SIMOC Documentation

Release 1.0

The SIMOC Team

Sep 24, 2023

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Basic Case

The *AgentModel* class is the main point of interaction between a user and SIMOC. Below is a very basic example of loading a configuration, initializing a model, running the model, and exporting the results.

```
import json
from agent_model import AgentModel

with open('data_files/config_1hrad.json') as f:
    config = json.load(f)

model = AgentModel.from_config(config, data_collection=True)

model.step_to(n_steps=601)

data = model.get_data(debug=True)
```

1.2 Preset Configurations

Presets include everything required for a successful simulation, including:

- All full ECLSS system comprised of 8-10 individual agents
- Enough potable water and food rations to support the crew
- An earth-normal atmosphere in the habitat and/or greenhouse

The *Config* object specifies which agents are included and at what quantity, starting currency allocations, termination conditions, step order, and other fields. The SIMOC repository includes 5 preset configurations, matching the 5 preset simulations on the web interface.

- `config_1h.json`: A small crew quarters with one human
- `config_1hrad.json`: A small crew quarters with one human and one radish
- `config_4h.json`: A medium crew quarters with four humans
- `config_4hg.json`: A medium crew quarters with four humans and a small garden.
- `config_1hg_sam.json`: The SAM habitat with 1 human and a small garden.

Presets are stored as `.json` files in the `data_files` directory. Load presets into Python, then generate a SIMOC simulation using the `AgentModel.from_config()` method.

```
with open('data_files/config_1hrad.json') as f:
    config = json.load(f)
model = AgentModel.from_config(config)
```

1.3 Custom Configurations

A best practice for custom configurations is to first load a preset, and then modify it. This assures that all required fields are included, and the atmosphere currencies are initialized correctly.

```
# Load preset
with open('data_files/config_1hrad.json') as f:
    config = json.load(f)
# Adjust agent amount
config['agents']['radish']['amount'] += 10
# Adjust starting currency allocation
config['agents']['ration_storage']['ration'] += 10
# Add a new agent
config['agents']['rice'] = dict(amount=10)
```

1.4 Visualization

`AgentModel.get_data()` returns a *Model Data*, which can be used to generate visualizations. Below are two useful functions for inspecting the data:

1.4.1 Inspect A Group

Agents' data includes up to 4 'group' fields, or fields that include several currencies or variables. These are:

- `growth`: All variables related to tracking and coordinating agent growth (e.g. `current_growth`)
- `storage`: The agent's current balance of stored currencies (e.g. `internal_biomass`)
- `flows`: The actual amount of each currency exchange (i.e. input/output)
- `deprive`: The size and status of the deprivation buffer for each relevant currency

Each group is a dict of lists, and each list is the value of a variable at every step of the simulation. Because they have a common structure, it is convenient to view them with a single function, shown below. Also included in the function are a list of variables to exclude, and an optional start/finish step.

```
import matplotlib.pyplot as plt

def plot_group(group, exclude=[], i=None, j=None):
    plt.figure(figsize=(12,6))
    length = len(next(iter(group.values())))
    i = i or 0
    j = j or length-1
    steps = range(j-i)
    for currency, values in group.items():
        if sum(values) == 0 or currency in exclude:
```

(continues on next page)

(continued from previous page)

```
            continue
        plt.plot(steps, values[i:j], label=currency)
    plt.legend()
    plt.show()

# Inspect the co2 and h2o levesl of the crew habitat
plot_group(data['crew_habitat_small']['storage'], exclude=['n2', 'o2'])
```

1.4.2 Inspect a Currency

Sometimes you want to view all flows of a particular currency across all agents. Below is a function that does this by iterating through the agents, searching for the currency, and plotting it if found.

Note that flows are recorded in two different variables in agent data:

- `flows`: the total absolute value of exchanges for a currency on a step.
- `flow_records`: the amount input and output for each exchange of a currency on a step.

Above we used `flows`, and here we'll use `flow_records` so we can plot both postive and negative values:

```
import matplotlib.pyplot as plt

def plot_currency(data, currency, exclude=[], i=None, j=None):
    flows = {}
    length = None
    for agent_name, agent_data in data.items():
        if 'flow_records' not in agent_data:
            continue
        for currency_name, currency_data in agent_data['flow_records'].items():
            if currency_name != currency:
                continue
            flow_records = {}
            if not length:
                length = len(currency_data)
            for n, step in enumerate(currency_data):
                at_least_one = False
                for record in step:
                    if agent_name not in flows:
                        flows[agent_name] = [0] * n
                    if not at_least_one:
                        at_least_one = True
                        flows[agent_name].append(-record['amount'])
                    else:
                        flows[agent_name][-1] -= record['amount']
                if not at_least_one and agent_name in flows:
                    flows[agent_name].append(0)
    plt.figure(figsize=(12,6))
    i = i or 0
    j = j or length
    steps = range(j-i)
    for agent_name, agent_data in flows.items():
        if agent_name in exclude:
```

(continues on next page)

(continued from previous page)

```
            continue
        pad_zeros = len(steps) - len(agent_data)
        if pad_zeros > 0:
            agent_data += [0] * pad_zeros
        plt.plot(steps, agent_data[i:j], label=agent_name)
    plt.legend(loc='lower right')
    plt.show()

plot_currency(data, 'o2')
```

1.5 Custom Agents

SIMOC agent descriptions are spread across five data objects. `.json` files for built-in agents are in the `data_files` directory, and are loaded automatically when initializing a model. These data objects are:

- *Currency Description*: All currencies of exchange used by agents
- *Agent Description*: Inputs, outputs and charateristics that define how the agent functions
- *Agent Connections*: Which agent(s) it is connected to for a particular currency
- *Agent Variation* (optional): Initial and step-wise variation parameters
- *Agent Events* (optional): Random events with their likelihood, duration and effects.

To add custom agents, first build custom description files for that agent, then add description files as arguments to `AgentModel.from_config()`.

```
currency_desc = {...}
agent_desc = {...}
agent_conn = {...}
model = AgentModel.from_config(config,
                               currency_desc=currency_desc,
                               agent_desc=agent_desc,
                               agent_conn=agent_conn)
```

User-defined data objects are merged with the built-ins, replacing existing fields of the same name, or adding new fields if not defined. This allows the user to add new agents and currencies by defining an agent with a new name, or modify built-in agents and currencies by mirroring the structure of the default description but modifying specific fields.

1.5.1 Defining a new agent: Mushrooms

Here we define a very basic mushroom agent. First let's describe our target behavior, then we'll integrate it into SIMOC.

- Consumes 10 grams of oxygen and 10 grams of inedible biomass per hour, following a sigmoid lifetime growth curve (i.e. mature plant consumes more)
- Grows 20g per hour (sum of oxygen and inedible biomass), following a normal lifetime growth curve (i.e. growth is fastest in the middle of lifetime)
- Live for 30 days, then harvest
- At harvest time, convert 90% of biomass to `mushrooms`, and 10% to `mushroom_waste`. Here we create a separate currency from `inedible_bimoass` (which the other plants produce) so that the mushroom can't eat itself.

The first task is to update the *Currency Description*. Oxygen, biomass and inedible biomass are already defined, so we just need to add mushrooms and mushroom waste. Make sure to nest them within the currect currency class.

```
custom_currency_desc = {
    'food': {
        'mushroom': {
            'label': 'Mushroom'
        }
    },
    'nutrients': {
        'mushroom_waste': {
            'label': 'Mushroom Waste'
        }
    }
}
```

Next, we need to add capacity for these currencies to our storage agents in the *Agent Description* data object. We do this by adding a new characteristic to the existing storages. Again, don't forget to nest the agents within the correct agent class.

```
custom_agent_desc = {
    'storage': {
        'food_storage': {
            'data': {
                'characteristics': [{
                    'type': 'capacity_mushroom',
                    'value': 1000,
                    'unit': 'kg'}]}},
        'nutrient_storage': {
            'data': {
                'characteristics': [{
                    'type': 'capacity_mushroom_waste',
                    'value': 1000,
                    'unit': 'kg'}]}}
    }
}
```

Now we'll add our new mushroom agent and define its inputs, outputs and characteristics. We can add this to the `agent_desc` object we created in the previous step.

```
custom_agent_desc['plants'] = {
    'mushroom': {
        'data': {
            'input': [{
                'type': 'o2',
                'value': 0.01,
                'required': 'desired',
                'flow_rate': dict(unit='kg', time='hour'),
                'growth': dict(lifetime={'type': 'sigmoid'}),
                'deprive':  dict(value=72, unit='hour')
            }, {
                'type': 'inedible_biomass',
                'value': 0.01,
                'required': 'desired',
```

(continues on next page)

(continued from previous page)

```
                'flow_rate': dict(unit='kg', time='hour'),
                'growth': dict(lifetime={'type': 'sigmoid'}),
                'deprive':  dict(value=72, unit='hour')
            }, {
                'type': 'biomass',
                'value': 1,
                'weighted': 'current_growth',
                'flow_rate': dict(unit='kg', time='hour'),
                'criteria': dict(name='grown', 'limit=', value=True)
            }],
            'output': [{
                'type': 'biomass',
                'value': 0.02,
                'flow_rate': dict(unit='kg', time='hour'),
                'growth': dict(lifetime={'type': 'sigmoid'}),
            }, {
                'type': 'mushroom',
                'value': 0.9,
                'weighted': 'current_growth',
                'flow_rate': dict(unit='kg', time='hour'),
                'criteria': dict(name='grown', 'limit=', value=True)
            }, {
                'type': 'mushroom_waste',
                'value': 0.1,
                'weighted': 'current_growth',
                'flow_rate': dict(unit='kg', time='hour'),
                'criteria': dict(name='grown', 'limit=', value=True)
            }],
            'characteristics': [
                dict(type='lifetime', value=720, unit='hour'),
                dict(type='growth_criteria', value='out_biomass'),
                dict(type='capacity_biomass', value=100, unit='kg')
            ]
        }
    }
}
```

Then we add connections for each currency exchanges to the *Agent Connections*.

```
custom_agent_conn = [
    dict(from='greenhouse.o2', to='mushroom.o2'),
    dict(from='nutrient_storage.inedible_biomass', to='mushroom.inedible_biomass'),
    dict(from='mushroom.biomass', to='mushroom.biomass'),
    dict(from='mushroom.mushroom', to='food_storage.mushroom'),
    dict(from='mushroom.mushroom_waste', to='nutrient_storage.mushroom_waste'),
]
```

Finally, we import a preset configuration, add some mushrooms to it, and instantiate a new model using our custom data objects:

```
with open('data_files/config_1hrad.json') as f:
    config = json.load(f)
```

(continues on next page)

(continued from previous page)

```
config['agents']['mushroom'] = {'amount': 10}
config['agents']['nutrient_storage']['inedible_biomass'] = 200
model = AgentModel.from_config(config,
                               data_collection=True,
                               currency_desc=custom_currency_desc,
                               agent_desc=custom_agent_desc,
                               connections=custom_agent_conn)
model.step_to(n_steps=1000)
data = model.get_data(debug=True)
```

CHAPTER

TWO

CLASSES

2.1 AgentModel

2.2 GeneralAgent

2.3 PlantAgent

CHAPTER

THREE

DATA OBJECTS

3.1 Config

Describes a single simulation.

```
config = {                          # * = Required, ! = supports default arg only
    'agents': {                     # *
        'rice': {
            'amount': 10,           # *
            'id': 1,
            <currency>: 10,
            'connections': [<Agent>],
            'delay_start': 0,
        },
        #...
    },
    'termination': [{
        'condition': 'time',        # !
        'value': 100,
        'unit': 'day'               # 'min', 'hour', 'day', 'year'
    }],
    'priorities': [                 # by agent class; sub-agents step randomly
        'structures', 'storage',
        'power_generation',
        'inhabitants', 'eclss',
        'plants'
    ],
    'seed': 12345,                  # Must be an integer
    'global_entropy': 0,            # 0 = no variation, 1 = max variation
    'location': 'mars'              # !
    'minutes_per_step': 60,         # !
    'single_agent': 1,              # !
}
```

3.2 Model Data

The data returned by `AgentModel.get_data()`. To return all fields use with `debug=True`. All lists have one entry for each step of the simulation.

```
data = {                                # * = default fields
    'rice': {
        'name': 'rice',                 # *
        'full_amount': 10,              # *
        'lifetime': 720,
        'reproduce': True,
        'initial_variable': 0.98765,
        'capacity': {<storage>: 100},
        'amount': [10, ...],            # *
        'age': [0, 1, ...],
        'step_variable': [1.2, 2.3, ...],
        'storage': {<storage>: [0, ...]},
        'storage_ratios': {<storage>: [1, ...]},
        'flows': {
            'in': {<currency>: {<storage>: [1, ...]}},
            'out': {<currency>: {<storage>: [1, ...]}},
        },
        'buffer': {<currency>: [8, ...]},
        'deprive': {<currency>: [720, ...]},
        'growth': {
            'total_growth': [0, ...],
            'growth': [0, ...],
            'grown': [False, ...],
            'agent_step_num': [0, 1, ...],
        },
        'events': [{
            <event>: [{
                'magnitude': 0.8,
                'duration': 10
            }]
        }],
        'event_multipliers': {<event>: [0, ...]}
    }
}
```

3.3 Currency Description

```
currency_desc = {
    'food': {                                   # Currency class
        'radish': {                             # * = Required, ^ = food only
            'label': 'Radish',                  # * Display name
            'description': 'Radishes, fresh',
            'source': <url>,                    # ^ Nutrition data reference
            'unit': 'kg',                       # ^
            'nutrition': {                      # ^ Grams per <unit>
```

(continues on next page)

(continued from previous page)

```
                "kcal": 180,
                "water": 941,
                "protein": 10,
                "carbohydrate": 25,
                "fat": 2
            }
        },
        # ...
    }
    # ...
}
```

Currency classes: `atmosphere`, `nutrients`, `food`, `water`, `energy`

3.4 Agent Description

```
agent_desc = {
    'plants': {                                 # Agent class
        'radish': {                             # Agent name
            'description': '',                  # Text description
            'data': {
                'inputs': [...<Input>],         # Currencies consumed
                'outputs': [...<Output>],       # Currencies produced
                'characteristics': [...<Char>]  # Misc params
            }
        }
        # ...
    }
    # ...
}

<Input/Output> = {                  # * = Required
    'type': 'co2',                  # * Currency name, must be in currency_desc
    'value': 0.006534,              # * Amount exchagned
    'flow_rate': {                  # * Units applied to currency exchanged
        'unit': 'kg',
        'time': 'hour'
    },
    'required': 'mandatory',        # 'mandatory' = if unavailable, skip step()
                                    # 'desired' = if unavailable, continue step()
    'deprive': {                    # If unavailable, how long to survive
        'value': 72,
        'unit': 'hour'
    }
    'growth': {                     # Map value across the hours in a day
                                    # and/or hours in agent's lifetime such
                                    # that mean hourly value is as defined.
        "lifetime": {
            "type": "sigmoid"       # 'sigmoid' = greater and end-of-lifetime
                                    # 'normal' = greater at mid-life
```

(continues on next page)

(continued from previous page)

```
        },
        "daily": {
            "type": "normal"        # 'normal' = greatest in middle of day
                                    # 'clipped' = reduced early/late values
                                    # 'switch' = boolean for 'is daylight'
        }
    },
    'requires': ['h2'],             # If input is missing, skip flow
    'weighted': 'current_growth'    # Multiply value by agent storage amount or attribute
    'criteria': {                   # Activate flow based on view of a connected agent
        'name': 'co2_ratio_in',     # '<currency>_<view>_<direction>'
        'limit': '>',               # '=', '>', '<'
        'value': 0.001,             # What the returned value is compared to
        'buffer': 2                 # Wait until valid for N steps before activating.
    }
}

<Char> = {
    'type': 'capacity_o2',      # Characteristic type
    'value': 10000,             # Supports bool, int, float or string
    'unit': 'kg'                # Optional
}
```

Agent classes: `inhabitants`, `eclss`, `plants`, `isru`, `structures`, `fabrication`, `power_generation`, `mobility`, `communication`, `storage`

Characteristic types:

- `capacity_<currency>`: The maximum amount of a particular currency that can be stored.
- `lifetime`: Length of one growth cycle
- `carbon_fixation`: 'c3' or 'c4', determines if/how plant responds to ambient co2.
- `volume`: m**3
- `mass`: kg
- `category`: sub-class, e.g. 'habitat'
- `reproduce`: boolean; whether lifecycle ends or is repeated
- `custom_function`: two are included in the SIMOC repo: `atmosphere_equalizer` and `rate_finder`.
- `threshold_lower_<currency>`: Agent is killed if ambient currency falls below

3.5 Agent Connections

Connections are directional links between agents which determine the source of inputs or destination of outputs.

The `to`/`from` fields specify an agent and currency. For the agent field, two additional options, `habitat` and `greenhouse`, are used; when a model is initialized, those options are replaced with the agent that includes the word 'habitat' or 'greenhouse' (e.g. 'greenhouse.o2' -> 'greenhouse_medium.o2')

The `priority` field is optional. If present, when the first connection (priority=0) is empty, the initiating agent will change to the second (priority=1) connection, and so on.

```
agent_conn = [{
    'from': '<agent>.<currency>',
    'to': '<agent>.<currency>',
    'priority': 0
}, ...]
```

3.6 Agent Variation

Agent variation is off by default. To activate, set the `global_entropy` parameter in `config` to a number 0 < N <= 1.

When active, all currency exchange values are scaled up or down when initialized and/or every step. Scalars are a random number from a defined probability density function. The `upper` and `lower` parameters specify the maximum absolute distance up or down from 1 (no effect).

```
agent_variation = {
    'plants': {                         # Can be agent or agent class
        'initial': {                    # Applied to values on initialization
            'upper': 0.5,               # Multiplier upper bound
            'lower': 0.5,               # Multiplier lower bound
            'distribution': 'normal'    # Probability: 'normal' or 'exponential'
        },
        'step': {
            'upper': 0.1,
            'lower': 0.1,
            'distribution': 'normal'
        }
    }
```

Alternatively, upper and lower values can be defined for each individual currency.

```
agent_variation['humans'] = {
    'initial': {
        "upper": {
            "o2": 0.045,
            # ...
        },
        "lower": {
            "o2": 0.025417,
            # ...
        },
        "distribution": "normal",
        "stdev_range": 1.65,
        "characteristics": ["mass"]
    }
    # ...
}
```

3.7 Agent Events

Agent events are off by default. To activate, set the `global_entropy` parameter in `config` to a number 0 < N <= 1.

```
agent_events = {
    "solar_pv_array_mars": [
        {
            "type": "duststorm",
            "function": "multiplier",          # 'multiplier': apply to all flows
                                               # 'termination': kill agent
            "scope": "group",                  # 'group': affects all instances
                                               # 'agent': affects a single instance
            "probability": {                   # Per group/individual based on scope
                "value": 0.0004566210046,      # Likelihood per step (if not active)
                "unit": "hour"
            },
            "magnitude": {
                "value": 1,
                "variation": {
                    "upper": 0,                # Maximum remains 1x, no effect
                    "lower": 0.9,              # Minimum is 0.1x
                    "distribution": "normal"   # Mean is 0.55x
                }
            },
            "duration": {
                "value": 24,                   # How long the effect lasts
                "unit": "hour",
                "variation": {
                    "upper": 60,               # "From 1 to 60 days"
                    "lower": 1,
                    "distribution": "exponential"  # Likely a low number
                }
            }
        },
        # ...
    ]
}
```

CHAPTER

FOUR

SIMOC DEVELOPER'S GUIDE

4.1 Setting up SIMOC

Regular users can access SIMOC at https://ngs.simoc.space.

If you want to run SIMOC on your machine or if you are a SIMOC developer, you can follow the instructions below to install SIMOC on your local machine.

Note: SIMOC is mainly developed on Linux and the setup process is optimized for Debian/Ubuntu. It is possible to run it on MacOS and Windows (using WSL), but some dependencies must be installed manually.

4.1.1 Initial Setup

These instructions explain how to set up SIMOC locally. You will need to install the SIMOC backend (`simoc`) and optionally the web frontend (`simoc-web`).

1. (Optional) If you are using a system other than Debian/Ubuntu, *install the dependencies* manually first.
2. Clone the SIMOC repositories using the following commands:

   ```
   git clone https://github.com/overthesun/simoc.git
   git clone https://github.com/overthesun/simoc-web.git  # optional
   ```

   If you are a SIMOC developer with write access, you can use these commands instead:

   ```
   git clone git@github.com:overthesun/simoc.git
   git clone git@github.com:overthesun/simoc-web.git  # optional
   ```

   The `simoc-web` repository is only needed for the web frontend.
3. (Optional) To set up the web frontend run:

   ```
   cd simoc-web
   python3 simoc-web.py setup
   ```

   At the end of the setup it will ask you about copying the `dist` directory that contains the built frontend — answer `Y`.
4. To set up the backend run:

```
# cd ..  # return to parent dir if you are in simoc-web/
cd simoc
python3 simoc.py setup
```

Note: During the setup of either the frontend or backend (steps 3/4), you might be asked to log out and log in again (or restart your machine) to complete the installation of Docker. After logging back in, you can resume the installation by running the same command again.

If you are a developer and want to contribute to the project, you will find useful information in the *Repository setup* section.

4.1.2 Installing Dependencies

SIMOC requires three dependencies:

- docker
- docker-compose
- Jinja2

On Debian/Ubuntu, the `simoc.py` script will check if any of these dependencies are missing, and install them automatically using `apt`.

On other systems you will have to install them manually.

4.1.3 Using `simoc.py`

You can run `python3 simoc.py <command>` to control SIMOC:

- Initial setup: `simoc.py setup`
- Remove SIMOC: `simoc.py teardown`
- Reinstall: `simoc.py reset`
- Start/update the containers: `simoc.py up`
- Stop/remove the containers: `simoc.py down`
- Show the help: `simoc.py --help`

Run `python3 simoc.py --help` for the full list of commands.

Running the dev backend

If you are editing the backend code and want your changes to be reflected in the Docker containers, you can mount the local directory in the container using:

```
python3 simoc.py --with-dev-backend
```

Since the directory is shared, you can keep making changes and they will automatically be visible within the container, but you might need to restart the containers using `simoc.py restart`.

Using a custom `agent_desc.json`

Note: Due to recent changes and improvements to the ways agents are handled, this command doesn't currently work. We are working on restoring this functionality.

It is also possible to specify a custom `agent_desc.json` file:

```
python3 simoc.py --agent-desc agent_desc.json reset-db
```

This will override the `agent_desc.json` file in the container with the file specified after the `--agent-desc` option. Since the file must be loaded in the DB, the `reset-db` command is used to reinitialize the DB.

If you want to return to the original version of the file, simply run:

```
python3 simoc.py reset-db
```

4.2 SIMOC's Architecture

The two main components of SIMOC are the **backend** and the **frontend**:

- The **backend** receives the parameters of the simulation, performs all the calculations, and returns the result as a list of step data.
- The **frontend** provides a convenient web interface used to visualize the results through graphs and tables.

Technically both parts are independent: the backend should be able to work without the frontend (e.g. from the command line), and the frontend should be able to work without the backend (e.g. by loading simdata from a file or by showing live data). In practice they are mostly used together.

The **backend** is written in Python, whereas the **frontend** is written in JavaScript. The source code can be found in two different repositories:

- Backend: https://github.com/overthesun/simoc
- Frontend: https://github.com/overthesun/simoc-web

4.2.1 The Backend Docker Containers

In order to run locally, the SIMOC backend uses 5 Docker containers orchestrated using `docker-compose`:

- `flask-app`
- `nginx`
- `celery-worker`
- `redis`
- `simoc-db`

They can be controlled through the `simoc.py` script, or directly through the `docker` and `docker-compose` commands.

Flask

The `flask-app` container runs a Flask application that listens to the requests and sends back responses to the frontend. The app is defined in the `simoc_server` directory.

The app also serves the frontend, that is built and included in the `simoc_server/dist` directory of the container during deployment.

Nginx

Nginx is the webserver used to forward requests and responses between the frontend and the backend.

Celery

Celery is an asynchronous task queue used to schedule simulation requests received from the frontend.

Redis

Redis is used to share data between Flask and Celery.

SIMOC DB

This container runs a MySQL instance connected to a Docker volume that currently only stores the users that created an account through the SIMOC frontend.

4.2.2 The Frontend Docker Container

The frontend is build and developed using an additional Docker container, that can be controlled using the `simoc-web.py` script.

This container resides in the frontend repo, and communicates with the backend containers.

Note that both the frontend container and the build included in the flask container can be used at the same time, since they use different `localhost` ports.

4.3 Git Guidelines

4.3.1 Repository setup

If you just want to have a local clone of the repository use:

```
$ git clone git@github.com:overthesun/simoc.git
```

If you are also planning to contribute, fork the repo on GitHub and use:

```
$ git clone git@github.com:<yourusername>/simoc.git
```

Add an `upstream` remote with:

```
$ git remote add upstream git@github.com:overthesun/simoc.git
```

By doing this you can use `origin` to refer to your fork and `upstream` to refer to the main repository. You can verify this by running:

```
$ git remote -v
origin  git@github.com:<yourusername>/simoc.git (fetch)
origin  git@github.com:<yourusername>/simoc.git (push)
upstream  git@github.com:overthesun/simoc.git (fetch)
upstream  git@github.com:overthesun/simoc.git (push)
```

4.3.2 Workflow

1. Update the `master` branch of your local copy using:

   ```
   git switch master
   git pull upstream master
   ```

2. Create a new branch from `master` using:

   ```
   git switch -c <name-of-the-branch>
   ```

3. Make changes and review, add, and *commit* them:

   ```
   git status           # check current branch and modified files
   git diff             # review changes
   git add -p           # review and add individual changes
   git commit -m '...'  # commit the changes
   ```

4. Push the changes to your fork:

   ```
   git push origin <name-of-the-branch>
   ```

5. Create a *pull request (PR)* against the upstream repo, either from the GitHub interface or by following the link that should appear after pushing.

6. Once the PR has been reviewed and *merged*, go back to step 1 to update your local clone, and delete the merged branch with:

   ```
   git branch -d <name-of-the-branch>
   ```

Note: Commits should never be created or pushed directly on `master`. A new branch and a PR should always be created.

Note: On older versions of Git, the `switch` command is not available. Use `git checkout -b <branch>` to create a new branch and `git checkout <branch>` to switch to a different branch, or update Git to a more recent version.

Pull Requests (PRs)

Each pull request should be self-contained and address a single issue (usually by either adding a feature or fixing a bug). After a PR is merged, the code should still run without errors. Incomplete and work-in-progress PRs can be marked as drafts and should not be merged, unless they are part of a *Feature branch*.

Each pull request can contain multiple commits. It is possible to add more commits to the PR by simply creating them in the branch and pushing the changes to your fork with `git push origin <branch>`.

PRs should include tests and documentation whenever possible.

Commits

Commits should also be self-contained and address a single issue, and can be used to break down an issue in smaller tasks that are tackled separately.

Commits should not include unrelated changes (e.g. reformatting/linting code other than the one that is being added/updated).

Try to limit the number of commits by using `git status`, `git diff`, and `git add -p` to review your changes carefully before adding them and creating the commit. Having a smaller number of focused commit with a clear commit message makes the history easier to navigate and understand.

Commit Messages

The commit message should be a short sentence, starting with a capitalized word and ending with a period, that describes the change being committed. It's common to use an imperative verb at the beginning of the message (e.g. `'Add <feature>.'`, `'Fix <bug>.'`, `'Improve <something>.'`).

It's also possible to leave an empty line followed by one or more paragraphs to explain the changes in detail. If the commit addresses a specific GitHub issues, adding the issue number is recommended.

Feature Branches

Feature branches can be used to develop complex features that require multiple separate pull requests. Once a feature branch is created, other branches and PRs can be created from and merged into the feature branch.

This makes it easier to develop and test the feature branch incrementally without affecting the `master` branch and without creating a giant pull request. Once the feature is completed and tested, the whole branch can be merged into `master`.

Merge strategy

In most cases PRs against `master` should be *squash merged* from the GitHub interface. A regular merge can be used in some cases, e.g. while using chained PRs against a feature branch.

4.4 SIMOC's Deployment

There are currently two available deployments of SIMOC: **NGS** and **beta**.

- **NGS** (https://ngs.simoc.space): official version, sponsored by National Geographic
- **Beta** (https://beta.simoc.space): intended for testing/experimenting

Testing on Beta is recommended before deploying on NGS.

4.4.1 How to deploy

While the deployment for both locations is automated through *GitHub Actions*, it must be triggered manually.

In order to deploy:

1. go to https://github.com/overthesun/simoc/actions (you might need privileges)
2. select the "Deploy SIMOC" workflow on the left
3. click on "Run workflow" on the right and specify the frontend and backend branches and the host from the dropdown
4. click on the green "Run workflow" button to start the deployment
5. wait until the deployment is completed and make sure that beta/NGS have been updated successfully

The branches you specify in step 3 can be:

- a named branch (such as `master`)
- a pull request, by specifying `refs/pull/`*`NUM`*`/head` (where *NUM* is the PR number)
- a specific commit ID

`master` is considered the stable branch for both the `simoc` and `simoc-web` repository, and generally only `master` should be deployed on NGS.

4.4.2 Workflow files

The current worflow and some legacy workflows are defined by files listed at https://github.com/overthesun/simoc/tree/master/.github/workflows.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search