ClimateSim User Manual

Release 1.0

Erik Dahlström, Carl Dehlin, Marcus Ekström, Kerstin Söderqvist

Dec 08, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

This is the official user and developer manual for the ClimateSim project. The manual contains three logical sections, each section can be used independently for different purposes. The user guide explains how you as end user can use the application when up and running. The installation guide contains the necessary information for installing and setting up the project on a server. Finally, the code base documentation serves as a reference for further development of the project.

CHAPTER

TWO

END USER GUIDE

This is the end user guide for using the ClimateSim application. The guide will show the user what they can do with the application and how to do it. The instructions are presented with images with corresponding text describing what is being done at each step.

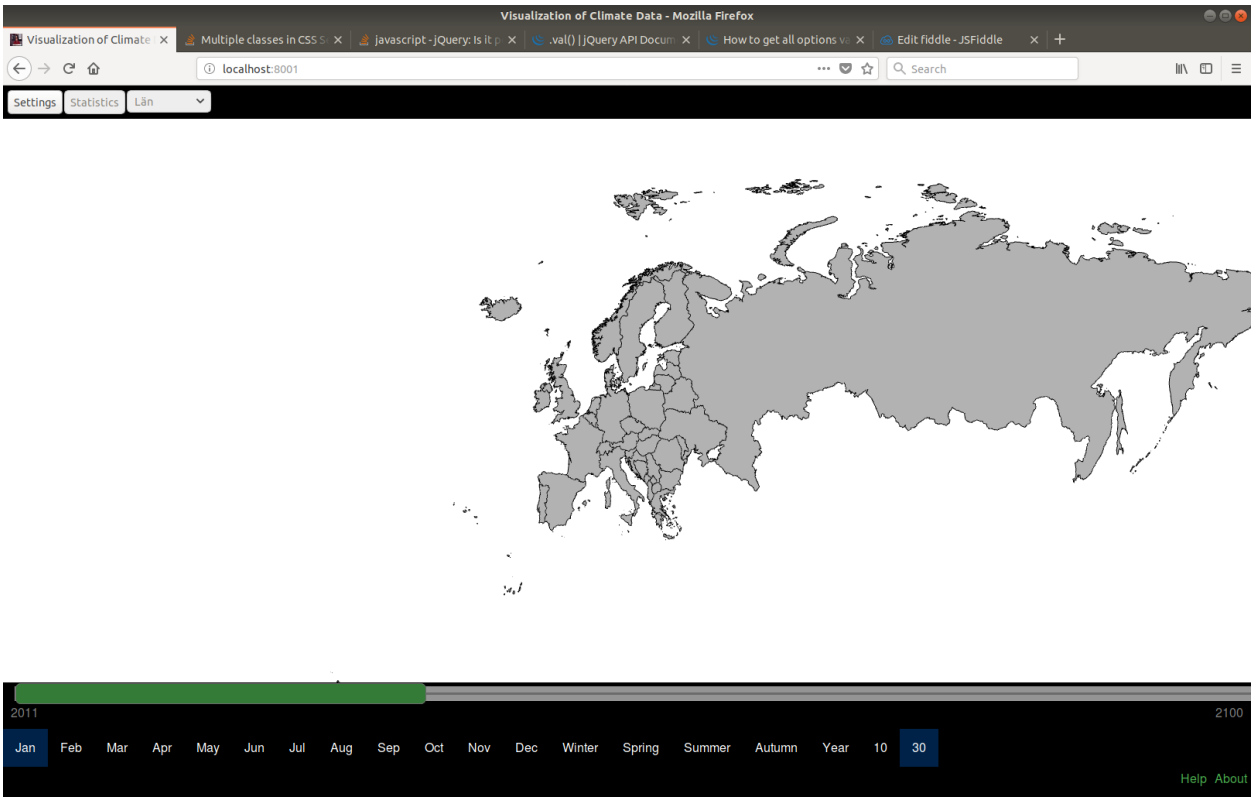
2.1 Starting the application

To run the application a browser with WebGL support must be installed on the local machine. This is usally supported but older browsers may lack this functionality. If this feature is not supported by the browser a message will show up telling the user this is the case.

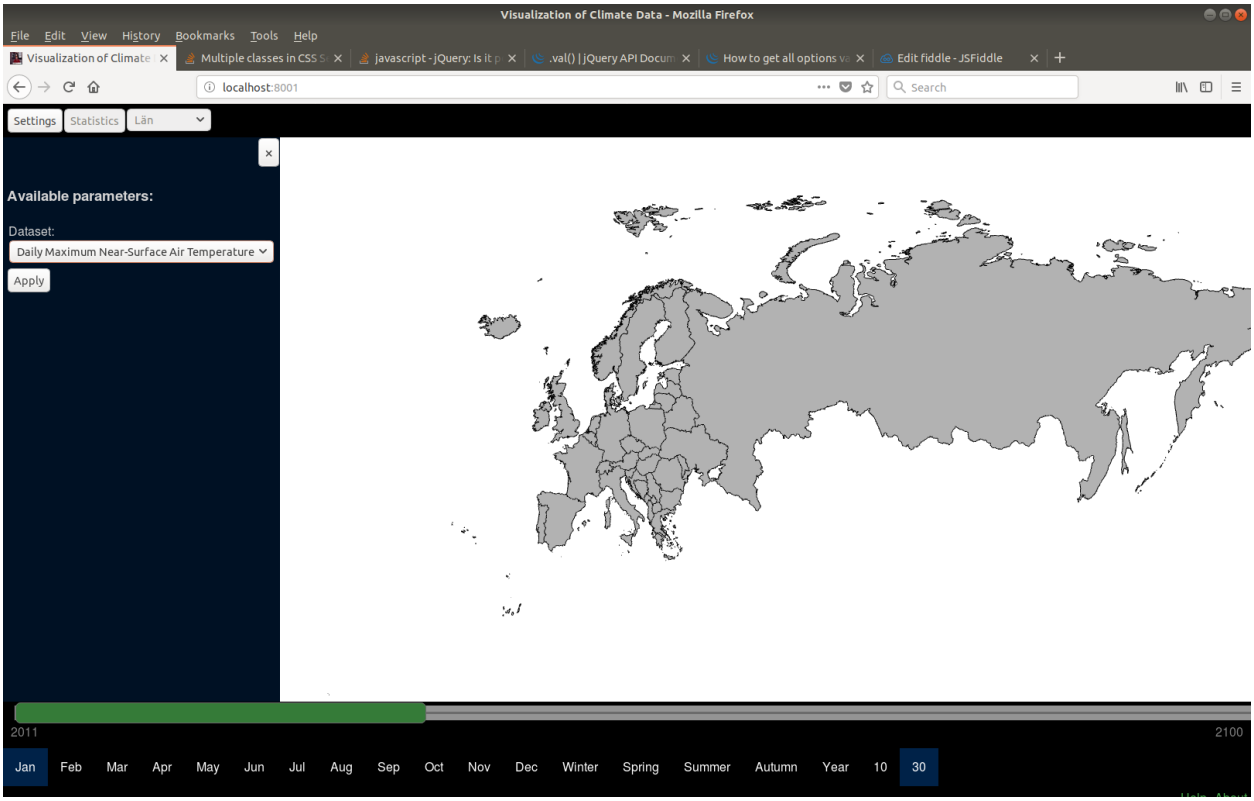
The application is started by entering the url http://climatesim.ddns.net in the browser. If running the application locally, start the server by running *./server -i localhost* in a terminal from the project root directory. Note that this requires root privileges. When the server has started and done all initialization, enter http://localhost in the browser. Observe that if running locally you must first perform the steps described in *Installation guide*.

2.2 Navigating the user interface

The following section will show the user how to nagivate the application from a browser. After the application is opened and done loading in the browser window, the user can start to explore climate data. In the following we illustrate how the appliation typically would be used.



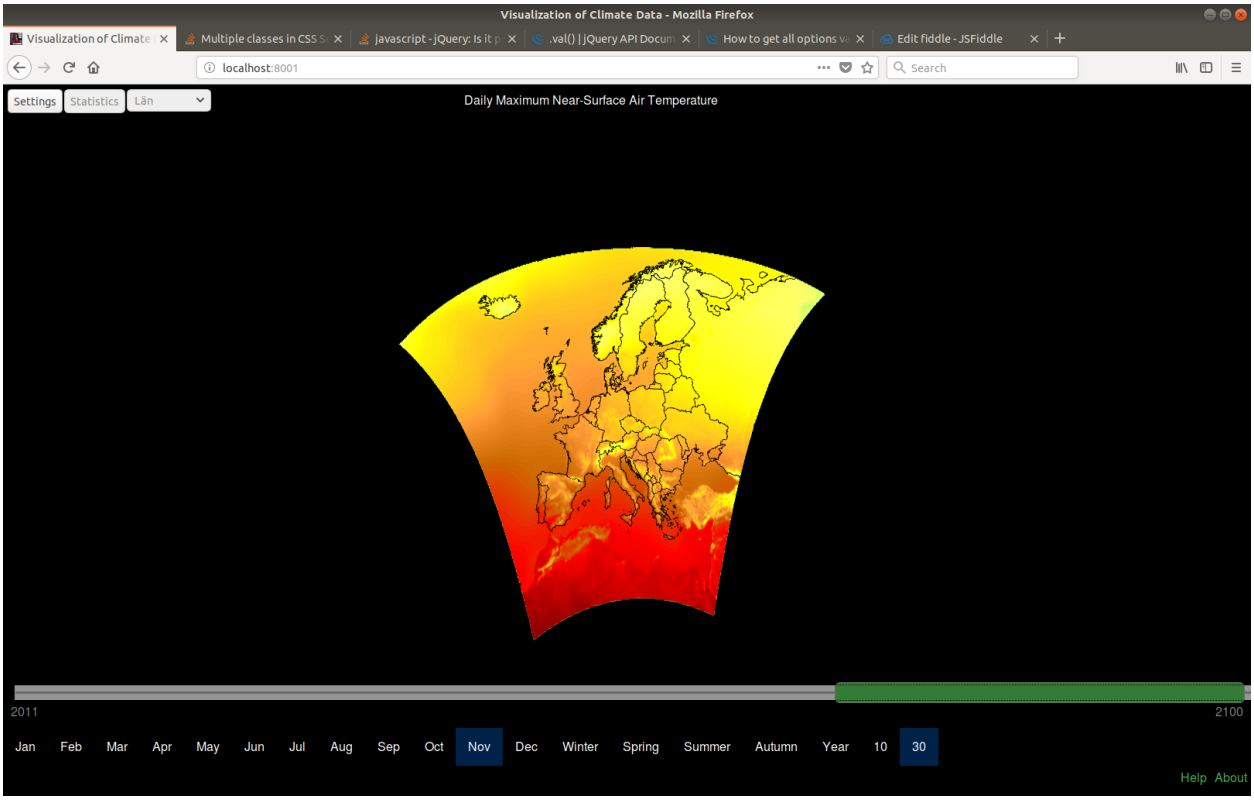
This is the view first presented when opening the application.



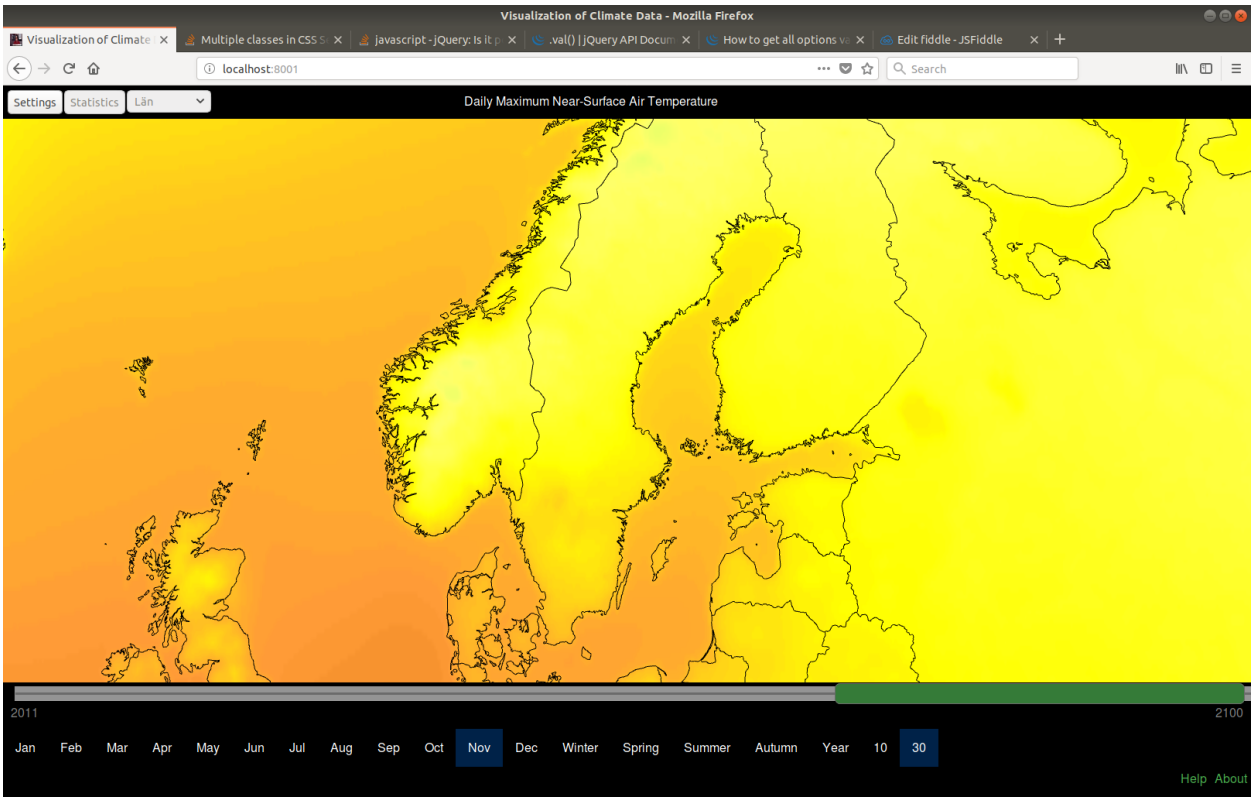
To select a parameter, click the settings button in the upper left corner. Clicking the button makes a navigation bar

unfold. Choose your parameter from the dropdown menu Dataset, then click Apply.

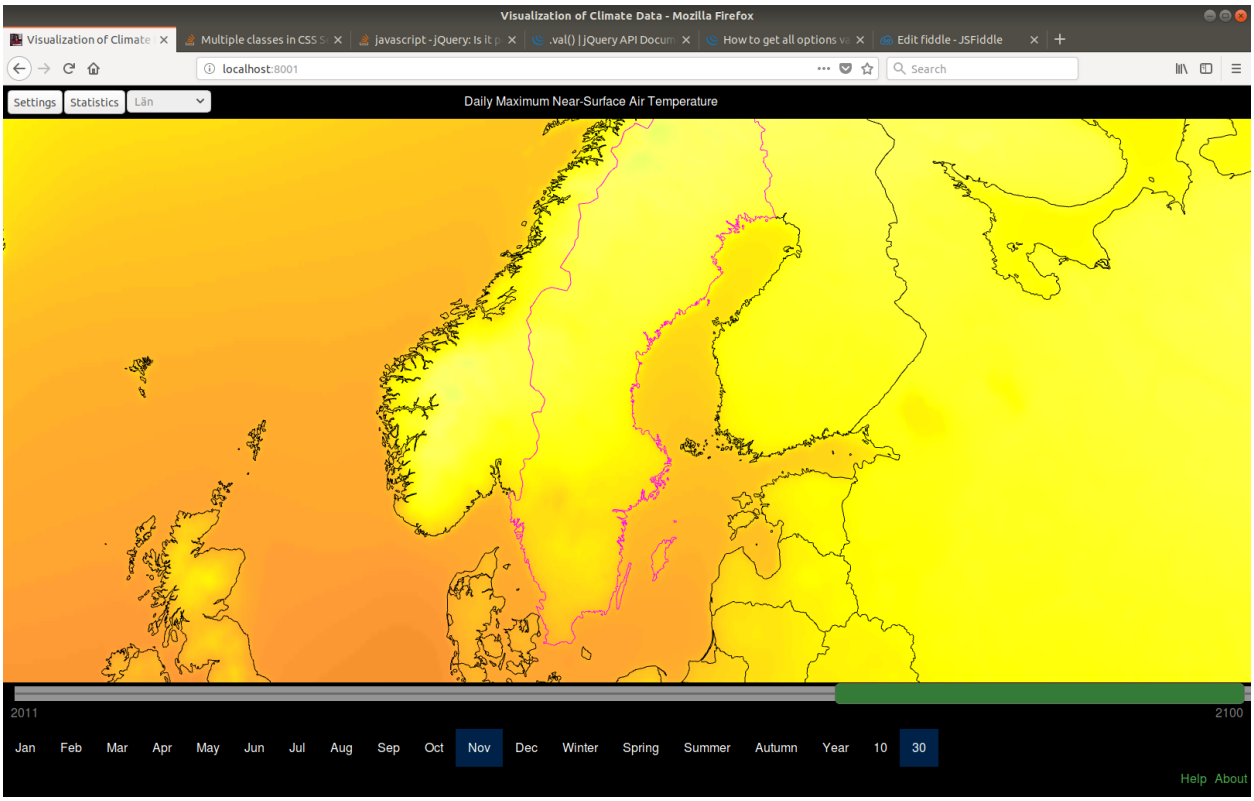
Depending on your download speed the data will be shown a moment later. If running the server on your localmachine the data should appear do arrive instantly. If running on a rmote serer this may however take up to a second depending on the network speed. When the data arrives it will be overlayed ontop of the map of europe.



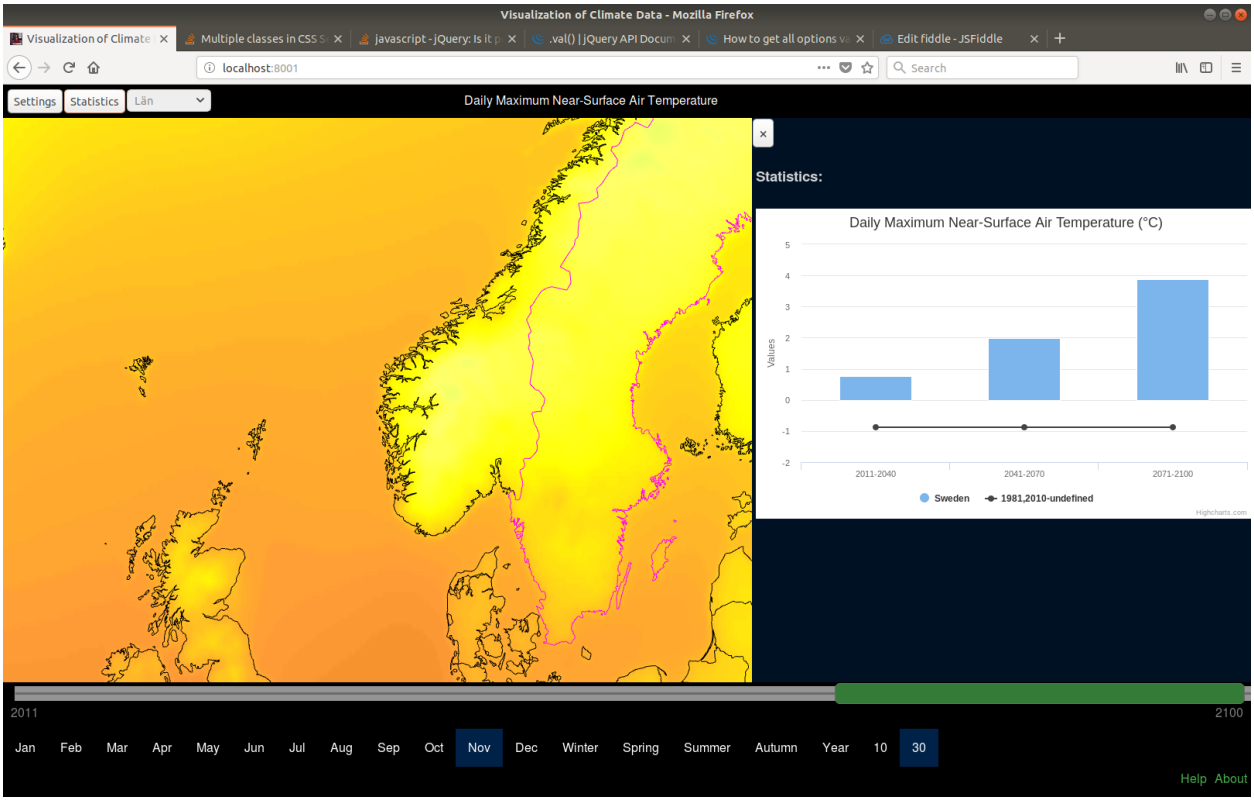
To select year interval to display, drag the slider at the bottom. To select different periods such as month, season or simply all year average of the data, click any of the buttons at the bottom of the page.



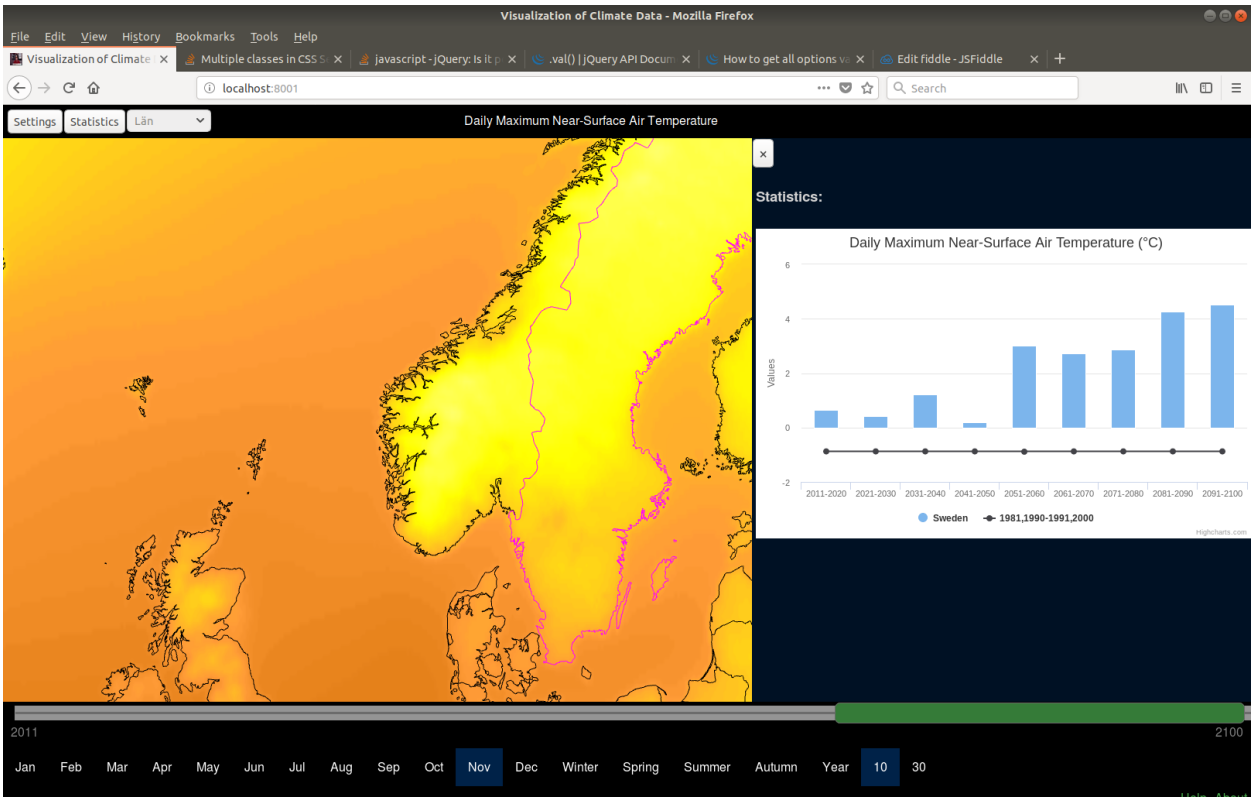
Drag with the mouse to move the around. Use the mouse scroll wheel / trackpad to zoom in and out. Now, zoom in on Sweden.



To select a country, just click on it. The selected country border will be highlighted in pink to clarify the selection.



To show statistics for the selected region, click the statistics button in the upper left corner. Clicking the statistics button makes a plot appear on the right.



Now try to switch the interval to average the data over. This can be done by selecting any of the buttons with numbers at the buttom, such as 10 or 30.

CHAPTER

THREE

INSTALLATION GUIDE

This section provides the documentation necessary for installing and setting up the project on a server. The guide does not assume the reader to have any deep knowledge about the project inner structure such as the code base. However, the reader is assumed to have some experience with a unix shell. After the installation a python flask server can be started running either locally on the computer or on an external IP address. The installation guide assumes a clean installation of Ubuntu 17.01 on a machine with atleast 8 GB RAM and 128 GB disk space available.

Warning: The server framework packaged with this project is only a development server. Support for handling many users by dispatching client requests to worker threads or advanced security options is not considered at all. The application can be configured to run through a more dedicated server such as Apache together with WSGI scripts. This is left to the reader of this document to configure if necessary.

3.1 System depencencies

This section contains instructions for installing required dependencies for running this project. The instructions are executed in a linux terminal, and assumes a clean installation of Ubuntu 17.01.

Start the installation by opening a terminal. All commands needs to be executed as root user. To execute them as root user, prefix all commands with *sudo* or become super user by typing *su* and then type all commands as written here.

3.1.1 Basic installation

The following commands will install all required dependencies for starting the server, handling of data etc. Observe that it may be necessary to execute them in the order described here. Each command may ask the user for permission to install or take additional space on the computer, so this is a manuall installation with no guarantees that it will work if done automatically. If any dependency fails to install, double check that the command is typed correctly and that the package is up to date. After installing all dependencies, if the server still can not be started then inspect the error log and install further required dependencies.

```
apt-get install pkg-config
apt-get install autoconf
apt-get install libtool
apt-get install zlib1g-dev
apt-get install libhdf5-dev / apt-get install libhdf5-serial-dev
apt-get install libcurl4-openssl-dev
apt-get install python-pip
apt-get install openssh-server
apt-get install python3-tk
apt-get install cdo
```

```
apt-get install nco
pip3 install numpy
pip3 install scipy
pip3 install bs4
pip3 install pyshp
pip3 install matplotlib
pip3 install flask
pip3 install simplejson
```

3.1.2 Dependencies for making the docs

This manual is built using Sphinx. To make this documenation guide, further dependencies must be installed.

```
apt-get install npm
pip3 install sphinx
pip3 install sphinx-js
pip3 install sphinxcontrib-plantuml
npm install -g jsdoc
apt-get install latexmk
apt-get install graphviz
```

3.2 Project setup

After performing the steps described in *System depencencies*, this section explains how to setup and run this project. The necessary steps before running a useful applications is getting data. After that further data may optionally be created. Finally all the data needs to be preprocessed in order for the application to run in real time.

3.2.1 Getting data

- Get an ESGF OpenID licence.
- Open a terminal. From the project root directory, type **cd data**.
- **type make**
  - Enter OpenID license url
  - Enter OpenID password
- All required data are now being downloaded (this will take some time, and space)
- If you want more files you can search for them in the Cordex database.

To download new data, either do it manually file by file and put them under **data/raw/<parameter_name>**. The data is assumed to cover the years 2011 - 2100. Also the program expects a corresponding reference dataset covering the years 1981 - 2010 to exist. The reference period data file/files needs to be downloaded and put under **data/raw_ref/<parameter_name>**. Here **<parameter_name>** is the keyword for the parameter in the dataset. For example, if downloading precipitation data the keyword for the parameter is **pr**, so the files for the years 2011-2100 should be put under **data/raw/pr** and the files for the years 1981-2010 should be put under **data/raw_ref/pr**.

Alternatively, download the wget-scripts for the datasets, put the scripts under data/wget-scripts and add two new lines in data/Makefile (one line for the data and one line for the reference data).

The wget script name should be on the form

```
wget-<informative-name>.sh
wget-<informative-name-for-reference-data>.sh
```

The corresponding lines in **data/Makefile** should be

```
raw/<parameter-keyname>:
     $(call, get-data,<informative-name>)
raw_ref/<parameter-keyname>:
     $(call, get-data,<informative-name-for-reference-data>)
```

where **<informative-name>** and **<informative-name-for-reference-data>** can be any unique file names. For example, if adding the parameter **hfss** to the project, get a wget-script for the data and the reference data for parameter *hfss* from Cordex, rename the first to **wget-hffs.sh** and the second to **wget-hffs-ref.sh** and put them both under the **data/wget-scripts** directory. Then add the the following lines into the Makefile

```
raw/hfss:
    $(call, get-data,hfss)
raw_ref/hfss:
    $(call, get-data,hfss-ref)
```

Finally, from the **data** folder, run **make**.

3.2.2 Generating derived data

After downloading all required data as described in *Getting data*, but before launching the application the dowloaded data needs to be processed. There is also possible to create secondary higher order statistics such as calculating skyfalls from precipitation data or heatwaves from temperature data.

This project formats the data in into a directry tree for easier management later. This is done through the **genData.sh** shell script. If any new paramters over the baseline parameters included in the makefile is added, these must be marked explicitly for inclusion into the project file structure. This is done by editing the **genData.sh** script in the root directory.

The lines that needs to be added follows any of the following three convenient templates for preparing data.

```
createDataset <parameter-name> <ref-start-year> <ref-end-year> <start-year> <end-year>
↪ <year-averaging>
```

```
createDerivedDataset <parameter-name> <derived-parameter-name> <derived-parameter-
↪display-name>
```

```
createAggregatedDataset  <derived-parameter-name> <ref-start-year> <ref-end-year>
↪<start-year> <end-year> <year-averaging>
```

However, only the first will probably be used since the other two are only implemented for the parameters already included in the baseline data.

To include the example parameter **hffs** as downloaded as described *Getting data* and create datasets on 10 and 30 year averages, then add the following lines to the **genData.sh** script.

```
createDataset hfss 1981 2010 2011 2100 10
createDataset hfss 1981 2010 2011 2100 30
```

3.2.3 Creating data statistics

When all data is downloaded and converted to the projects internal format, statistics needs to be calculated for all regions and parameters. To create statistics for all countries and parameters, run the following from the root directory

```
python3 genStats.py
python3 AnalyzeData.py
```

3.2.4 Make the docs

If you have followed the guide for installing the tools needed for creating the documentation, open a terminal and type the following commands from the project root directory. Following these steps will generate this manual.

```
cd doc
make html
make latexpdf
```

A html page with documentation is now located in **doc/build/html/index.html** and a pdf file with documentation is now located in **doc/build/latex/ClimateSim.pdf**

CHAPTER

FOUR

CODEBASE DOCUMENTATION

This section documents the code base for this project. This should be used as a reference for developers, either for extending the code base with new functionality or fixing bugs. The code base is not completely documented and only the fundamental modules are included.

4.1 Server

This section documents the server side codebase. All code is written in python3.

4.1.1 Statistics

synopsis This module implements various functions for calculating statistics from netCDF data together with shape files for map data

statistics.**calc_country_means**(*param_data*, *index_image*, *weight_image*, *country_indices*)
Extracts the mean value of a parameter for each country

Parameters

- **param_data** (*numpy.ndarray*) – 3D-grid (time, longitude, latitude) with parameter data
- **index_image** (*numpy.ndarray*) – 2D-grid with integer indices representing countries
- **weight_image** (*numpy.ndarray*) – 2D-grid representing pixel relative sizes
- **country_indices** – dictionary on the form {country_name: country_index}

Returns Mean value for a parameter for each country

statistics.**create_index_image**(*map_data*, *lon_data*, *lat_data*)
Renders a map to an image

Parameters

- **map_data** (*dict*) – nested dicts on the format { country_name: country }
- **lon_data** (*numpy.ndarray*) – 1D-grid of longitude coordinates, assumed sorted
- **lat_data** (*numpy.ndarray*) – 1D-grid of latitude coordinates, assumed sorted

Returns Dictionary with indices on the form {country_name: country_index}

> **Warning:** If data is not interpolated then lon_data and lat_data are not 1D-grids! Use rlon and rlat variables in the netcdf file instead.

statistics.**create_lon_lat_pixel_transform**(*lon_data*, *lat_data*, *width*, *height*)
Creates a transform from longitude/latidude coordinates to pixel coordinates

Parameters

- **lon_data** (*numpy.ndarray*) – Sorted 1D-array with longitude values
- **lat_data** (*numpy.ndarray*) – Sorted 1D-array with longitude values
- **width** (*float*) – Screen width
- **height** (*float*) – Screen height

Returns 2x3 transform mapping lon-lan to pixel coordinates

statistics.**create_weight_image**(*lon_data*, *lat_data*)
Creates an image encoding the scaling factor when integrating over spherical coordinates. Useful when creating averages over lon-lat coordinates

Parameters

- **lon_data** (*numpy.ndarray*) – Sorted 1D-array with longitude values
- **lat_data** (*numpy.ndarray*) – Sorted 1D-array with longitude values

Returns 2D-array with size (len(lon_data), len(lat_data))

statistics.**getStatistics2**(*nc_root*, *europe_data*, *sweden_data*)
Calculates statistics for each country in europe and county in sweden

Parameters

- **nc_root** (*dict*) – Dictionary on the form { key: NCData }
- **europe_data** (*dict*) – Dictionary on the form { country_name : country }
- **sweden_data** (*dict*) – Dictionary on the form { area_name: area }

Returns Statistics for all countries in europe and counties in sweden

statistics.**get_country_means**(*param*, *lon*, *lat*, *map_data*)
Extracts the mean value of a parameter for each country in a map. Wraps the lower level functions calc_country_means and create_index_image

Parameters

- **param** (*numpy.ndarray*) – 3D-grid (time, longitude, latitude) with parameter data
- **lon** (*numpy.ndarray*) – Sorted 1D-array with longitude values
- **lat** (*numpy.ndarray*) – Sorted 1D-array with longitude values
- **map_data** (*dict*) – Nested dicts on the format { country_name: country }

statistics.**render_area**(*pil_draw*, *area_coords*, *transform*, *index*)
Renders an area of a country to a PIL image

Parameters

- **pil_draw** (*PIL.ImageDraw*) – PIL image draw handle
- **area_coords** (*numpy.ndarray*) – Nx3 coordinate matrix
- **transform** (*numpy.ndarray*) – 3x3 coordinate transform matrix
- **index** (*int*) – Value to draw into the image handle

Returns None

> **Warning:** pil_draw is not a PIL Image but an ImageDraw object

statistics.**render_country**(*pil_draw*, *country_data*, *transform*, *index*)
Renders a country to a PIL image

Parameters

- **pil_draw** (*PIL.ImageDraw*) – PIL image draw handle
- **country_data** (*dict*) – Nested dicts on the format { area_number: area_coordinates }
- **transform** (*numpy.ndarray*) – 3x3 coordinate transform matrix
- **index** (*int*) – Value to draw into the image handle

Returns None

statistics.**render_map**(*pil_draw*, *map_data*, *transform*, *country_indices*)
Renders countries to a PIL image

Parameters

- **pil_draw** (*PIL.ImageDraw*) – PIL image draw handle
- **map_data** (*dict*) – Nested dicts on the format { country_name: country }
- **transform** (*numpy.ndarray*) – 3x3 coordinate transform matrix
- **country_indices** (*dict*) – Dictionary mapping country names to integer values

Returns None

statistics.**weighted_mean**(*data*, *weight*)
Calculates the weighted mean of a 2D-array

Parameters

- **data** (*numpy.ndarray*) – 2D-array data image
- **weight** (*numpy.ndarray*) – 2D-array weight image

Returns Float value

4.1.2 NetCDF utilities

synopsis This module implements various utility functions for handling netCDF files.

class netcdf_utils.**NCData**(*name*, *nc_root*, *info_path=None*)
Wrapper class for netCDF files. Calculates various useful variables and make them more accessible.

class netcdf_utils.**ncopen**(*root_folder*)
Opens a file hierarchy of netCDF files.

Example usage: with ncopen_single('root_folder') as nc_root: ... Do something here

Parameters **root_folder** (*String*) – The top folder in the file hierachy

Returns Dictionary on the form { name: year: period: NCData }

class netcdf_utils.**ncopen_single**(*root_folder*)
Opens a single directorie with netCDF files.

Example usage: with ncopen_single('folder') as nc_root: ... Do something here

Parameters **root_folder** (*String*) – The top folder in the file hierachy

Returns NCData

4.2 Client

This section documents the client side codebase. All code is written in JavaScript, HTML and CSS.

4.2.1 Data Models

DataProxy

class DataProxy()
Proxy to server side objects

Arguments

- **baseUrl** (*String*) – Server side endpoint for the data
- **responseType** (*String*) – XMLHTTP response type (e.g. json, arraybuffer etc.)

DataProxy.**get**(*path*, *queries*)

Arguments

- **path** (*Array.<String>*) – Path to remote data resource
- **queries** (*KeyValuePairs*) – Optional queries to specify data

Returns **ResponseType** – The response

ParamData

class ParamData()
Wraps raw param and mask data from server

Arguments

- **data** (*ArrayBuffer*) – Raw data response from server
- **mask** (*ArrayBuffer*) – Raw mask response from server
- **meta** (*ParamMetaData*) – Meta data for parameter data

ParamDataProxy

class ParamDataProxy()
Proxy to server side parameter data

Arguments

- **meta** (*ParamMetaData*) – Parameter meta data provided by server

ParamDataProxy.**get**(*dataIndex*, *yearInterval*, *period*)

Arguments

- **dataIndex** (*Int*) – Index of time slice in remote 3D-array

- **yearInterval** (*Int*) – The year interval used in averaging the data
- **period** (*Int*) – The period for the data, can be any month (Jan, Feb, . . .), season (Winter, Spring,. . .) or year

Returns **Promise(ParamData)** – A promise of parameter data

StatData

class StatData()
Wraps raw statistics data from server

Arguments

- **json** (*JSON*) – Raw JSON response from server

StatDataProxy

class StatDataProxy()
Proxy to server side statistics data

Arguments

- **meta** (*StatMetaData*) – The meta data for statistics provided by server

StatDataProxy.**get**(*yearInterval*, *period*)

Arguments

- **yearInterval** (*Int*) – The year interval used in averaging the data
- **period** (*Int*) – The period for the data, can be any month (Jan, Feb, . . .), season (Winter, Spring,. . .) or year

Returns Promise(StatData) A promise of statistics data

This is an overview of the data model library

4.2.2 Rendering modules

DataRenderer

class DataRenderer()
Manages rendering of parameter data

Arguments

- **ShaderProgram** – program A shader program
- **ParamMetaData** – meta Parameter meta data

DataRenderer.**render**(*uniforms*)

Arguments

- **uniforms** (*Dict*) – Dictionary of uniform variables to set in shader

DataRenderer.**update**(*paramData*)

Arguments

- **paramData** (ParamData) – Updates param data to render

BorderRenderer

class BorderRenderer()
Manages rendering of country borders

Arguments

- **ShaderProgram** – program A shader program
- **MapData** – borderData Country and other area data

BorderRenderer.**renderAreas**(*uniforms*)

Arguments

- **uniforms** (*Dict*) – Renders areas of countries and regions

BorderRenderer.**renderBorders**(*uniforms*)

Arguments

- **uniforms** (*Dict*) – Dictionary of uniform variables to set in shader Renders borders of countries and areas

StatRenderer

class StatRenderer()
Manages rendering of statistics

Arguments

- **String** – chartID Classic js-style identifier for the html chart element

StatRenderer.**update**(*stat*, *area*, *interval*, *period*, *meta*)

Arguments

- **StatData** – stat The statistics data to render
- **String** – area The area to select statistics from
- **yearInterval** (*Int*) – The year interval used in averaging the data
- **period** (*Int*) – The period for the data, can be any month (Jan, Feb, . . .), season (Winter, Spring,. . .) or year

CanvasManager

class CanvasManager()
Manages the HTML canvas and WebGL context. Assumes the document has a format with three distinct parts: header, body, footer and the canvas element wrapped in a html div

Arguments

- **String** – canvasID jQuery style string for retrieving the HTML canvas
- **String** – canvasContainerID jQuery style string for retrieving the canvas container
- **String** – headerID jQuery style string for retrieving the html upper part of the document
- **String** – bodyID jQuery style string for retrieving the html middle part of the document
- **String** – footerID jQuery style string for retrieving the html lower part of the document

CanvasManager.**getContext**()

Returns **WebGLContext** – The webgl context

This is an overview of the rendering library

- genindex
- search

PYTHON MODULE INDEX

n

s

INDEX