, resolution=10

) v_rel, p_rel = _calc_land_decay(tracks.data, land_geom, pool=tracks.pool)

climada.hazard.tc_tracks_synth.**calc_perturbed_trajectories**(*tracks*, *nb_synth_tracks=9*, *max_shift_ini=0.75*, *max_dspeed_rel=0.3*, *max_ddirection=0.008726646259971648*, *autocorr_dspeed=0.85*, *autocorr_ddirection=0.5*, *seed=54*, *decay=True*, *use_global_decay_params=True*, *pool=None*)

Generate synthetic tracks based on directed random walk. An ensemble of nb_synth_tracks synthetic tracks is computed for every track contained in self.

The methodology perturbs the tracks locations, and if decay is True it additionally includes decay of wind speed and central pressure drop after landfall. No other track parameter is perturbed. The track starting point location is perturbed by random uniform values of magnitude up to max_shift_ini in both longitude and latitude. Then, each segment between two consecutive points is perturbed in direction and distance (i.e., translational speed). These perturbations can be correlated in time, i.e., the perturbation in direction applied to segment i is correlated with the perturbation in direction applied to segment i-1 (and similarly for the perturbation in translational speed). Perturbations in track direction and temporal auto-correlations in perturbations are on an hourly basis, and the perturbations in translational speed is relative. Hence, the parameter values are relatively insensitive

to the temporal resolution of the tracks. Note however that all tracks should be at the same temporal resolution, which can be achieved using equal_timestep(). max_dspeed_rel and autocorr_dspeed control the spread along the track ('what distance does the track run for'), while max_ddirection and autocorr_ddirection control the spread perpendicular to the track movement ('how does the track diverge in direction'). max_dspeed_rel and max_ddirection control the amplitude of perturbations at each track timestep but perturbations may tend to compensate each other over time, leading to a similar location at the end of the track, while autocorr_dspeed and autocorr_ddirection control how these perturbations persist in time and hence the amplitude of the perturbations towards the end of the track.

Note that the default parameter values have been only roughly calibrated so that the frequency of tracks in each 5x5degree box remains approximately constant. This is not an in-depth calibration and should be treated as such. The object is mutated in-place.

Parameters

- **tracks** (*climada.hazard.TCTracks*) – Tracks data.
- **nb_synth_tracks** (*int, optional*) – Number of ensemble members per track. Default: 9.
- **max_shift_ini** (*float, optional*) – Amplitude of max random starting point shift in decimal degree (up to +/-max_shift_ini for longitude and latitude). Default: 0.75.
- **max_dspeed_rel** (*float, optional*) – Amplitude of translation speed perturbation in relative terms (e.g., 0.2 for +/-20%). Default: 0.3.
- **max_ddirection** (*float, optional*) – Amplitude of track direction (bearing angle) perturbation per hour, in radians. Default: pi/360.
- **autocorr_dspeed** (*float, optional*) – Temporal autocorrelation in translation speed perturbation at a lag of 1 hour. Default: 0.85.
- **autocorr_ddirection** (*float, optional*) – Temporal autocorrelation of translational direction perturbation at a lag of 1 hour. Default: 0.5.
- **seed** (*int, optional*) – Random number generator seed for replicability of random walk. Put negative value if you don't want to use it. Default: configuration file.
- **decay** (*bool, optional*) – Whether to apply landfall decay in probabilistic tracks. Default: True.
- **use_global_decay_params** (*bool, optional*) – Whether to use precomputed global parameter values for landfall decay obtained from IBTrACS (1980-2019). If False, parameters are fitted using historical tracks in input parameter 'tracks', in which case the landfall decay applied depends on the tracks passed as an input and may not be robust if few historical tracks make landfall in this object. Default: True.
- **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. If not given, the pool attribute of *tracks* will be used. Default: None

climada.hazard.trop_cyclone module

class climada.hazard.trop_cyclone.**TropCyclone**(*pool=None*)

Bases: *climada.hazard.base.Hazard*

Contains tropical cyclone events.

category

for every event, the TC category using the Saffir-Simpson scale:

-1 tropical depression 0 tropical storm 1 Hurrican category 1 2 Hurrican category 2 3 Hurrican category 3 4 Hurrican category 4 5 Hurrican category 5

Type np.array(int)

basin
basin where every event starts 'NA' North Atlantic 'EP' Eastern North Pacific 'WP' Western North Pacific 'NI' North Indian 'SI' South Indian 'SP' Southern Pacific 'SA' South Atlantic

Type list(str)

intensity_thres = 17.5
intensity threshold for storage in m/s

vars_opt = {'category'}
Name of the variables that aren't need to compute the impact.

__init__(*pool=None*)
Initialize values.

Parameters **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation when applicable. Default: None

set_from_tracks(**args*, ***kwargs*)
This function is deprecated, use TropCyclone.from_tracks instead.

classmethod from_tracks(*tracks*, *centroids=None*, *pool=None*, *description=''*, *model='H08'*, *ignore_distance_to_coast=False*, *store_windfields=False*, *metric='equirect'*, *intensity_thres=17.5*)
Create new TropCyclone instance that contains windfields from the specified tracks.

This function sets the *intensity* attribute to contain, for each centroid, the maximum wind speed (1-minute sustained winds at 10 meters above ground) experienced over the whole period of each TC event in m/s. The wind speed is set to 0 if it doesn't exceed the threshold *intensity_thres*.

The *category* attribute is set to the value of the *category*-attribute of each of the given track data sets.

The *basin* attribute is set to the genesis basin for each event, which is the first value of the *basin*-variable in each of the given track data sets.

Optionally, the time dependent, vectorial winds can be stored using the *store_windfields* function parameter (see below).

Parameters

- **tracks** (*TCTracks*) – Tracks of storm events.
- **centroids** (*Centroids, optional*) – Centroids where to model TC. Default: global centroids at 360 arc-seconds resolution.
- **pool** (*pathos.pool, optional*) – Pool that will be used for parallel computation of wind fields. Default: None
- **description** (*str, optional*) – Description of the event set. Default: "".
- **model** (*str, optional*) – Parametric wind field model to use: one of "H1980" (the prominent Holland 1980 model), "H08" (Holland 1980 with b-value from Holland 2008), or "H10" (Holland et al. 2010). Default: "H08".
- **ignore_distance_to_coast** (*boolean, optional*) – If True, centroids far from coast are not ignored. Default: False.
- **store_windfields** (*boolean, optional*) – If True, the Hazard object gets a list *windfields* of sparse matrices. For each track, the full velocity vectors at each centroid and track position are stored in a sparse matrix of shape (npositions, ncentroids * 2) that can be reshaped to a full ndarray of shape (npositions, ncentroids, 2). Default: False.

- **metric** (*str, optional*) – Specify an approximation method to use for earth distances: * "equirect": Distance according to sinusoidal projection. Fast, but inaccurate for large distances and high latitudes.
  - "geosphere": Exact spherical distance. Much more accurate at all distances, but slow.

  Default: "equirect".
- **intensity_thres** (*float, optional*) – Wind speeds (in m/s) below this threshold are stored as 0. Default: 17.5

Raises `ValueError` –

Return type *TropCyclone*

`apply_climate_scenario_knu`(*ref_year=2050*, *rcp_scenario=45*)

From current TC hazard instance, return new hazard set with future events for a given RCP scenario and year based on the parametrized values derived from Table 3 in Knutson et al 2015. https://doi.org/10.1175/JCLI-D-15-0129.1 . The scaling for different years and RCP scenarios is obtained by linear interpolation.

Note: The parametrized values are derived from the overall changes in statistical ensemble of tracks. Hence, this method should only be applied to sufficiently large tropical cyclone event sets that approximate the reference years 1981 - 2008 used in Knutson et. al.

The frequency and intensity changes are applied independently from one another. The mean intensity factors can thus slightly deviate from the Knutson value (deviation was found to be less than 1% for default IBTrACS event sets 1980-2020 for each basin).

Parameters

- **ref_year** (*int*) – year between 2000 ad 2100. Default: 2050
- **rcp_scenario** (*int*) – 26 for RCP 2.6, 45 for RCP 4.5, 60 for RCP 6.0 and 85 for RCP 8.5. The default is 45.

Returns haz_cc – Tropical cyclone with frequencies and intensity scaled according to the Knutson criterion for the given year and RCP. Returns a new instance of climada.hazard.TropCyclone, self is not modified.

Return type climada.hazard.TropCyclone

`set_climate_scenario_knu`(**args*, ***kwargs*)

This function is deprecated, use TropCyclone.apply_climate_scenario_knu instead.

`classmethod video_intensity`(*track_name*, *tracks*, *centroids*, *file_name=None*, *writer=<matplotlib.animation.PillowWriter object>*, *figsize=(9, 13)*, *adapt_fontsize=True*, ***kwargs*)

Generate video of TC wind fields node by node and returns its corresponding TropCyclone instances and track pieces.

Parameters

- **track_name** (*str*) – name of the track contained in tracks to record
- **tracks** (*climada.hazard.TCTracks*) – tropical cyclone tracks
- **centroids** (*climada.hazard.Centroids*) – centroids where wind fields are mapped
- **file_name** (*str, optional*) – file name to save video (including full path and file extension)

- **writer** (*matplotlib.animation.*, optional*) – video writer. Default is pillow with bitrate=500
- **figsize** (*tuple, optional*) – figure size for plt.subplots
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- **kwargs** (*optional*) – arguments for pcolormesh matplotlib function used in event plots

Returns tc_list, tc_coord

Return type list(*TropCyclone*), list(np.array)

Raises `ValueError` –

`frequency_from_tracks`(*tracks*)

Set hazard frequency from tracks data.

Parameters **tracks** (*list of xarray.Dataset*)

`classmethod from_single_track`(*track*, *centroids*, *coastal_idx*, *model='H08'*, *store_windfields=False*, *metric='equirect'*, *intensity_thres=17.5*)

Generate windfield hazard from a single track dataset

Parameters

- **track** (*xr.Dataset*) – Single tropical cyclone track.
- **centroids** (*Centroids*) – Centroids instance.
- **coastal_idx** (*np.array*) – Indices of centroids close to coast.
- **model** (*str, optional*) – Parametric wind field model, one of "H1980" (the prominent Holland 1980 model), "H08" (Holland 1980 with b-value from Holland 2008), or "H10" (Holland et al. 2010). Default: "H08".
- **store_windfields** (*boolean, optional*) – If True, store windfields. Default: False.
- **metric** (*str, optional*) – Specify an approximation method to use for earth distances: "equirect" (faster) or "geosphere" (more accurate). See *dist_approx* function in *climada.util.coordinates*. Default: "equirect".
- **intensity_thres** (*float, optional*) – Wind speeds (in m/s) below this threshold are stored as 0. Default: 17.5

Raises `ValueError, KeyError` –

Returns haz

Return type *TropCyclone*

7.1.4 climada.util package

climada.util.api_client module

`class` climada.util.api_client.`Download`(**args*, ***kwargs*)

Bases: `peewee.Model`

Database entry keeping track of downloaded files from the CLIMADA data API

`url = <CharField: Download.url>`

path = <CharField: Download.path>

startdownload = <DateTimeField: Download.startdownload>

enddownload = <DateTimeField: Download.enddownload>

exception Failed
Bases: Exception

The download failed for some reason.

DoesNotExist
alias of climada.util.api_client.DownloadDoesNotExist

id = <AutoField: Download.id>

class climada.util.api_client.**FileInfo**(*uuid: str*, *url: str*, *file_name: str*, *file_format: str*, *file_size: int*, *check_sum: str*)

Bases: object

file data from CLIMADA data API.

uuid: str

url: str

file_name: str

file_format: str

file_size: int

check_sum: str

__init__(*uuid: str*, *url: str*, *file_name: str*, *file_format: str*, *file_size: int*, *check_sum: str*) → None

class climada.util.api_client.**DataTypeInfo**(*data_type: str*, *data_type_group: str*, *status: str*, *description: str*, *properties: list*)

Bases: object

data type meta data from CLIMADA data API.

data_type: str

data_type_group: str

status: str

description: str

properties: list

__init__(*data_type: str*, *data_type_group: str*, *status: str*, *description: str*, *properties: list*) → None

class climada.util.api_client.**DataTypeShortInfo**(*data_type: str*, *data_type_group: str*)
Bases: object

data type name and group from CLIMADA data API.

data_type: str

data_type_group: str

__init__(*data_type: str*, *data_type_group: str*) → None

class climada.util.api_client.**DatasetInfo**(*uuid: str*, *data_type:* climada.util.api_client.DataTypeShortInfo, *name: str*, *version: str*, *status: str*, *properties: dict*, *files: list*, *doi: str*, *description: str*, *license: str*, *activation_date: str*, *expiration_date: str*)

Bases: `object`

dataset data from CLIMADA data API.

uuid: str

data_type: *climada.util.api_client.DataTypeShortInfo*

name: str

version: str

status: str

properties: dict

files: list

doi: str

description: str

license: str

activation_date: str

expiration_date: str

static from_json(*jsono*)

creates a DatasetInfo object from the json object returned by the CLIMADA data api server.

Parameters **jsono** (*dict*)

Return type *DatasetInfo*

__init__(*uuid: str*, *data_type:* climada.util.api_client.DataTypeShortInfo, *name: str*, *version: str*, *status: str*, *properties: dict*, *files: list*, *doi: str*, *description: str*, *license: str*, *activation_date: str*, *expiration_date: str*) → None

climada.util.api_client.**checksize**(*local_path*, *fileinfo*)

Checks sanity of downloaded file simply by comparing actual and registered size.

Parameters

- **local_path** (*Path*) – the downloaded file
- **filinfo** (*FileInfo*) – file information from CLIMADA data API

Raises *Download.Failed* – if the file is not what it's supposed to be

climada.util.api_client.**checkhash**(*local_path*, *fileinfo*)

Checks sanity of downloaded file by comparing actual and registered check sum.

Parameters

- **local_path** (*Path*) – the downloaded file
- **filinfo** (*FileInfo*) – file information from CLIMADA data API

Raises *Download.Failed* – if the file is not what it's supposed to be

class climada.util.api_client.**Client**

Bases: object

Python wrapper around REST calls to the CLIMADA data API server.

MAX_WAITING_PERIOD = 6

UNLIMITED = 100000

exception AmbiguousResult

Bases: Exception

Custom Exception for Non-Unique Query Result

exception NoResult

Bases: Exception

Custom Exception for No Query Result

__init__()

Constructor of Client.

Data API host and chunk_size (for download) are configurable values. Default values are 'climada.ethz.ch' and 8096 respectively.

list_dataset_infos(*data_type=None*, *name=None*, *version=None*, *properties=None*, *status='active'*)

Find all datasets matching the given parameters.

Parameters

- **data_type** (*str, optional*) – data_type of the dataset, e.g., 'litpop' or 'draught'
- **name** (*str, optional*) – the name of the dataset
- **version** (*str, optional*) – the version of the dataset
- **properties** (*dict, optional*) – search parameters for dataset properties, by default None any property has a string for key and can be a string or a list of strings for value
- **status** (*str, optional*) – valid values are 'preliminary', 'active', 'expired', 'test_dataset' and None by default 'active'

Return type list of DatasetInfo

get_dataset_info(*data_type=None*, *name=None*, *version=None*, *properties=None*, *status='active'*)

Find the one dataset that matches the given parameters.

Parameters

- **data_type** (*str, optional*) – data_type of the dataset, e.g., 'litpop' or 'draught'
- **name** (*str, optional*) – the name of the dataset
- **version** (*str, optional*) – the version of the dataset
- **properties** (*dict, optional*) – search parameters for dataset properties, by default None any property has a string for key and can be a string or a list of strings for value
- **status** (*str, optional*) – valid values are 'preliminary', 'active', 'expired', 'test_dataset', None by default 'active'

Return type *DatasetInfo*

Raises

- ***AmbiguousResult*** – when there is more than one dataset matching the search parameters

- **NoResult** – when there is no dataset matching the search parameters

get_dataset_info_by_uuid(*uuid*)

Returns the data from 'https://climada.ethz.ch/data-api/v1/dataset/{uuid}' as DatasetInfo object.

Parameters **uuid** (*str*) – the universal unique identifier of the dataset

Return type *DatasetInfo*

Raises **NoResult** – if the uuid is not valid

list_data_type_infos(*data_type_group=None*)

Returns all data types from the climada data API belonging to a given data type group.

Parameters **data_type_group** (*str, optional*) – name of the data type group, by default None

Return type list of DataTypeInfo

get_data_type_info(*data_type*)

Returns the metadata of the data type with the given name from the climada data API.

Parameters **data_type** (*str*) – data type name

Return type *DataTypeInfo*

Raises **NoResult** – if there is no such data type registered

download_dataset(*dataset*, *target_dir=PosixPath('/home/docs/climada/data')*, *organize_path=True*)

Download all files from a given dataset to a given directory.

Parameters

- **dataset** (*DatasetInfo*) – the dataset
- **target_dir** (*Path, optional*) – target directory for download, by default *climada.util.constants.SYSTEM_DIR*
- **organize_path** (*bool, optional*) – if set to True the files will end up in subdirectories of target_dir: [target_dir]/[data_type_group]/[data_type]/[name]/[version] by default True

Returns

- **download_dir** (*Path*) – the path to the directory containing the downloaded files, will be created if organize_path is True
- **downloaded_files** (*list of Path*) – the downloaded files themselves

Raises **Exception** – when one of the files cannot be downloaded

static purge_cache(*local_path*)

Removes entry from the sqlite database that keeps track of files downloaded by *cached_download*. This may be necessary in case a previous attempt has failed in an uncontroled way (power outage or the like).

Parameters

- **local_path** (*Path*) – target destination
- **fileinfo** (*FileInfo*) – file object as retrieved from the data api

get_hazard(*hazard_type*, *name=None*, *version=None*, *properties=None*, *status='active'*, *dump_dir=PosixPath('/home/docs/climada/data')*)

Queries the data api for hazard datasets of the given type, downloads associated hdf5 files and turns them into a climada.hazard.Hazard object.

Parameters

- **hazard_type** (*str*) – Type of climada hazard.
- **name** (*str, optional*) – the name of the dataset
- **version** (*str, optional*) – the version of the dataset
- **properties** (*dict, optional*) – search parameters for dataset properties, by default None any property has a string for key and can be a string or a list of strings for value
- **status** (*str, optional*) – valid values are 'preliminary', 'active', 'expired', 'test_dataset', None by default 'active'
- **dump_dir** (*str, optional*) – Directory where the files should be downoladed. Default: SYSTEM_DIR If the directory is the SYSTEM_DIR (as configured in climada.conf, i.g. ~/climada/data), the eventual target directory is organized into dump_dir > hazard_type > dataset name > version

Returns The combined hazard object

Return type climada.hazard.Hazard

`to_hazard`(*dataset*, *dump_dir=PosixPath('/home/docs/climada/data')*)

Downloads hdf5 files belonging to the given datasets reads them into Hazards and concatenates them into a single climada.Hazard object.

Parameters

- **dataset** (*DatasetInfo*) – Dataset to download and read into climada.Hazard object.
- **dump_dir** (*str, optional*) – Directory where the files should be downoladed. Default: SYSTEM_DIR (as configured in climada.conf, i.g. ~/climada/data). If the directory is the SYSTEM_DIR, the eventual target directory is organized into dump_dir > hazard_type > dataset name > version

Returns The combined hazard object

Return type climada.hazard.Hazard

`get_exposures`(*exposures_type*, *name=None*, *version=None*, *properties=None*, *status='active'*, *dump_dir=PosixPath('/home/docs/climada/data')*)

Queries the data api for exposures datasets of the given type, downloads associated hdf5 files and turns them into a climada.entity.exposures.Exposures object.

Parameters

- **exposures_type** (*str*) – Type of climada exposures.
- **name** (*str, optional*) – the name of the dataset
- **version** (*str, optional*) – the version of the dataset
- **properties** (*dict, optional*) – search parameters for dataset properties, by default None any property has a string for key and can be a string or a list of strings for value
- **status** (*str, optional*) – valid values are 'preliminary', 'active', 'expired', 'test_dataset', None by default 'active'
- **dump_dir** (*str, optional*) – Directory where the files should be downoladed. Default: SYSTEM_DIR If the directory is the SYSTEM_DIR, the eventual target directory is organized into dump_dir > hazard_type > dataset name > version

Returns The combined exposures object

Return type climada.entity.exposures.Exposures

`to_exposures`(*dataset*, *dump_dir=PosixPath('/home/docs/climada/data')*)

Downloads hdf5 files belonging to the given datasets reads them into Exposures and concatenates them into a single climada.Exposures object.

Parameters

- **dataset** (*DatasetInfo*) – Dataset to download and read into climada.Exposures objects.
- **dump_dir** (*str, optional*) – Directory where the files should be downoladed. Default: SYSTEM_DIR (as configured in climada.conf, i.g. ~/climada/data). If the directory is the SYSTEM_DIR, the eventual target directory is organized into dump_dir > exposures_type > dataset name > version

Returns The combined exposures object

Return type climada.entity.exposures.Exposures

`get_litpop_default`(*country=None*, *dump_dir=PosixPath('/home/docs/climada/data')*)

Get a LitPop instance on a 150arcsec grid with the default parameters: exponents = (1,1) and fin_mode = 'pc'.

Parameters

- **country** (*str or list, optional*) – List of country name or iso3 codes for which to create the LitPop object. If None is given, a global LitPop instance is created. Defaut is None
- **dump_dir** (*str*) – directory where the files should be downoladed. Default: SYSTEM_DIR

Returns default litpop Exposures object

Return type climada.entity.exposures.Exposures

`static get_property_values`(*dataset_infos*, *known_property_values=None*, *exclude_properties=None*)

Returns a dictionnary of possible values for properties of a data type, optionally given known property values.

Parameters

- **dataset_infos** (*list of DataSetInfo*) – as returned by list_dataset_infos
- **known_properties_value** (*dict, optional*) – dict {'property':'value1, 'property2':'value2'}, to provide only a subset of property values that can be combined with the given properties.
- **exclude_properties** (*list of str, optional*) – properties in this list will be excluded from the resulting dictionary, e.g., because they are strictly metadata and don't provide any information essential to the dataset. Default: 'creation_date', 'climada_version'

Returns of possibles property values

Return type dict

`static into_datasets_df`(*dataset_infos*)

Convenience function providing a DataFrame of datasets with properties.

Parameters **dataset_infos** (*list of DatasetInfo*) – as returned by list_dataset_infos

Returns of datasets with properties as found in query by arguments

Return type pandas.DataFrame

`static into_files_df`(*dataset_infos*)

Convenience function providing a DataFrame of files aligned with the input datasets.

Parameters **datasets** (*list of DatasetInfo*) – as returned by list_dataset_infos

Returns of the files' informations including dataset informations

Return type pandas.DataFrame

climada.util.checker module

climada.util.checker.**size**(*exp_len*, *var*, *var_name*)
Check if the length of a variable is the expected one.

Raises **ValueError** –

climada.util.checker.**shape**(*exp_row*, *exp_col*, *var*, *var_name*)
Check if the length of a variable is the expected one.

Raises **ValueError** –

climada.util.checker.**array_optional**(*exp_len*, *var*, *var_name*)
Check if array has right size. Warn if array empty. Call check_size.

Parameters

- **exp_len** (*str*) – expected array size
- **var** (*np.array*) – numpy array to check
- **var_name** (*str*) – name of the variable. Used in error/warning msg

Raises **ValueError** –

climada.util.checker.**array_default**(*exp_len*, *var*, *var_name*, *def_val*)
Check array has right size. Set default value if empty. Call check_size.

Parameters

- **exp_len** (*str*) – expected array size
- **var** (*np.array*) – numpy array to check
- **var_name** (*str*) – name of the variable. Used in error/warning msg
- **def_val** (*np.array*) – nump array used as default value

Raises **ValueError** –

Return type Filled array

climada.util.config module

climada.util.constants module

climada.util.constants.**SYSTEM_DIR = PosixPath('/home/docs/climada/data')**
Folder containing the data used internally

climada.util.constants.**DEMO_DIR = PosixPath('/home/docs/climada/demo/data')**
Folder containing the data used for tutorials

climada.util.constants.**ENT_DEMO_TODAY =**
PosixPath('/home/docs/climada/demo/data/demo_today.xlsx')
Entity demo present in xslx format.

climada.util.constants.**ENT_DEMO_FUTURE =**
PosixPath('/home/docs/climada/demo/data/demo_future_TEST.xlsx')
Entity demo future in xslx format.

climada.util.constants.**HAZ_DEMO_MAT** = PosixPath('/home/docs/climada/demo/data/atl_prob_nonames.mat')

hurricanes from 1851 to 2011 over Florida with 100 centroids.

Type Hazard demo from climada in MATLAB

climada.util.constants.**HAZ_DEMO_FL** = PosixPath('/home/docs/climada/demo/data/SC22000_VE__M1.grd.gz')

Raster file of flood over Venezuela. Model from GAR2015

climada.util.constants.**ENT_TEMPLATE_XLS** = PosixPath('/home/docs/climada/data/entity_template.xlsx')

Entity template in xls format.

climada.util.constants.**HAZ_TEMPLATE_XLS** = PosixPath('/home/docs/climada/data/hazard_template.xlsx')

Hazard template in xls format.

climada.util.constants.**ONE_LAT_KM** = 111.12

Mean one latitude (in degrees) to km

climada.util.constants.**EARTH_RADIUS_KM** = 6371

Earth radius in km

climada.util.constants.**GLB_CENTROIDS_MAT** = PosixPath('/home/docs/climada/data/GLB_NatID_grid_0360as_adv_2.mat')

Global centroids

climada.util.constants.**GLB_CENTROIDS_NC** = PosixPath('/home/docs/climada/data/NatID_grid_0150as.nc')

For backwards compatibility, it remains available under its old name.

climada.util.constants.**ISIMIP_GPWV3_NATID_150AS** = PosixPath('/home/docs/climada/data/NatID_grid_0150as.nc')

Compressed version of National Identifier Grid in 150 arc-seconds from ISIMIP project, based on GPWv3. Location in ISIMIP repository:

ISIMIP2a/InputData/landuse_humaninfluences/population/ID_GRID/Nat_id_grid_ISIMIP.nc

More references:

- https://www.isimip.org/gettingstarted/input-data-bias-correction/details/13/
- https://sedac.ciesin.columbia.edu/data/set/gpw-v3-national-identifier-grid

climada.util.constants.**NATEARTH_CENTROIDS** = {150: PosixPath('/home/docs/climada/data/NatEarth_Centroids_150as.hdf5'), 360: PosixPath('/home/docs/climada/data/NatEarth_Centroids_360as.hdf5')}

Global centroids at XXX arc-seconds resolution, including region ids from Natural Earth. The 360 AS file includes distance to coast from NASA.

climada.util.constants.**RIVER_FLOOD_REGIONS_CSV** = PosixPath('/home/docs/climada/data/NatRegIDs.csv')

Look-up table for river flood module

climada.util.constants.**TC_ANDREW_FL** = PosixPath('/home/docs/climada/demo/data/ibtracs_global_intp-None_1992230N11325.csv')

Tropical cyclone Andrew in Florida

climada.util.constants.**HAZ_DEMO_H5** = PosixPath('/home/docs/climada/demo/data/tc_fl_1990_2004.h5')

IBTrACS from 1990 to 2004 over Florida with 2500 centroids.

Type Hazard demo in hdf5 format

climada.util.constants.**EXP_DEMO_H5 = PosixPath('/home/docs/climada/demo/data/exp_demo_today.h5')**
Exposures over Florida

climada.util.constants.**WS_DEMO_NC = [PosixPath('/home/docs/climada/demo/data/fp_lothar_crop-test.nc'), PosixPath('/home/docs/climada/demo/data/fp_xynthia_crop-test.nc')]**
Winter storm in Europe files. These test files have been generated using the netCDF kitchen sink:

```
>>> ncks -d latitude,50.5,54.0 -d longitude,3.0,7.5 ./file_in.nc ./file_out.nc
```

climada.util.constants.**TEST_UNC_OUTPUT_IMPACT = 'test_unc_output_impact'**
Demo uncertainty impact output

climada.util.constants.**TEST_UNC_OUTPUT_COSTBEN = 'test_unc_output_costben'**
Demo uncertainty costben output

climada.util.coordinates module

climada.util.coordinates.**NE_EPSG = 4326**
Natural Earth CRS EPSG

climada.util.coordinates.**NE_CRS = 'epsg:4326'**
Natural Earth CRS

climada.util.coordinates.**TMP_ELEVATION_FILE = PosixPath('/home/docs/climada/data/tmp_elevation.tif')**
Path of elevation file written in set_elevation

climada.util.coordinates.**DEM_NODATA = -9999**
Value to use for no data values in DEM, i.e see points

climada.util.coordinates.**MAX_DEM_TILES_DOWN = 300**
Maximum DEM tiles to dowload

climada.util.coordinates.**NEAREST_NEIGHBOR_THRESHOLD = 100**
Distance threshold in km for coordinate assignment. Nearest neighbors with greater distances are not considered.

climada.util.coordinates.**latlon_to_geosph_vector**(*lat*, *lon*, *rad=False*, *basis=False*)
Convert lat/lon coodinates to radial vectors (on geosphere)

Parameters

- **lat, lon** (*ndarrays of floats, same shape*) – Latitudes and longitudes of points.
- **rad** (*bool, optional*) – If True, latitude and longitude are not given in degrees but in radians.
- **basis** (*bool, optional*) – If True, also return an orthonormal basis of the tangent space at the given points in lat-lon coordinate system. Default: False.

Returns

- **vn** (*ndarray of floats, shape (..., 3)*) – Same shape as lat/lon input with additional axis for components.
- **vbasis** (*ndarray of floats, shape (..., 2, 3)*) – Only present, if *basis* is True. Same shape as lat/lon input with additional axes for components of the two basis vectors.

climada.util.coordinates.**lon_normalize**(*lon*, *center=0.0*)

Normalizes degrees such that always -180 < lon - center <= 180

The input data is modified in place!

Parameters

- **lon** (*np.array*) – Longitudinal coordinates
- **center** (*float, optional*) – Central longitude value to use instead of 0. If None, the central longitude is determined automatically.

Returns **lon** – Normalized longitudinal coordinates. Since the input *lon* is modified in place (!), the returned array is the same Python object (instead of a copy).

Return type np.array

climada.util.coordinates.**lon_bounds**(*lon*, *buffer=0.0*)

Bounds of a set of degree values, respecting the periodicity in longitude

The longitudinal upper bound may be 180 or larger to make sure that the upper bound is always larger than the lower bound. The lower longitudinal bound will never lie below -180 and it will only assume the value -180 if the specified buffering enforces it.

Note that, as a consequence of this, the returned bounds do not satisfy the inequality *lon_min* <= *lon* <= *lon_max* in general!

Usually, an application of this function is followed by a renormalization of longitudinal values around the longitudinal middle value:

```
>>> bounds = lon_bounds(lon)
>>> lon_mid = 0.5 * (bounds[0] + bounds[2])
>>> lon = lon_normalize(lon, center=lon_mid)
>>> np.all((bounds[0] <= lon) & (lon <= bounds[2]))
```

Example

```
>>> lon_bounds(np.array([-179, 175, 178]))
(175, 181)
>>> lon_bounds(np.array([-179, 175, 178]), buffer=1)
(174, 182)
```

Parameters

- **lon** (*np.array*) – Longitudinal coordinates
- **buffer** (*float, optional*) – Buffer to add to both sides of the bounding box. Default: 0.0.

Returns **bounds** – Bounding box of the given points.

Return type tuple (lon_min, lon_max)

climada.util.coordinates.**latlon_bounds**(*lat*, *lon*, *buffer=0.0*)

Bounds of a set of degree values, respecting the periodicity in longitude

See *lon_bounds* for more information about the handling of longitudinal values crossing the antimeridian.

Example

```
>>> latlon_bounds(np.array([0, -2, 5]), np.array([-179, 175, 178]))
(175, -2, 181, 5)
>>> latlon_bounds(np.array([0, -2, 5]), np.array([-179, 175, 178]), buffer=1)
(174, -3, 182, 6)
```

Parameters

- **lat** (*np.array*) – Latitudinal coordinates
- **lon** (*np.array*) – Longitudinal coordinates
- **buffer** (*float, optional*) – Buffer to add to all sides of the bounding box. Default: 0.0.

Returns **bounds** – Bounding box of the given points.

Return type tuple (lon_min, lat_min, lon_max, lat_max)

climada.util.coordinates.**dist_approx**(*lat1*, *lon1*, *lat2*, *lon2*, *log=False*, *normalize=True*, *method='equirect'*, *units='km'*)

Compute approximation of geodistance in specified units

Parameters

- **lat1, lon1** (*ndarrays of floats, shape (nbatch, nx)*) – Latitudes and longitudes of first points.
- **lat2, lon2** (*ndarrays of floats, shape (nbatch, ny)*) – Latitudes and longitudes of second points.
- **log** (*bool, optional*) – If True, return the tangential vectors at the first points pointing to the second points (Riemannian logarithm). Default: False.
- **normalize** (*bool, optional*) – If False, assume that lon values are already between -180 and 180. Default: True
- **method** (*str, optional*) –

  Specify an approximation method to use:

  – "equirect": Distance according to sinusoidal projection. Fast, but inaccurate for large distances and high latitudes.

  – "geosphere": Exact spherical distance. Much more accurate at all distances, but slow.

  Note that ellipsoidal distances would be even more accurate, but are currently not implemented. Default: "equirect".
- **units** (*str, optional*) –

  Specify a unit for the distance. One of:

  – "km": distance in km.

  – "degree": angular distance in decimal degrees.

  – "radian": angular distance in radians.

  Default: "km".

Returns

- **dists** (*ndarray of floats, shape (nbatch, nx, ny)*) – Approximate distances in specified units.

- **vtan** (*ndarray of floats, shape (nbatch, nx, ny, 2)*) – If *log* is True, tangential vectors at first points in local lat-lon coordinate system.

climada.util.coordinates.**get_gridcellarea**(*lat*, *resolution=0.5*, *unit='km2'*)

The area covered by a grid cell is calculated depending on the latitude 1 degree = ONE_LAT_KM (111.12km at the equator) longitudal distance in km = ONE_LAT_KM*resolution*cos(lat) latitudal distance in km = ONE_LAT_KM*resolution area = longitudal distance * latitudal distance

Parameters

- **lat** (*np.array*) – Latitude of the respective grid cell
- **resolution** (*int, optional*) – raster resolution in degree (default: 0.5 degree)
- **unit** (*string, optional*) – unit of the output area (default: km2, alternative: m2)

climada.util.coordinates.**grid_is_regular**(*coord*)

Return True if grid is regular. If True, returns height and width.

Parameters **coord** (*np.array*) – Each row is a lat-lon-pair.

Returns

- **regular** (*bool*) – Whether the grid is regular. Only in this case, the following width and height are reliable.
- **height** (*int*) – Height of the supposed grid.
- **width** (*int*) – Width of the supposed grid.

climada.util.coordinates.**get_coastlines**(*bounds=None*, *resolution=110*)

Get Polygones of coast intersecting given bounds

Parameters

- **bounds** (*tuple*) – min_lon, min_lat, max_lon, max_lat in EPSG:4326
- **resolution** (*float, optional*) – 10, 50 or 110. Resolution in m. Default: 110m, i.e. 1:110.000.000

Returns **coastlines** – Polygons of coast intersecting given bounds.

Return type GeoDataFrame

climada.util.coordinates.**convert_wgs_to_utm**(*lon*, *lat*)

Get EPSG code of UTM projection for input point in EPSG 4326

Parameters

- **lon** (*float*) – longitude point in EPSG 4326
- **lat** (*float*) – latitude of point (lat, lon) in EPSG 4326

Returns **epsg_code** – EPSG code of UTM projection.

Return type int

climada.util.coordinates.**utm_zones**(*wgs_bounds*)

Get EPSG code and bounds of UTM zones covering specified region

Parameters **wgs_bounds** (*tuple*) – lon_min, lat_min, lon_max, lat_max

Returns **zones** – EPSG code and bounding box in WGS coordinates.

Return type list of pairs (zone_epsg, zone_wgs_bounds)

climada.util.coordinates.**dist_to_coast**(*coord_lat*, *lon=None*, *signed=False*)
Compute (signed) distance to coast from input points in meters.

Parameters

- **coord_lat** (*GeoDataFrame or np.array or float*) –

  One of the following:

  - GeoDataFrame with geometry column in epsg:4326
  - np.array with two columns, first for latitude of each point and second with longitude in epsg:4326
  - np.array with one dimension containing latitudes in epsg:4326
  - float with a latitude value in epsg:4326

- **lon** (*np.array or float, optional*) –

  One of the following:

  - np.array with one dimension containing longitudes in epsg:4326
  - float with a longitude value in epsg:4326

- **signed** (*bool*) – If True, distance is signed with positive values off shore and negative values on land. Default: False

Returns **dist** – (Signed) distance to coast in meters.

Return type np.array

climada.util.coordinates.**dist_to_coast_nasa**(*lat*, *lon*, *highres=False*, *signed=False*)
Read interpolated (signed) distance to coast (in m) from NASA data

Note: The NASA raster file is 300 MB and will be downloaded on first run!

Parameters

- **lat** (*np.array*) – latitudes in epsg:4326
- **lon** (*np.array*) – longitudes in epsg:4326
- **highres** (*bool, optional*) – Use full resolution of NASA data (much slower). Default: False.
- **signed** (*bool*) – If True, distance is signed with positive values off shore and negative values on land. Default: False

Returns **dist** – (Signed) distance to coast in meters.

Return type np.array

climada.util.coordinates.**get_land_geometry**(*country_names=None*, *extent=None*, *resolution=10*)
Get union of the specified (or all) countries or the points inside the extent.

Parameters

- **country_names** (*list, optional*) – list with ISO3 names of countries, e.g ['ZWE', 'GBR', 'VNM', 'UZB']
- **extent** (*tuple, optional*) – (min_lon, max_lon, min_lat, max_lat)
- **resolution** (*float, optional*) – 10, 50 or 110. Resolution in m. Default: 10m, i.e. 1:10.000.000

Returns **geom** – Polygonal shape of union.

Return type shapely.geometry.multipolygon.MultiPolygon

climada.util.coordinates.**coord_on_land**(*lat*, *lon*, *land_geom=None*)
Check if points are on land.

Parameters

- **lat** (*np.array*) – latitude of points in epsg:4326
- **lon** (*np.array*) – longitude of points in epsg:4326
- **land_geom** (*shapely.geometry.multipolygon.MultiPolygon, optional*) – If given, use these as profiles of land. Otherwise, the global landmass is used.

Returns **on_land** – Entries are True if corresponding coordinate is on land and False otherwise.

Return type np.array(bool)

climada.util.coordinates.**nat_earth_resolution**(*resolution*)
Check if resolution is available in Natural Earth. Build string.

Parameters **resolution** (*int*) – resolution in millions, 110 == 1:110.000.000.

Returns **res_name** – Natural Earth name of resolution (e.g. '110m')

Return type str

Raises **ValueError** –

climada.util.coordinates.**get_country_geometries**(*country_names=None*, *extent=None*, *resolution=10*)
Natural Earth country boundaries within given extent

If no arguments are given, simply returns the whole natural earth dataset.

Take heed: we assume WGS84 as the CRS unless the Natural Earth download utility from cartopy starts including the projection information. (They are saving a whopping 147 bytes by omitting it.) Same goes for UTF.

Parameters

- **country_names** (*list, optional*) – list with ISO 3166 alpha-3 codes of countries, e.g ['ZWE', 'GBR', 'VNM', 'UZB']
- **extent** (*tuple (min_lon, max_lon, min_lat, max_lat), optional*) – Extent, assumed to be in the same CRS as the natural earth data.
- **resolution** (*float, optional*) – 10, 50 or 110. Resolution in m. Default: 10m

Returns **geom** – Natural Earth multipolygons of the specified countries, resp. the countries that lie within the specified extent.

Return type GeoDataFrame

climada.util.coordinates.**get_region_gridpoints**(*countries=None*, *regions=None*, *resolution=150*, *iso=True*, *rect=False*, *basemap='natearth'*)
Get coordinates of gridpoints in specified countries or regions

Parameters

- **countries** (*list, optional*) – ISO 3166-1 alpha-3 codes of countries, or internal numeric NatID if *iso* is set to False.
- **regions** (*list, optional*) – Region IDs.
- **resolution** (*float, optional*) – Resolution in arc-seconds, either 150 (default) or 360.
- **iso** (*bool, optional*) – If True, assume that countries are given by their ISO 3166-1 alpha-3 codes (instead of the internal NatID). Default: True.

- **rect** (*bool, optional*) – If True, a rectangular box around the specified countries/regions is selected. Default: False.
- **basemap** (*str, optional*) – Choose between different data sources. Currently available: "isimip" and "natearth". Default: "natearth".

Returns

- **lat** (*np.array*) – Latitude of points in epsg:4326.
- **lon** (*np.array*) – Longitude of points in epsg:4326.

climada.util.coordinates.**assign_grid_points**(*x*, *y*, *grid_width*, *grid_height*, *grid_transform*)

To each coordinate in *x* and *y*, assign the closest centroid in the given raster grid

Make sure that your grid specification is relative to the same coordinate reference system as the *x* and *y* coordinates. In case of lon/lat coordinates, make sure that the longitudinal values are within the same longitudinal range (such as [-180, 180]).

If your grid is given by bounds instead of a transform, the functions *rasterio.transform.from_bounds* and *pts_to_raster_meta* might be helpful.

Parameters

- **x, y** (*np.array*) – x- and y-coordinates of points to assign coordinates to.
- **grid_width** (*int*) – Width (number of columns) of the grid.
- **grid_height** (*int*) – Height (number of rows) of the grid.
- **grid_transform** (*affine.Affine*) – Affine transformation defining the grid raster.

Returns **assigned_idx** – Index into the flattened *grid*. Note that the value *-1* is used to indicate that no matching coordinate has been found, even though *-1* is a valid index in NumPy!

Return type np.array of size equal to the size of x and y

climada.util.coordinates.**assign_coordinates**(*coords*, *coords_to_assign*, *distance='euclidean'*, *threshold=100*, ***kwargs*)

To each coordinate in *coords*, assign a matching coordinate in *coords_to_assign*

If there is no exact match for some entry, an attempt is made to assign the geographically nearest neighbor. If the distance to the nearest neighbor exceeds *threshold*, the index *-1* is assigned.

Currently, the nearest neighbor matching works with lat/lon coordinates only. However, you can disable nearest neighbor matching by setting *threshold* to 0, in which case only exactly matching coordinates are assigned to each other.

Make sure that all coordinates are according to the same coordinate reference system. In case of lat/lon coordinates, the "haversine" distance is able to correctly compute the distance across the antimeridian. However, when exact matches are enforced with *threshold=0*, lat/lon coordinates need to be given in the same longitudinal range (such as (-180, 180)).

Parameters

- **coords** (*np.array with two columns*) – Each row is a geographical coordinate pair. The result's size will match this array's number of rows.
- **coords_to_assign** (*np.array with two columns*) – Each row is a geographical coordinate pair. The result will be an index into the rows of this array. Make sure that these coordinates use the same coordinate reference system as *coords*.
- **distance** (*str, optional*) – Distance to use for non-exact matching. Possible values are "euclidean", "haversine" and "approx". Default: "euclidean"

- **threshold** (*float, optional*) – If the distance to the nearest neighbor exceeds *threshold*, the index *-1* is assigned. Set *threshold* to 0 to disable nearest neighbor matching. Default: 100 (km)
- **kwargs** (*dict, optional*) – Keyword arguments to be passed on to nearest-neighbor finding functions in case of non-exact matching with the specified *distance*.

Returns **assigned_idx** – Index into *coords_to_assign*. Note that the value *-1* is used to indicate that no matching coordinate has been found, even though *-1* is a valid index in NumPy!

Return type np.array of size equal to the number of rows in *coords*

Notes

By default, the 'euclidean' distance metric is used to find the nearest neighbors in case of non-exact matching. This method is fast for (quasi-)gridded data, but introduces innacuracy since distances in lat/lon coordinates are not equal to distances in meters on the Earth surface, in particular for higher latitude and distances larger than 100km. If more accuracy is needed, please use the 'haversine' distance metric. This however is slower for (quasi-)gridded data.

climada.util.coordinates.**region2isos**(*regions*)

Convert region names to ISO 3166 alpha-3 codes of countries

Parameters **regions** (*str or list of str*) – Region name(s).

Returns **isos** – Sorted list of iso codes of all countries in specified region(s).

Return type list of str

climada.util.coordinates.**country_to_iso**(*countries*, *representation='alpha3'*, *fillvalue=None*)

Determine ISO 3166 representation of countries

Example

```
>>> country_to_iso(840)
'USA'
>>> country_to_iso("United States", representation="alpha2")
'US'
>>> country_to_iso(["United States of America", "SU"], "numeric")
[840, 810]
```

Some geopolitical areas that are not covered by ISO 3166 are added in the "user-assigned" range of ISO 3166-compliant values:

```
>>> country_to_iso(["XK", "Dhekelia"], "numeric")  # XK for Kosovo
[983, 907]
```

Parameters

- **countries** (*one of str, int, list of str, list of int*) – Country identifiers: name, official name, alpha-2, alpha-3 or numeric ISO codes. Numeric representations may be specified as str or int.
- **representation** (*str (one of "alpha3", "alpha2", "numeric", "name"), optional*) – All countries are converted to this representation according to ISO 3166. Default: "alpha3".
- **fillvalue** (*str or int or None, optional*) – The value to assign if a country is not recognized by the given identifier. By default, a LookupError is raised. Default: None

Returns **iso_list** – ISO 3166 representation of countries. Will only return a list if the input is a list. Numeric representations are returned as integers.

Return type one of str, int, list of str, list of int

climada.util.coordinates.**country_iso_alpha2numeric**(*iso_alpha*)
Deprecated: Use *country_to_iso* with *representation="numeric"* instead

climada.util.coordinates.**country_natid2iso**(*natids*, *representation='alpha3'*)
Convert internal NatIDs to ISO 3166-1 alpha-3 codes

Parameters

- **natids** (*int or list of int*) – NatIDs of countries (or single ID) as used in ISIMIP's version of the GPWv3 national identifier grid.
- **representation** (*str, one of "alpha3", "alpha2" or "numeric"*) – All countries are converted to this representation according to ISO 3166. Default: "alpha3".

Returns **iso_list** – ISO 3166 representation of countries. Will only return a list if the input is a list. Numeric representations are returned as integers.

Return type one of str, int, list of str, list of int

climada.util.coordinates.**country_iso2natid**(*isos*)
Convert ISO 3166-1 alpha-3 codes to internal NatIDs

Parameters **isos** (*str or list of str*) – ISO codes of countries (or single code).

Returns **natids** – Will only return a list if the input is a list.

Return type int or list of int

climada.util.coordinates.**natearth_country_to_int**(*country*)
Integer representation (ISO 3166, if possible) of Natural Earth GeoPandas country row

Parameters **country** (*GeoSeries*) – Row from Natural Earth GeoDataFrame.

Returns **iso_numeric** – Integer representation of given country.

Return type int

climada.util.coordinates.**get_country_code**(*lat*, *lon*, *gridded=False*)
Provide numeric (ISO 3166) code for every point.

Oceans get the value zero. Areas that are not in ISO 3166 are given values in the range above 900 according to NATEARTH_AREA_NONISO_NUMERIC.

Parameters

- **lat** (*np.array*) – latitude of points in epsg:4326
- **lon** (*np.array*) – longitude of points in epsg:4326
- **gridded** (*bool*) – If True, interpolate precomputed gridded data which is usually much faster. Default: False.

Returns **country_codes** – Numeric code for each point.

Return type np.array(int)

climada.util.coordinates.**get_admin1_info**(*country_names*)
Provide Natural Earth registry info and shape files for admin1 regions

Parameters **country_names** (*list or str*) – string or list with strings, either ISO code or names of countries, e.g.: ['ZWE', 'GBR', 'VNM', 'UZB', 'Kenya', '051'] For example, for Armenia, the following inputs work:

'Armenia', 'ARM', 'AM', '051', 51

Returns

- **admin1_info** (*dict*) – Data according to records in Natural Earth database.
- **admin1_shapes** (*dict*) – Shape according to Natural Earth.

climada.util.coordinates.**get_admin1_geometries**(*countries*)

return geometries, names and codes of admin 1 regions in given countries in a GeoDataFrame. If no admin 1 regions are defined, all regions in countries are returned.

Parameters **countries** (*list or str or int*) – string or list containing strings, either ISO3 code or ISO2 code or names names of countries, e.g.: ['ZWE', 'GBR', 'VNM', 'UZB', 'Kenya', '051'] For example, for Armenia, the following inputs work:

'Armenia', 'ARM', 'AM', '051', 51

Returns

gdf –

geopandas.GeoDataFrame instance with columns:

"admin1_name" [str] name of admin 1 region

"iso_3166_2" [str] iso code of admin 1 region

"geometry" [Polygon or MultiPolygon] shape of admin 1 region as shapely geometry object

"iso_3n" [str] numerical iso 3 code of country (admin 0)

"iso_3a" [str] alphabetical iso 3 code of country (admin 0)

Return type GeoDataFrame

climada.util.coordinates.**get_resolution_1d**(*coords*, *min_resol=1e-08*)

Compute resolution of scalar grid

Parameters

- **coords** (*np.array*) – scalar coordinates
- **min_resol** (*float, optional*) – minimum resolution to consider. Default: 1.0e-8.

Returns **res** – Resolution of given grid.

Return type float

climada.util.coordinates.**get_resolution**(**coords*, *min_resol=1e-08*)

Compute resolution of n-d grid points

Parameters

- **X, Y, ...** (*np.array*) – Scalar coordinates in each axis
- **min_resol** (*float, optional*) – minimum resolution to consider. Default: 1.0e-8.

Returns **resolution** – Resolution in each coordinate direction.

Return type pair of floats

climada.util.coordinates.**pts_to_raster_meta**(*points_bounds*, *res*)

Transform vector data coordinates to raster.

If a raster of the given resolution doesn't exactly fit the given bounds, the raster might have slightly larger (but never smaller) bounds.

Parameters

- **points_bounds** (*tuple*) – points total bounds (xmin, ymin, xmax, ymax)
- **res** (*tuple*) – resolution of output raster (xres, yres)

Returns

- **nrows** (*int*) – Number of rows.
- **ncols** (*int*) – Number of columns.
- **ras_trans** (*affine.Affine*) – Affine transformation defining the raster.

climada.util.coordinates.**raster_to_meshgrid**(*transform*, *width*, *height*)

Get coordinates of grid points in raster

Parameters

- **transform** (*affine.Affine*) – Affine transform defining the raster.
- **width** (*int*) – Number of points in first coordinate axis.
- **height** (*int*) – Number of points in second coordinate axis.

Returns

- **x** (*np.array*) – x-coordinates of grid points.
- **y** (*np.array*) – y-coordinates of grid points.

climada.util.coordinates.**to_crs_user_input**(*crs_obj*)

Returns a crs string or dictionary from a hdf5 file object.

bytes are decoded to str if the string starts with a '{' it is assumed to be a dumped string from a dictionary and ast is used to parse it.

Parameters **crs_obj** (*int, dict or str or bytes*) – the crs object to be converted user input

Returns to eventually be used as argument of rasterio.crs.CRS.from_user_input and pyproj.crs.CRS.from_user_input

Return type str or dict

Raises **ValueError** – if type(crs_obj) has the wrong type

climada.util.coordinates.**equal_crs**(*crs_one*, *crs_two*)

Compare two crs

Parameters

- **crs_one** (*dict, str or int*) – user crs
- **crs_two** (*dict, str or int*) – user crs

Returns **equal** – Whether the two specified CRS are equal according tho rasterio.crs.CRS.from_user_input

Return type bool

climada.util.coordinates.**read_raster**(*file_name*, *band=None*, *src_crs=None*, *window=None*, *geometry=None*, *dst_crs=None*, *transform=None*, *width=None*, *height=None*, *resampling='nearest'*)

Read raster of bands and set 0-values to the masked ones.

Parameters

- **file_name** (*str*) – name of the file
- **band** (*list(int), optional*) – band number to read. Default: 1
- **window** (*rasterio.windows.Window, optional*) – window to read
- **geometry** (*shapely.geometry, optional*) – consider pixels only in shape
- **dst_crs** (*crs, optional*) – reproject to given crs
- **transform** (*rasterio.Affine*) – affine transformation to apply
- **wdith** (*float*) – number of lons for transform
- **height** (*float*) – number of lats for transform
- **resampling** (*int or str, optional*) – Resampling method to use, encoded as an integer value (see *rasterio.enums.Resampling*). String values like *"nearest"* or *"bilinear"* are resolved to attributes of *rasterio.enums.Resampling*. Default: "nearest"

Returns

- **meta** (*dict*) – Raster meta (height, width, transform, crs).
- **data** (*np.array*) – Each row corresponds to one band (raster points are flattened, can be reshaped to height x width).

climada.util.coordinates.**read_raster_bounds**(*path*, *bounds*, *res=None*, *bands=None*, *resampling='nearest'*, *global_origin=None*, *pad_cells=1.0*)

Read raster file within given bounds at given resolution

By default, not only the grid cells of the destination raster whose cell centers fall within the specified bounds are selected, but one additional row/column of grid cells is added as a padding in each direction (pad_cells=1). This makes sure that the extent of the selected cell centers encloses the specified bounds.

The axis orientations (e.g. north to south, west to east) of the input data set are preserved.

Parameters

- **path** (*str*) – Path to raster file to open with rasterio.
- **bounds** (*tuple*) – (xmin, ymin, xmax, ymax)
- **res** (*float or pair of floats, optional*) – Resolution of output. Note that the orientation (sign) of these is overwritten by the input data set's axis orientations (e.g. north to south, west to east). Default: Resolution of input raster file.
- **bands** (*list of int, optional*) – Bands to read from the input raster file. Default: [1]
- **resampling** (*int or str, optional*) – Resampling method to use, encoded as an integer value (see *rasterio.enums.Resampling*). String values like *"nearest"* or *"bilinear"* are resolved to attributes of *rasterio.enums.Resampling*. Default: "nearest"
- **global_origin** (*pair of floats, optional*) – If given, align the output raster to a global reference raster with this origin. By default, the data set's origin (according to it's transform) is used.

- **pad_cells** (*float, optional*) – The number of cells to add as a padding (in terms of the destination grid that is inferred from *res* and/or *global_origin* if those parameters are given). This defaults to 1 to make sure that applying methods like bilinear interpolation to the output of this function is well-defined everywhere within the specified bounds. Default: 1.0

Returns

- **data** (*3d np.array*) – First dimension is for the selected raster bands. Second dimension is y (lat) and third dimension is x (lon).
- **transform** (*rasterio.Affine*) – Affine transformation defining the output raster data.

climada.util.coordinates.**read_raster_sample**(*path*, *lat*, *lon*, *intermediate_res=None*, *method='linear'*, *fill_value=None*)

Read point samples from raster file.

Parameters

- **path** (*str*) – path of the raster file
- **lat** (*np.array*) – latitudes in file's CRS
- **lon** (*np.array*) – longitudes in file's CRS
- **intermediate_res** (*float, optional*) – If given, the raster is not read in its original resolution but in the given one. This can increase performance for files of very high resolution.
- **method** (*str, optional*) – The interpolation method, passed to scipy.interp.interpn. Default: 'linear'.
- **fill_value** (*numeric, optional*) – The value used outside of the raster bounds. Default: The raster's nodata value or 0.

Returns **values** – Interpolated raster values for each given coordinate point.

Return type np.array of same length as lat

climada.util.coordinates.**interp_raster_data**(*data*, *interp_y*, *interp_x*, *transform*, *method='linear'*, *fill_value=0*)

Interpolate raster data, given as array and affine transform

Parameters

- **data** (*np.array*) – 2d numpy array containing the values
- **interp_y** (*np.array*) – y-coordinates of points (corresp. to first axis of data)
- **interp_x** (*np.array*) – x-coordinates of points (corresp. to second axis of data)
- **transform** (*affine.Affine*) – affine transform defining the raster
- **method** (*str, optional*) –

  The interpolation method, passed to scipy.interp.interpn. Default: 'linear'.
- **fill_value** (*numeric, optional*) –

  The value used outside of the raster bounds. Default: 0.

Returns **values** – Interpolated raster values for each given coordinate point.

Return type np.array

climada.util.coordinates.**refine_raster_data**(*data*, *transform*, *res*, *method='linear'*, *fill_value=0*)

Refine raster data, given as array and affine transform

Parameters

- **data** (*np.array*) – 2d array containing the values
- **transform** (*affine.Affine*) – affine transform defining the raster
- **res** (*float or pair of floats*) – new resolution
- **method** (*str, optional*) –

  The interpolation method, passed to scipy.interp.interpn. Default: 'linear'.

Returns

- **new_data** (*np.array*) – 2d array containing the interpolated values.
- **new_transform** (*affine.Affine*) – Affine transform defining the refined raster.

climada.util.coordinates.**read_vector**(*file_name*, *field_name*, *dst_crs=None*)

Read vector file format supported by fiona.

Parameters

- **file_name** (*str*) – vector file with format supported by fiona and 'geometry' field.
- **field_name** (*list(str)*) – list of names of the columns with values.
- **dst_crs** (*crs, optional*) – reproject to given crs

Returns

- **lat** (*np.array*) – Latitudinal coordinates.
- **lon** (*np.array*) – Longitudinal coordinates.
- **geometry** (*GeoSeries*) – Shape geometries.
- **value** (*np.array*) – Values associated to each shape.

climada.util.coordinates.**write_raster**(*file_name*, *data_matrix*, *meta*, *dtype=<class 'numpy.float32'>*)

Write raster in GeoTiff format.

Parameters

- **file_name** (*str*) – File name to write.
- **data_matrix** (*np.array*) – 2d raster data. Either containing one band, or every row is a band and the column represents the grid in 1d.
- **meta** (*dict*) – rasterio meta dictionary containing raster properties: width, height, crs and transform must be present at least. Include *compress="deflate"* for compressed output.
- **dtype** (*numpy dtype, optional*) – A numpy dtype. Default: np.float32

climada.util.coordinates.**points_to_raster**(*points_df*, *val_names=None*, *res=0.0*, *raster_res=0.0*, *crs='EPSG:4326'*, *scheduler=None*)

Compute raster (as data and transform) from GeoDataFrame.

Parameters

- **points_df** (*GeoDataFrame*) – contains columns latitude, longitude and those listed in the parameter *val_names*.
- **val_names** (*list of str, optional*) – The names of columns in *points_df* containing values. The raster will contain one band per column. Default: ['value']
- **res** (*float, optional*) – resolution of current data in units of latitude and longitude, approximated if not provided.

- **raster_res** (*float, optional*) – desired resolution of the raster
- **crs** (*object (anything accepted by pyproj.CRS.from_user_input), optional*) – If given, overwrites the CRS information given in *points_df*. If no CRS is explicitly given and there is no CRS information in *points_df*, the CRS is assumed to be EPSG:4326 (lat/lon). Default: None
- **scheduler** (*str*) – used for dask map_partitions. "threads", "synchronous" or "processes"

Returns

- **data** (*np.array*) – 3d array containing the raster values. The first dimension has the same size as *val_names* and represents the raster bands.
- **meta** (*dict*) – Dictionary with 'crs', 'height', 'width' and 'transform' attributes.

climada.util.coordinates.**subraster_from_bounds**(*transform*, *bounds*)

Compute a subraster definition from a given reference transform and bounds.

The axis orientations (sign of resolution step sizes) in *transform* are not required to be north to south and west to east. The given orientation is preserved in the result.

Parameters

- **transform** (*rasterio.Affine*) – Affine transformation defining the reference grid.
- **bounds** (*tuple of floats (xmin, ymin, xmax, ymax)*) – Bounds of the subraster in units and CRS of the reference grid.

Returns

- **dst_transform** (*rasterio.Affine*) – Subraster affine transformation. The axis orientations of the input transform (e.g. north to south, west to east) are preserved.
- **dst_shape** (*tuple of ints (height, width)*) – Number of pixels of subraster in vertical and horizontal direction.

climada.util.coordinates.**align_raster_data**(*source*, *src_crs*, *src_transform*, *dst_crs=None*, *dst_resolution=None*, *dst_bounds=None*, *global_origin=(-180, 90)*, *resampling='nearest'*, *conserve=None*, ***kwargs*)

Reproject 2D np.ndarray to be aligned to a reference grid.

This function ensures that reprojected data with the same dst_resolution and global_origins are aligned to the same global grid, i.e., no offset between destination grid points for different source grids that are projected to the same target resolution.

Note that the origin is required to be in the upper left corner. The result is always oriented left to right (west to east) and top to bottom (north to south).

Parameters

- **source** (*np.ndarray*) – The source is a 2D ndarray containing the values to be reprojected.
- **src_crs** (*CRS or dict*) – Source coordinate reference system, in rasterio dict format.
- **src_transform** (*rasterio.Affine*) – Source affine transformation.
- **dst_crs** (*CRS, optional*) – Target coordinate reference system, in rasterio dict format. Default: *src_crs*
- **dst_resolution** (*tuple (x_resolution, y_resolution) or float, optional*) – Target resolution (positive pixel sizes) in units of the target CRS. Default: *(abs(src_transform[0]), abs(src_transform[4]))*

- **dst_bounds** (*tuple of floats (xmin, ymin, xmax, ymax), optional*) – Bounds of the target raster in units of the target CRS. By default, the source's bounds are reprojected to the target CRS.
- **global_origin** (*tuple (west, north) of floats, optional*) – Coordinates of the reference grid's upper left corner. Default: (-180, 90). Make sure to change *global_origin* for non-geographical CRS!
- **resampling** (*int or str, optional*) – Resampling method to use, encoded as an integer value (see *rasterio.enums.Resampling*). String values like *"nearest"* or *"bilinear"* are resolved to attributes of *rasterio.enums.Resampling*. Default: "nearest"
- **conserve** (*str, optional*) – If provided, conserve the source array's 'mean' or 'sum' in the transformed data or normalize the values of the transformed data ndarray ('norm'). WARNING: Please note that this procedure will not apply any weighting of values according to the geographical cell sizes, which will introduce serious biases for lat/lon grids in case of areas spanning large latitudinal ranges. Default: None (no conservation)
- **kwargs** (*dict, optional*) – Additional arguments passed to *rasterio.warp.reproject*.

Raises `ValueError` –

Returns

- **destination** (np.ndarray with same dtype as *source*) – The transformed 2D ndarray.
- **dst_transform** (*rasterio.Affine*) – Destination affine transformation.

`climada.util.coordinates.`**`mask_raster_with_geometry`**(*raster*, *transform*, *shapes*, *nodata=None*, ***kwargs*)

Change values in *raster* that are outside of given *shapes* to *nodata*.

This function is a wrapper for rasterio.mask.mask to allow for in-memory processing. This is done by first writing data to memfile and then reading from it before the function call to rasterio.mask.mask(). The MemoryFile will be discarded after exiting the with statement.

Parameters

- **raster** (*numpy.ndarray*) – raster to be masked with dim: [H, W].
- **transform** (*affine.Affine*) – the transform of the raster.
- **shapes** (*GeoJSON-like dict or an object that implements the Python geo*) – interface protocol (such as a Shapely Polygon) Passed to rasterio.mask.mask
- **nodata** (*int or float, optional*) – Passed to rasterio.mask.mask: Data points outside *shapes* are set to *nodata*.
- **kwargs** (*optional*) – Passed to rasterio.mask.mask.

Returns **masked** – raster with dim: [H, W] and points outside shapes set to *nodata*

Return type numpy.ndarray or numpy.ma.MaskedArray

`climada.util.coordinates.`**`set_df_geometry_points`**(*df_val*, *scheduler=None*, *crs=None*)

Set given geometry to given dataframe using dask if scheduler.

Parameters

- **df_val** (*GeoDataFrame*) – contains latitude and longitude columns
- **scheduler** (*str, optional*) – used for dask map_partitions. "threads", "synchronous" or "processes"

- **crs** (*object (anything readable by pyproj4.CRS.from_user_input), optional*) – Coordinate Reference System, if omitted or None: df_val.geometry.crs

climada.util.coordinates.**fao_code_def**()
Generates list of FAO country codes and corresponding ISO numeric-3 codes.

Returns

- **iso_list** (*list*) – list of ISO numeric-3 codes
- **faocode_list** (*list*) – list of FAO country codes

climada.util.coordinates.**country_faocode2iso**(*input_fao*)
Convert FAO country code to ISO numeric-3 codes.

Parameters **input_fao** (*int or array*) – FAO country codes of countries (or single code)

Returns **output_iso** – ISO numeric-3 codes of countries (or single code)

Return type int or array

climada.util.coordinates.**country_iso2faocode**(*input_iso*)
Convert ISO numeric-3 codes to FAO country code.

Parameters **input_iso** (*iterable of int*) – ISO numeric-3 code(s) of country/countries

Returns **output_faocode** – FAO country code(s) of country/countries

Return type numpy.array

climada.util.dates_times module

climada.util.dates_times.**date_to_str**(*date*)
Compute date string in ISO format from input datetime ordinal int. :Parameters: **date** (*int or list or np.array*) – input datetime ordinal

Return type str or list(str)

climada.util.dates_times.**str_to_date**(*date*)
Compute datetime ordinal int from input date string in ISO format. :Parameters: **date** (*str or list*) – idate string in ISO format, e.g. '2018-04-06'

Return type int

climada.util.dates_times.**datetime64_to_ordinal**(*datetime*)
Converts from a numpy datetime64 object to an ordinal date. See https://stackoverflow.com/a/21916253 for the horrible details. :Parameters: **datetime** (*np.datetime64, or list or np.array*) – date and time

Return type int

climada.util.dates_times.**last_year**(*ordinal_vector*)
Extract first year from ordinal date

Parameters **ordinal_vector** (*list or np.array*) – input datetime ordinal

Return type int

climada.util.dates_times.**first_year**(*ordinal_vector*)
Extract first year from ordinal date

Parameters **ordinal_vector** (*list or np.array*) – input datetime ordinal

Return type int

climada.util.dwd_icon_loader module

climada.util.dwd_icon_loader.**download_icon_grib**(*run_datetime*, *model_name='icon-eu-eps'*, *parameter_name='vmax_10m'*, *max_lead_time=None*, *download_dir=None*)

download the gribfiles of a weather forecast run for a certain weather parameter from opendata.dwd.de/weather/nwp/.

Parameters

- **run_datetime** (*datetime*) – The starting timepoint of the forecast run
- **model_name** (*str*) – the name of the forecast model written as it appears in the folder structure in opendata.dwd.de/weather/nwp/ or 'test'
- **parameter_name** (*str*) – the name of the meteorological parameter written as it appears in the folder structure in opendata.dwd.de/weather/nwp/
- **max_lead_time** (*int*) – number of hours for which files should be downloaded, will default to maximum available data
- **download_dir** (*: str or Path*) – directory where the downloaded files should be saved in

Returns **file_names** – a list of filenames that link to all just downloaded or available files from the forecast run, defined by the input parameters

Return type list

climada.util.dwd_icon_loader.**delete_icon_grib**(*run_datetime*, *model_name='icon-eu-eps'*, *parameter_name='vmax_10m'*, *max_lead_time=None*, *download_dir=None*)

delete the downloaded gribfiles of a weather forecast run for a certain weather parameter from opendata.dwd.de/weather/nwp/.

Parameters

- **run_datetime** (*datetime*) – The starting timepoint of the forecast run
- **model_name** (*str*) – the name of the forecast model written as it appears in the folder structure in opendata.dwd.de/weather/nwp/
- **parameter_name** (*str*) – the name of the meteorological parameter written as it appears in the folder structure in opendata.dwd.de/weather/nwp/
- **max_lead_time** (*int*) – number of hours for which files should be deleted, will default to maximum available data
- **download_dir** (*str or Path*) – directory where the downloaded files are stored at the moment

climada.util.dwd_icon_loader.**download_icon_centroids_file**(*model_name='icon-eu-eps'*, *download_dir=None*)

create centroids based on netcdf files provided by dwd, links found here: https://www.dwd.de/DE/leistungen/opendata/neuigkeiten/opendata_dez2018_02.html https://www.dwd.de/DE/leistungen/opendata/neuigkeiten/opendata_aug2020_01.html

Parameters

- **model_name** (*str*) – the name of the forecast model written as it appears in the folder structure in opendata.dwd.de/weather/nwp/
- **download_dir** (*str or Path*) – directory where the downloaded files should be saved in

Returns **file_name** – absolute path and filename of the downloaded and decompressed netcdf file

Return type str

climada.util.earth_engine module

climada.util.files_handler module

climada.util.files_handler.**to_list**(*num_exp*, *values*, *val_name*)
Check size and transform to list if necessary. If size is one, build a list with num_exp repeated values.

Parameters

- **num_exp** (*int*) – expected number of list elements
- **values** (*object or list(object)*) – values to check and transform
- **val_name** (*str*) – name of the variable values

Return type list

climada.util.files_handler.**get_file_names**(*file_name*)
Return list of files contained. Supports globbing.

Parameters **file_name** (*str or list(str)*) – Either a single string or a list of strings that are either - a file path - or the path of the folder containing the files - or a globbing pattern.

Return type list(str)

climada.util.finance module

climada.util.finance.**net_present_value**(*years*, *disc_rates*, *val_years*)
Compute net present value.

Parameters

- **years** (*np.array*) – array with the sequence of years to consider.
- **disc_rates** (*np.array*) – discount rate for every year in years.
- **val_years** (*np.array*) – chash flow at each year.

Return type float

climada.util.finance.**income_group**(*cntry_iso*, *ref_year*, *shp_file=None*)
Get country's income group from World Bank's data at a given year, or closest year value. If no data, get the natural earth's approximation.

Parameters

- **cntry_iso** (*str*) – key = ISO alpha_3 country
- **ref_year** (*int*) – reference year
- **shp_file** (*cartopy.io.shapereader.Reader, optional*) – shape file with INCOME_GRP attribute for every country. Load Natural Earth admin0 if not provided.

climada.util.finance.**gdp**(*cntry_iso*, *ref_year*, *shp_file=None*, *per_capita=False*)
Get country's (current value) GDP from World Bank's data at a given year, or closest year value. If no data, get the natural earth's approximation.

Parameters

- **cntry_iso** (*str*) – key = ISO alpha_3 country

- **ref_year** (*int*) – reference year
- **shp_file** (*cartopy.io.shapereader.Reader, optional*) – shape file with INCOME_GRP attribute for every country. Load Natural Earth admin0 if not provided.
- **per_capita** (*boolean, optional*) – If True, GDP is returned per capita

Return type float

climada.util.hdf5_handler module

climada.util.hdf5_handler.**read**(*file_name*, *with_refs=False*)
Load a hdf5 data structure from a file.

Parameters

- **file_name** – file to load
- **with_refs** – enable loading of the references. Default is unset, since it increments the execution time considerably.

Returns dictionary structure containing all the variables.

Return type contents

Examples

```
>>> # Contents contains the Matlab data in a dictionary.
>>> contents = read("/pathto/dummy.mat")
>>> # Contents contains the Matlab data and its reference in a dictionary.
>>> contents = read("/pathto/dummy.mat", True)
```

Raises **Exception while reading** –

climada.util.hdf5_handler.**get_string**(*array*)
Form string from input array of unisgned integers.

Parameters **array** – array of integers

Return type string

climada.util.hdf5_handler.**get_str_from_ref**(*file_name*, *var*)
Form string from a reference HDF5 variable of the given file.

Parameters

- **file_name** – matlab file name
- **var** – HDF5 reference variable

Return type string

climada.util.hdf5_handler.**get_list_str_from_ref**(*file_name*, *var*)
Form list of strings from a reference HDF5 variable of the given file.

Parameters

- **file_name** – matlab file name
- **var** – array of HDF5 reference variable

Return type string

climada.util.hdf5_handler.**get_sparse_csr_mat**(*mat_dict*, *shape*)

Form sparse matrix from input hdf5 sparse matrix data type.

Parameters

- **mat_dict** – dictionary containing the sparse matrix information.
- **shape** – tuple describing output matrix shape.

Return type sparse csr matrix

climada.util.plot module

climada.util.plot.**geo_bin_from_array**(*array_sub*, *geo_coord*, *var_name*, *title*, *pop_name=True*, *buffer=1.0*, *extend='neither'*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *shapes=True*, *axes=None*, *figsize=(9, 13)*, *adapt_fontsize=True*, ***kwargs*)

Plot array values binned over input coordinates.

Parameters

- **array_sub** (*np.array(1d or 2d) or list(np.array)*) – Each array (in a row or in the list) are values at each point in corresponding geo_coord that are binned in one subplot.
- **geo_coord** (*2d np.array or list(2d np.array)*) – (lat, lon) for each point in a row. If one provided, the same grid is used for all subplots. Otherwise provide as many as subplots in array_sub.
- **var_name** (*str or list(str)*) – label to be shown in the colorbar. If one provided, the same is used for all subplots. Otherwise provide as many as subplots in array_sub.
- **title** (*str or list(str)*) – subplot title. If one provided, the same is used for all subplots. Otherwise provide as many as subplots in array_sub.
- **pop_name** (*bool, optional*) – add names of the populated places, by default True.
- **buffer** (*float, optional*) – border to add to coordinates, by default BUFFER
- **extend** (*str, optional*) – extend border colorbar with arrows. ['neither' | 'both' | 'min' | 'max'], by default 'neither'
- **proj** (*ccrs, optional*) – coordinate reference system of the given data, by default ccrs.PlateCarree()
- **shapes** (*bool, optional*) – Overlay Earth's countries coastlines to matplotlib.pyplot axis. The default is True
- **axes** (*Axes or ndarray(Axes), optional*) – by default None
- **figsize** (*tuple, optional*) – figure size for plt.subplots, by default (9, 13)
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- ****kwargs** – arbitrary keyword arguments for hexbin matplotlib function

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

Raises `ValueError:` – Input array size missmatch

climada.util.plot.**geo_im_from_array**(*array_sub*, *coord*, *var_name*, *title*, *proj=None*, *smooth=True*, *axes=None*, *figsize=(9, 13)*, *adapt_fontsize=True*, ***kwargs*)

Image(s) plot defined in array(s) over input coordinates.

Parameters

- **array_sub** (*np.array(1d or 2d) or list(np.array)*) – Each array (in a row or in the list) are values at each point in corresponding geo_coord that are ploted in one subplot.
- **coord** (*2d np.array*) – (lat, lon) for each point in a row. The same grid is used for all subplots.
- **var_name** (*str or list(str)*) – label to be shown in the colorbar. If one provided, the same is used for all subplots. Otherwise provide as many as subplots in array_sub.
- **title** (*str or list(str)*) – subplot title. If one provided, the same is used for all subplots. Otherwise provide as many as subplots in array_sub.
- **proj** (*ccrs, optional*) – coordinate reference system used in coordinates, by default None
- **smooth** (*bool, optional*) – smooth plot to RESOLUTIONxRESOLUTION, by default True
- **axes** (*Axes or ndarray(Axes), optional*) – by default None
- **figsize** (*tuple, optional*) – figure size for plt.subplots, by default (9, 13)
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.
- ****kwargs** – arbitrary keyword arguments for pcolormesh matplotlib function

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

Raises `ValueError` –

climada.util.plot.**make_map**(*num_sub=1*, *figsize=(9, 13)*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *adapt_fontsize=True*)

Create map figure with cartopy.

Parameters

- **num_sub** (*int or tuple*) – number of total subplots in figure OR number of subfigures in row and column: (num_row, num_col).
- **figsize** (*tuple*) – figure size for plt.subplots
- **proj** (*cartopy.crs projection, optional*) – geographical projection, The default is PlateCarree default.
- **adapt_fontsize** (*bool, optional*) – If set to true, the size of the fonts will be adapted to the size of the figure. Otherwise the default matplotlib font size is used. Default is True.

Returns **fig, axis_sub**

Return type matplotlib.figure.Figure, cartopy.mpl.geoaxes.GeoAxesSubplot

climada.util.plot.**add_shapes**(*axis*)

Overlay Earth's countries coastlines to matplotlib.pyplot axis.

Parameters

- **axis** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – Cartopy axis
- **projection** (*cartopy.crs projection, optional*) – Geographical projection. The default is PlateCarree.

climada.util.plot.**add_populated_places**(*axis*, *extent*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *fontsize=None*)

Add city names.

Parameters

- **axis** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – cartopy axis.
- **extent** (*list*) – geographical limits [min_lon, max_lon, min_lat, max_lat]
- **proj** (*cartopy.crs projection, optional*) – geographical projection, The default is PlateCarree.
- **fontsize** (*int, optional*) – Size of the fonts. If set to None, the default matplotlib settings are used.

climada.util.plot.**add_cntry_names**(*axis*, *extent*, *proj=<Derived Projected CRS: +proj=eqc +ellps=WGS84 +a=6378137.0 +lon_0=0.0 +to ...> Name: unknown Axis Info [cartesian]: - E[east]: Easting (unknown) - N[north]: Northing (unknown) - h[up]: Ellipsoidal height (metre) Area of Use: - undefined Coordinate Operation: - name: unknown - method: Equidistant Cylindrical Datum: unknown - Ellipsoid: WGS 84 - Prime Meridian: Greenwich*, *fontsize=None*)

Add country names.

Parameters

- **axis** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – Cartopy axis.
- **extent** (*list*) – geographical limits [min_lon, max_lon, min_lat, max_lat]
- **proj (*cartopy.crs projection, optional*) – Geographical projection.** The default is PlateCarree.

  fontsize [int, optional] Size of the fonts. If set to None, the default matplotlib settings are used.

climada.util.save module

climada.util.save.**save**(*out_file_name*, *var*)

Save variable with provided file name. Uses configuration save_dir folder if no absolute path provided.

Parameters

- **out_file_name** (*str*) – file name (absolute path or relative to configured save_dir)
- **var** (*object*) – variable to save in pickle format

climada.util.save.**load**(*in_file_name*)

Load variable contained in file. Uses configuration save_dir folder if no absolute path provided.

Parameters **in_file_name** (*str*) – file name

Return type object

climada.util.scalebar_plot module

climada.util.scalebar_plot.**scale_bar**(*ax*, *location*, *length*, *metres_per_unit=1000*, *unit_name='km'*, *tol=0.01*, *angle=0*, *color='black'*, *linewidth=3*, *text_offset=0.005*, *ha='center'*, *va='bottom'*, *plot_kwargs=None*, *text_kwargs=None*, ***kwargs*)

Add a scale bar to CartoPy axes.

For angles between 0 and 90 the text and line may be plotted at slightly different angles for unknown reasons. To work around this, override the 'rotation' keyword argument with text_kwargs.

Parameters

- **ax** – CartoPy axes.
- **location** – Position of left-side of bar in axes coordinates.
- **length** – Geodesic length of the scale bar.
- **metres_per_unit** – Number of metres in the given unit. Default: 1000
- **unit_name** – Name of the given unit. Default: 'km'
- **tol** – Allowed relative error in length of bar. Default: 0.01
- **angle** – Anti-clockwise rotation of the bar.
- **color** – Color of the bar and text. Default: 'black'
- **linewidth** – Same argument as for plot.
- **text_offset** – Perpendicular offset for text in axes coordinates. Default: 0.005
- **ha** – Horizontal alignment. Default: 'center'
- **va** – Vertical alignment. Default: 'bottom'
- **plot_kwargs** – Keyword arguments for plot, overridden by **kwargs.
- **text_kwargs** – Keyword arguments for text, overridden by **kwargs.
- **kwargs** – Keyword arguments for both plot and text.

climada.util.select module

climada.util.select.**get_attributes_with_matching_dimension**(*obj*, *dims*)

Get the attributes of an object that have len(dims) number of dimensions or more, and all dims are individual parts of the attribute's shape.

Parameters

- **obj** (*object of any class*) – The object from which matching attributes are returned
- **dims** (*list[int]*) – List of dimensions size to match

Returns **list_of_attrs** – List of names of the attributes with matching dimensions

Return type list[str]

climada.util.value_representation module

climada.util.value_representation.**sig_dig**(*x*, *n_sig_dig=16*)
Rounds x to n_sig_dig number of significant digits. 0, inf, Nan are returned unchanged.

Examples

with n_sig_dig = 5:

1.234567 -> 1.2346, 123456.89 -> 123460.0

Parameters

- **x** (*float*) – number to be rounded
- **n_sig_dig** (*int, optional*) – Number of significant digits. The default is 16.

Returns Rounded number

Return type float

climada.util.value_representation.**sig_dig_list**(*iterable*, *n_sig_dig=16*)
Vectorized form of sig_dig. Rounds a list of float to a number of significant digits

Parameters

- **iterable** (*iter(float)*) – iterable of numbers to be rounded
- **n_sig_dig** (*int, optional*) – Number of significant digits. The default is 16.

Returns list of rounded floats

Return type list

climada.util.value_representation.**convert_monetary_value**(*values*, *abbrev*, *n_sig_dig=None*)

climada.util.value_representation.**value_to_monetary_unit**(*values*, *n_sig_dig=None*, *abbreviations=None*)
Converts list of values to closest common monetary unit.

0, Nan and inf have not unit.

Parameters

- **values** (*int or float, list(int or float) or np.ndarray(int or float)*) – Values to be converted
- **n_sig_dig** (*int, optional*) – Number of significant digits to return.

  Examples: n_sig_di=5: 1.234567 -> 1.2346, 123456.89 -> 123460.0

  Default: all digits are returned.
- **abbreviations** (*dict, optional*) – Name of the abbreviations for the money 1000s counts

  Default: { 1:'', 1000: 'K', 1000000: 'M', 1000000000: 'Bn', 1000000000000: 'Tn' }

Returns

- **mon_val** (*np.ndarray*) – Array of values in monetary unit
- **name** (*string*) – Monetary unit

Examples

values = [1e6, 2*1e6, 4.5*1e7, 0, Nan, inf] -> [1, 2, 4.5, 0, Nan, inf] ['M']

climada.util.yearsets module

climada.util.yearsets.**impact_yearset**(*imp*, *sampled_years*, *lam=None*, *correction_fac=True*, *seed=None*)

Create a yearset of impacts (yimp) containing a probabilistic impact for each year in the sampled_years list by sampling events from the impact received as input with a Poisson distribution centered around lam per year (lam = sum(imp.frequency)). In contrast to the expected annual impact (eai) yimp contains impact values that differ among years. When correction factor is true, the yimp are scaled such that the average over all years is equal to the eai.

Parameters

- **imp** (*climada.engine.Impact()*) – impact object containing impacts per event
- **sampled_years** (*list*) – A list of years that shall be covered by the resulting yimp.
- **seed** (*Any, optional*) – seed for the default bit generator default: None

Optional parameters

lam: int The applied Poisson distribution is centered around lam events per year. If no lambda value is given, the default lam = sum(imp.frequency) is used.

correction_fac [boolean] If True a correction factor is applied to the resulting yimp. It is scaled in such a way that the expected annual impact (eai) of the yimp equals the eai of the input impact

Returns

- **yimp** (*climada.engine.Impact()*) – yearset of impacts containing annual impacts for all sampled_years
- **sampling_vect** (*2D array*) – The sampling vector specifies how to sample the yimp, it consists of one sub-array per sampled_year, which contains the event_ids of the events used to calculate the annual impacts. Can be used to re-create the exact same yimp.

climada.util.yearsets.**impact_yearset_from_sampling_vect**(*imp*, *sampled_years*, *sampling_vect*, *correction_fac=True*)

Create a yearset of impacts (yimp) containing a probabilistic impact for each year in the sampled_years list by sampling events from the impact received as input following the sampling vector provided. In contrast to the expected annual impact (eai) yimp contains impact values that differ among years. When correction factor is true, the yimp are scaled such that the average over all years is equal to the eai.

Parameters

- **imp** (*climada.engine.Impact()*) – impact object containing impacts per event
- **sampled_years** (*list*) – A list of years that shall be covered by the resulting yimp.
- **sampling_vect** (*2D array*) – The sampling vector specifies how to sample the yimp, it consists of one sub-array per sampled_year, which contains the event_ids of the events used to calculate the annual impacts. It needs to be obtained in a first call, i.e. [yimp, sampling_vect] = climada_yearsets.impact_yearset(...) and can then be provided in this function to obtain the exact same sampling (also for a different imp object)

Optional parameter

correction_fac [boolean] If True a correction factor is applied to the resulting yimp. It is scaled in such a way that the expected annual impact (eai) of the yimp equals the eai of the input impact

Returns **yimp** – yearset of impacts containing annual impacts for all sampled_years

Return type climada.engine.Impact()

climada.util.yearsets.**sample_from_poisson**(*n_sampled_years*, *lam*, *seed=None*)
Sample the number of events for n_sampled_years

Parameters

- **n_sampled_years** (*int*) – The target number of years the impact yearset shall contain.
- **lam** (*int*) – the applied Poisson distribution is centered around lambda events per year
- **seed** (*int, optional*) – seed for numpy.random, will be set if not None default: None

Returns **events_per_year** – Number of events per sampled year

Return type array

climada.util.yearsets.**sample_events**(*events_per_year*, *freqs_orig*, *seed=None*)
Sample events uniformely from an array (indices_orig) without replacement (if sum(events_per_year) > n_input_events the input events are repeated (tot_n_events/n_input_events) times, by ensuring that the same events doens't occur more than once per sampled year).

Parameters

- **events_per_year** (*array*) – Number of events per sampled year
- **freqs_orig** (*array*) – Frequency of each input event
- **seed** (*Any, optional*) – seed for the default bit generator. Default: None

Returns **sampling_vect** – The sampling vector specifies how to sample the yimp, it consists of one sub-array per sampled_year, which contains the event_ids of the events used to calculate the annual impacts.

Return type 2D array

climada.util.yearsets.**compute_imp_per_year**(*imp*, *sampling_vect*)
Sample annual impacts from the given event_impacts according to the sampling dictionary

Parameters

- **imp** (*climada.engine.Impact()*) – impact object containing impacts per event
- **sampling_vect** (*2D array*) – The sampling vector specifies how to sample the yimp, it consists of one sub-array per sampled_year, which contains the event_ids of the events used to calculate the annual impacts.

Returns **imp_per_year** – Sampled impact per year (length = sampled_years)

Return type array

climada.util.yearsets.**calculate_correction_fac**(*imp_per_year*, *imp*)
Calculate a correction factor that can be used to scale the yimp in such a way that the expected annual impact (eai) of the yimp amounts to the eai of the input imp

Parameters

- **imp_per_year** (*array*) – sampled yimp
- **imp** (*climada.engine.Impact()*) – impact object containing impacts per event

Returns **correction_factor** – The correction factor is calculated as imp_eai/yimp_eai

Return type int

- genindex
- modindex

CHAPTER

EIGHT

LICENSE

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

Y