# Computational and Systems Biology at Yale-NUS: Documentation

August 31, 2019

The following is a brief documentation of the basic functionality of the scobrapy package for the Yale-NUS Computational and Systems Biology team. Scobrapy was designed by Maurice Cheung as a supplementary package to the basic functionality of cobrapy, and thus this documentation is best read in conjunction with the cobrapy documentation.

# Contents

# 1 Getting Started

## 1.1 Installing Python and PIP using Conda

Conda is an open source, cross-platform package manager and environment management system that helps with software installation and updating dependencies.

What makes conda so powerful is that it works to install packages using dependencies already on your computer. Further, conda keeps track of your packages for you in a special folder, so it is less likely that the $PATH variable fails. Finally, conda makes uninstalling very easy in cases something goes wrong.

To install conda, navigate to "https://docs.conda.io/projects/conda/en/latest/user-guide/install/" and follow the instructions.

To install python and pip, use

```
>>>  conda install python
```

on the terminal in OSX and Linux Machines. On Windows computers, the same command can be run on Anaconda Prompt/Anaconda Powershell Prompt, which can be found from the windows search box.

### 1.1.1 Manual installation

Scobra can also be installed manually:

1. Acquire the scobra directory from `https://github.com/mauriceccy/scobra` by direct download or through **git clone https://github.com/mauriceccy/scobra**.
2. Place the directory in the location you intend to keep it in. If you place the package in your home directory, you can skip the next step.
3. Update the path variable in python using the **sys.path.append(***path/to/scobrapy***)** command.

For example, here is a possible scobra installation (note: bash prompts use $, while python prompts are denoted >>>).

```
$ cd Home/Computational_biology
$ git clone https://github.com/mauriceccy/scobra
$ python
>>> import sys
>>> sys.path.append('Home/Computational_biology')
```

Now, when you wish to use scobra in your code, import it as you would any other package:

```
>>> import scobra
>>> from scobra.io import * [import all modules from subpackage io]
```

## 1.2 Installing Python Packages using PIP

Now that python is set up and pip is installed, you can begin to install the various packages you will be using for modelling.

### 1.2.1 Setting up a Virtual Environment

We recommend installing these packages in a virtual environment, to prevent problems that may arise with projects needing different version of the same packages/modules.

To set up a conda virtual environment, run

```
$ conda create –name ENVIRONMENT_NAME
```

To activate the virtual environment, run

```
$ conda activate ENVIRONMENT_NAME
```

You can check that you are in the correct environment by running:

```
$ conda info –envs
```

The environment with an asterisk next to it is the environment you are in.

### 1.2.2 Installing requirements with pip

Once in the virtual environment, install the requirements needed by scobra using

```
$ pip install -r scobra/requirements.txt
```

This effectively clones the environment we use for software development on the users' virtual machine.

## 1.3 Updates

Scobra packages are updated on occasion with bug fixes or additional packages. In order to maintain your local copy of scobra, it is recommended that you regularly check for updates.

If you downloaded the scobra package using the automatic installation, or if you downloaded the file manually, you will need to download the updated scobra and overwrite your local version.

If you downloaded scobra using git, you can take advantage of git branch management to update your local copy of scobra in place. To do so:

1. Navigate to the scobra directory using your command line interface.
2. Use **git status** to check for updates.
3. Run **git pull** to update with the master branch.

Note: If you have made changes to the scobra directory locally, **git pull** may return a merge conflict. In this case, consult with the git documentation (`https://git-scm.com/documentation`) to resolve the merge conflict, then run the code above as normal.

## 1.4 Using Jupyter

Once you have installed IPython/Jupyter, it is useful to learn some of its basics, as it can be very powerful when used well. In particular, Jupyter has its own set of shortcuts:

1. **shift + tab** will show the docstring of an object just typed
2. **ctrl + shift + -** will split the current cell at the cursor
3. **esc + 0** toggles output
4. **%cd new/working/directory** changes the working directory
5. **!command** executes shell commands
6. **%%kernel** runs the code in that cell using a different kernel (e.g. **%%ruby** runs that cell in ruby)

It also has a command mode, which allows the speedy modification of code:

1. The **esc** key will bring you to editor mode.
2. Directional keys navigate
3. **b** will add a new cell below
4. **a** will add a new cell above
5. **dd** will delete the current cell
6. **m** will change the current cell to markup mode
7. **y** will change the current cell to code
8. **enter** will return you to edit mode

## 1.5 Using the server

While it is convenient to have a version locally installed on every user's machine, we sometimes encounter difficulties with installing scobra on some machines. This is one of the reasons we provide users access to a remote server.

Apart from that, users who want to run heavier or parallel computations can also utilise this resource.

The only drawback to working on the server is that users need to be connected to the NUS network, which they need to establish a VPN into the NUS network when working off-campus.

For instructions on how to access the server and to run Jupyter on server, please refer to Section 4.

# 2 The Scobrapy Workflow

The typical Scobrapy workflow looks like the following steps:
Load → Modify → Set Constraint → Set Objective → Solve → Analyse.

In this section, we describe the first four steps that are common to both Flux Balance Analysis (FBA) and Flux Variants Analysis (FVA). The next two sections explains how we solve and analyse FBA an FVA separately.

## 2.1 Load Model

Constraints based metabolic modelling tool 'scobra' can handle model files both in sbml and scrumpy formats. To load a model, type

```
>>> m=scobra.Model('path/to/model/file')
```

Currently, models that we use for testing is stored in *scobra/TestPrograms/sample/*.

For example:

```
>>> m = scobra.Model('./scobra/TestPrograms/sample/testmodel.xls')
```

## 2.2 Modify Model

Once, loaded, users can explore and modify the loaded model using scobra. This involves listing reactions, metabolites and genes, adding and removing them, as well as more complicated exploratory analysis such as listing the reaction a single metabolite is involved in, listing dead end metabolites, etc.

### 2.2.1 Reactions

#### List of reactions

To obtain a list of reaction IDs, type:

```
>>> m.Reactions()
['R1', 'R2', 'R3', 'R4', 'R5', 'R6', 'R7', 'R8']

>>> len(m.Reactions()) [gives count of the reactions]
8
```

### Delete reaction(s)

A single reaction can be removed using the command DelReaction. To remove multiple reactions, a list of reactions can be entered using the **DelReactions(**[*List of reaction IDs*]**)** command:

```
>>> m.DelReaction('R4')
>>> m.Reactions()
['R1', 'R2', 'R3', 'R5', 'R6', 'R7', 'R8']
```

### Gene reaction association

To obtain either the reaction associated with a gene, or the genes associated with a reaction, use **GenesToReactionsAssociations** or **ReactionsToGenesAssociations**:

```
>>> m.GenesToReactionsAssociations {'Gr6': ['R6'], 'Gr7': ['R7'],
'Gr4': ['R4'], 'Gr5': ['R5'], ···}
>>> m.ReactionsToGenesAssociations
{'R4': ['Gr4'], 'R5': ['Gr5'], 'R6': ['Gr6'], 'R7': ['Gr7'], ···}
```

### Obtain reaction names

To obtain reaction names one can use the command **GetReactionName(***Reaction ID***)**.

```
>>> m.GetReactionName('R1')
'R1'
```

```
>>> m.GetReactionNames(['R1','R2'])
['R1', 'R2']
```

### *Reaction* object

We can get reactions as a *class* object in scobra that can be utilized for several other (useful) built-in *methods*. This can provide important information about the structure/function of the model. We can obtain a *reaction* class object and assign it to a variable, for example:

```
>>> r4=m.GetReaction('Reaction ID')
>>> r4,r5 = m.GetReactions(['R4','R5'])
```

Methods which can be used on a given class can be obtained using python's built-in function **dir()**:

```
>>> r4 = m.GetReaction('R4')
>>> dir(r4)
[list of attributes for the object r4]
```

### Number of substances in a reaction - degree

We can obtain a Python dictionary containing keys for the reactions and respective values for the number of involved substances. For example:

```
>>> m.ReactionsDegree()
{'R4': 4, 'R5': 3, 'R6': 2, 'R7': 1, 'R1': 1, 'R2': 1, 'R3': 1, 'R8': 1}

>>> m.ReactionsDegree()['R5']
3
```

### Print reactions

Model reactions can be printed out using the command **PrintReaction** or **PrintReactions**.

```
>>> m.PrintReaction('R5')
R5  B + 3.0 C <=> E
```

### Reaction to sub-systems association

We can obtain the part(s) of the metabolism in which specific reactions are involved using the method **ReactionsToSubsystemsAssociations**. The reverse is obtained using the **SubsystemsToReactionsAssociations** method. This method is applicable to model object created with sbml model format (m). Presently, ScrumPy model format does not support localization of a reactions in the model file.

```
>>> m.ReactionsToSubsystemsAssociations
{'R4': ['XR4_Metabolism'], 'R5': ['XR5_Metabolism'], ···}

>>> scrumpymodel.ReactionsToSubsystemsAssociations
{'R4': [], 'R5': [], 'R6': [], 'R7': [], 'R1': [], 'R2': [], 'R3': [], 'R8': []}

>>> m.SubsystemsToReactionsAssociations [for subsystem to reaction
association]
{'XR3_Metabolism': ['R3'], 'XR7_Metabolism': ['R7'], ···}
```

### Add reactions to scobra model

Reaction(s) can be added to the scobra model using the method **AddReaction**. Arguments in the method should be passed to describe a reaction. Reversible and irreversible reaction can be described by the third argument.

For example, new reaction 'R9' is added to the model object that includes reactants E and C (stoichiometric coefficient = -1 and -2, respectively) and product F (stoichiometric coefficient = 1).

```
>>> m.AddReaction('R9',{'E':-1,'C':-2,'F':1})
>>> m.PrintReaction('R9')
R9  2 C + E −− > F [irreversible]
```

```
>>> m.DelReactions(['R9'])
>>> m.AddReaction('R9',{'E':-1,'C':-2,'F':1},True)
>>> m.PrintReaction('R9')
R9  2 C + E <=> F [reversible]
```

Multiple reactions are easily created using loops and lists with tuples.

#### Change of reaction stoichiometry

Existing reaction's stoichiometry can be changed using **ChangeReaction-Stoichiometry**. Note that this method can also update a reaction by changing reactants and/or products (with an existing set of metabolites).

```
>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':-3,'F':2})
>>> m.PrintReaction('R9')
R9  3 C + E <=> 2 F

>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':3,'F':-2})
>>> m.PrintReaction('R9')
R9  E + 2 F <=> 3 C

>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':3,'F':-2,'A':1})
>>> m.PrintReaction('R9')
R9  E + 2 F <=> A + 3 C
```

### 2.2.2 Metabolites

#### Get metabolite

To get a list of metabolites present in the scobra model use the following command:

```
>>> m.Metabolites()
['A', 'B', 'C', 'D', 'E', 'F']
```

To get metabolite class object(s) use **GetMetabolite(***Metabolite***)** or **GetMetabolites([***list of metabolites***])**.

```
>>> met=m.GetMetabolite('A')
>>> met.name
'A_internal'
```

Various parcels of information are now available regarding the metabolite:

```
>>> met.reaction(s)
frozenset([<Reaction R1 at 0xa9f8b4c>, <Reaction R4 at 0xa9f8ccc>])

>>> met.compartment
'internal'

>>> met.charge [charge]
-2

>>> metA, MetB = m.GetMetabolites(['A','B'])
```

If you only need the name, and not the object, use **GetMetaboliteName(s)**.

```
>>> m.GetMetaboliteName('A')
'A'

>>> m.GetMetaboliteNames(['A','B'])
['A', 'B']
```

### Dead, Dead End and Peripheral Metabolites

#### Dead end metabolites

Dead end metabolites in the internal stoichiometric model present in the periphery can be identified by **DeadEndMetabolites**. A dead end metabolite is a metabolite found on the periphery of a model, where flux is zero. It is identified primarily as a peripheral metabolite.

```
>>> m.DeadEndMetabolites()
['F']
```

**Dead metabolites**
Dead metabolites are those which either produced or consumed by a reaction. Using **DeadMetabolites** one can find all metabolites that are not involved in any allowed reactions.

```
>>> m.DeadMetabolites()
['F']
```

A dead end metabolite differs from a dead metabolite in so far dead end metabolites are identified by recursively deleting all peripheral metabolites until none exist, while dead metabolites are identified by a lack of flux. The distinction is clarified with the addition a successive dead reaction that utilizes 'F' as a reactant:

```
>>> m.AddReaction('R9',{'F':-1,'G':1})
>>> m.DeadMetabolites()
['G', 'F']

>>> m.DeadEndMetabolites()
['G']
```

**Peripheral metabolites**
Instructions are shown with temporary addition of reaction R9.

```
>>> m.PeripheralMetabolites("all")
['F']

>>> m.AddReactions('R9', {'F':1, 'G':-1})
>>> m.PrintReactions
R1    –> A
R2    –> B
R3    –> C
R4    A + B + C –> D
R5    B + 3.0 C <=> E
R6    E –> F
R7    D –>
R8    E –>
```

```
R9     G –> F

>>> m.PeripheralMetabolites("all")
['G', 'F']

>>> m.PeripheralMetabolites("Produced")
['F']

>>> m.PeripheralMetabolites("Consumed")
['G']

>>> m.PeripheralMetabolites("Orphan")
['G']
```

### Blocked metabolites

Metabolites not involved in allowed reactions during flux analysis.

```
>>> m.BlockedMetabolites()
['F']
```

### Metabolites involved in reactions - degree

The degree of a metabolite can be seen using the **MetaboliteDegree(s)** method:

```
>>> m.MetabolitesDegree(['B','F'])
{'B': 3, 'F': 1}

>>> m.MetabolitesDegree('B')
{'B': 3}
```

### Delete metabolites

Metabolites can be deleted using the **DelMetabolites** method:

```
>>> m.Metabolites()
['A', 'B', 'C', 'D', 'E', 'F']

>>> m.DelMetabolite('A')
>>> m.Metabolites()
['B', 'C', 'D', 'E', 'F']

>>> m.DelMetabolites(['B','C'])
>>> m.Metabolites()
['D', 'E', 'F']
```

#### Produce metabolites

Metabolites those are either produced or not produced can be obtained using **ProduceMetabolites**.

```
>>> m.ProduceMetabolites()
{'Produced': ['A', 'B', 'C', 'D', 'E', 'F'], 'Not Produced': ['G']}
```

### 2.2.3 Genes

#### Get genes

A list of genes involved in a model:

```
>>> m.Genes()
['Gr1', 'Gr2', 'Gr3', 'Gr4', 'Gr5', 'Gr6', 'Gr7', 'Gr8']
```

#### Genes to metabolism

Similar to the method for reactions, a list of gene subsystems can be obtained using **GenesToSubsystemsAssociations** or **SubsystemsToGenesAssociations**. For example,

```
>>> m.GenesToSubsystemsAssociations
{'Gr6': ['XR6_Metabolism'], 'Gr7': ['XR7_Metabolism'], ···}

>>> m.SubsystemsToGenesAssociations
{'XR3_Metabolism': ['Gr3'], 'XR7_Metabolism': ['Gr7'], ···}
```

**Get gene name**

If you only need the name of a gene and not the gene object, use the **GetGeneName** method:

```
>>> m.GetGeneName('R1')
'R1'

>>> m.GetGeneNames(['R1','R2'])
['R1', 'R2']
```

**Get gene class object**

Gene objects also exist within the model, and can be called and assigned a variable.

```
>>> g1=m.GetGene('Gr1')
>>> g1,g2=m.GetGenes(['Gr1','Gr2'])
```

### 2.2.4 Reaction and metabolite involvements

Metabolites and reaction objects are obviously associated, and these associations can be called using **InvolvedWith**:

```
>>> m.InvolvedWith('A')
{<Reaction R4 at 0xa3ebf4c>: -1.0, <Reaction R1 at 0xa3ebd4c>:
1.0}

>>> m.InvolvedWith('R1')
{<Metabolite A at 0xa3ebacc>: 1.0}
```

### 2.2.5 Neighbours in the Metabolic Graph

Neighbours are immediately adjacent reactions or metabolites; in other words, a metabolites neighbour are the metabolites which share a reaction, while a reactions neighbours are those reactions which consume its product or produce its reactants. Neighbours can be called using the **GetNeighbours** or **GetNeighboursAsDic** methods.

```
>>> m.GetNeighbours('A')
['C', 'B', 'D']

>>> m.GetNeighbours('R1')
['R4']

>>> m.GetNeighboursAsDic('R3')
{'C': ['R4', 'R5']}
```

## 2.3 Set Constraints and Objectives

Once a model is created, one can start to run simulations on it. This involve setting constraints, and maximizing/minimizing a certain objective function.

For instance, one may be interested analysing the maximum amount of sucrose a plant model produces given a restriction on oxygen. In this case the constraint would be the oxygen intake reaction, and the objective would be to *maximize* the reactions producing glucose.

Another example is to find the minimum amount of carbon dioxide, that is needed to produce a fixed amount of biomass. In this case, the constraint would be the target amount of biomass, while the objective is to *minimize* carbon dioxide.

In this subsection, we explain functions that allows us to set these constraints and objectives.

### 2.3.1 Set flux bounds to the reactions

Upper and lower limit of a reaction can be set using **SetConstraint** for a single reaction or **SetConstraints(**{*dict of reactions and flux bounds*}).

```
>>> m.SetConstraint('R1',0,10)
>>> m.SetConstraints({'R2':(0,13),'R3':(0,30)})
```

### 2.3.2 Set reaction ratios

Constraints can also be applied using the **SetReactionFixedRatio** method. This constrains the ratio of flux between two or more reactions, and can be very useful for building a set of constraints. The method accepts as an argument a dictionary where they keys are reactions and the values are the respective flux ratios. For example,

```
>>> m.SetReactionFixedRatio({ R1:1.5, R2: 2.2, R3: 0, ··· })
```

### 2.3.3 Set objective

Suppose the objective is to calculate maximum growth rate in R7 and R8, i.e., maximizing *Dex* and *Eex* production. Objective reactions (reactions to minimize or maximize) and theire maximization or minimization can be set using **SetObjective** and **SetObjDirec**, respectively.

```
>>> m.SetObjective(['R7','R8'])
>>> m.SetObjDirec("Max")
```

# 3 Flux Analysis

## 3.1 Flux Balance Analysis (FBA)

Given the constraints and objectives, scobra models can be used to simulate metabolism using flux balance analysis. For the sake of simplicity, all simulations here use the simple toy model shown in this figure:

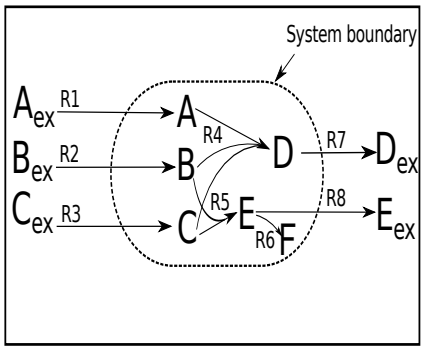


Figure 1: Simple model

### 3.1.1 Solving the Linear Programming (LP) problem

The constraints based LP problem can be solved using **Solve()**. For this example, the method returns an optimal growth rate, i.e., a feasible solution. For problems which are not possible to solve, **Solve()** will give return "infeasible":

```
>>> m.Solve()
optimal
```

### 3.1.2 Solutions

To get the flux solution as a dictionary use **GetSol**.

```
>>> m.GetSol()
{'R4': 10.0, 'R5': 3.0, 'R7': 10.0, 'R1': 10.0, 'R2': 13.0, ···}
```

### 3.1.3 Printing the solution

Flux distributions can also be obtained using **PrintSol**. For our example, the maximum growth rate (maximum possible flux under the input flux assumptions) for R7 and R8 are 10.0 and 3.0, respectively.

```
>>> m.PrintSol()
R3: 19.0
R2: 13.0
R7: 10.0
R4: 10.0
R1: 10.0
R8: 3.0
R5: 3.0
```

### 3.1.4 Fixing the flux

Flux through a reaction can be fixed to a specific value using **SetFixedFlux**.

```
>>> m.SetFixedFlux({'R1':0})
```

### 3.1.5 Get sum of fluxes

Get the sum of fluxes for a metabolite using **FluxSum(***Metabolite***)**.

```
>>> m.FluxSum('E')
3.0

>>> m.FluxSum('B')
13.0

>>> m.FluxSum('C')
19.0
```

### 3.1.6 Get constraints

In order to manipulate the behaviour of the plants, certain constraints are applied which limit the direction and extent of a specific reaction. Use **GetConstraint(s)** to view these constraints.

```
>>> m.GetConstraint('R1')
(0, 10)

>>> m.GetConstraints()
{'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), 'R6': (0.0, 1000.0), ···}
```

### 3.1.7 Summary information

The basic information of a solution is obtained using the **GetState** method:

```
>>> m.GetState()
{'objective_direction': 'maximize',
'solver': None,
'bounds': 1000.0,
'solution': <Solution 13.00 at 0xafdc7cc>,
'objective': {'R4': 0.0, 'R5': 0.0, 'R6': 0.0, 'R7': 1, 'R1': 0.0, 'R2': 0.0,
'R3': 0.0, 'R8': 1},
'quaduatic_component': None,
'constraints': {'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), ···}}
```

### 3.1.8 Get flux range

```
>>> m.AllFluxRange()
{'R4': (0.0, 10.0), 'R5': (0.0, 10.0), 'R6': (0.0, 0.0), 'R7': (0.0, 10.0),
'R1': (0.0, 10.0), 'R2': (0.0, 13.0), 'R3': (0.0, 30.0), 'R8': (0.0, 10.0)}

>>> m=scobra.Model('/home/user/toy.xml')
>>> m.AllFluxRange()
{'R4': (0.0, 1000.0), 'R5': (0.0, 333.33), 'R6': (0.0, 0.0), ···}
```

Note that the flux range is not equivalent to the constraints:

```
>>> m.GetConstraints()
{'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), 'R6': (0.0, 1000.0), ···}
```

## 3.2 Further Analysis

### 3.2.1 Flux Variability Analysis (FVA)

When scobra returns a flux solution for a given model and set of parameters, this solution is rarely unique. Rather, each reaction can sustain a variety of fluxes as the other fluxes vary. The variation of possible fluxes through a reaction can be important information regarding the importance of different reactions within a particular solution: smaller variation suggests greater importance.

In order to determine these different fluxes, we use the method **FVA**, with arguments for the reactions for which you wish to see the flux variation (as a list) and the number of processes you would like to run.

The example below returns flux variability for the first 3 reactions in the model *m*odel. It uses four processors to run the computation:

```
>>> model.FVA(model.Reactions()[0:3], processes=4)
```

The second example returns flux variability for the reactions in the list, using two processors:

```
>>> model.FVA(reaclist=['R4', 'R5'], processes=2)
```

# 4 The Server: Basics

This section will guide users on how to setup an account on the Yale-NUS computer cluster and how to navigate in the server using the command line and using Jupyter notebook.

## 4.1 Creating an account on the server

Please contact Assistant Professor Maurice Cheung for access to the server, users can contact him via email at **maurice.cheung@yale-nus.edu.sg**.

Once a user's access is granted, the user will be given a username and temporary password. To access the server, one can use

```
$ ssh username@172.25.20.52
```

Users are encouraged to change their temporary password using the command

```
$ passwd
```

## 4.2 Navigating from the command line

In order to control the server, you will need to know how to navigate from the command line interface. The server is a Linux Redhat system; as such, its commands are consistant with Some basic commands for navigation follow:

- pwd
- cd
- ls
- mkdir
- rm

## 4.3 Transferring Files to and from Remote Machine

The **scp** command can be used to copy files to and from the remote server. To copy files from the local machine into the remote machine, use

```
$ scp local/path username@172.25.20.52:remote/path
```

Similary, to copy files from remote to local, use

```
$ scp username@172.25.20.52:remote/path local/path
```

NOTE: to copy directories, replace the **scp** with **scp -r**.

To transfer files on a Windows Machine, one can also install the WinSCP program from winscp.com and use the GUI provided.

## 4.4 Group Account

Currently, there is also a group account that can be used to share code for others to look at and use. This account has the following details:

```
Username: group
Password: [on request from Maurice]
```

# 5 The Server: Advanced

## 5.1 Remote Access

In order to access the server while abroad, it is necessary to use a Virtual Private Network. To access the correct VPN, download and install the forticlient program. The remote gateway SoC VPN can be found at: *webvpn.comp.nus.edu.sg*.

## 5.2 Running local files remotely

You may desire to run local scripts remotely. Unfortunately, local scripts will not be interactive when they are run remotely. For this reason, we recommend copying the files to the remote machine before running. However, it is possible to run local files on a remote machine. The simplest method is to pipe any script on the local machine to the remote machine (replace python with the desired program, if you don't want to execute the script using python):

```
>>>  cat /path/to/file — ssh username@172.25.20.52 python -
```

## 5.3 Continuous Access Using Screen

In the event that it is necessary to run a extended calculation, you will make use of a powerful program called screen. To start screen, simply type:

```
>>>  screen
```

Screen is used to run multiple sessions over the same ssh connection. More importantly, screen allows the user to run commands while disconnected from the server. Screen has an extensive navigational syntax; to learn more:

```
press ctrl + a, then ?
```

Run processes as normal. When you wish to detach from screen:

```
press ctrl + a, then d
```

To reattach:

```
>>>  screen -r
```

To lock your session, use:

```
press ctrl + a, then x
```

To terminate your screen session:

```
press ctrl + a, then k
```

To terminate a detached screen session:

```
screen -X -S [session # you want to kill] kill
```

To list session numbers of active sessions:

```
screen -ls
```

## 5.4 Graphic User Interface

Download the GUI from https://www.realvnc.com/download/viewer/.

## 5.5 Setting up Jupyter notebook on Server

User can also use Jupyter notebook in the server using the following steps.

First, once logged into the server, activate Jupyter Notebook in the server using

```
$ jupyter notebook –no-browser –port=8889
```

This opens a Jupyter notebook session in the **remote** machine at port 8889. Since the remote machine has no display, we need to redirect the display to the local machine. We do that by running the following command **from the local machine**

```
$ ssh -N -f -L localhost:8888:localhost:8889 username@172.25.20.52
```

This command redirects whatever is happening in localhost:8889 of the remote machine to the localhost:8888 of the local machine.

Hence, to open the Jupyter notebook session in the server, simply open any browser and access

```
$ localhost:8888
```

If this is your first time connecting to the server, your browser will prompt you to provide it with a token. You can find it in the terminal connected to the remote computer.

NOTE: When many users are using the server at the same time port 8889 in the remote machine may not be available, in this case users may try displaying Jupyter notebook from other ports, e.g. you can try ports 8880-8889.

## 5.6 Git and GitHub

Git is a remote version control system used to backup projects and prevent undue risk when making individual changes to group projects. The Computational Biology git is currently being hosted on https://github.com/mauriceccy/scobra. In order to begin uploading files to the remote repository, you need to initiate a local instance of the repository, sync this repository with the remote repository and gain permission to upload to the master branch.

### 5.6.1 Setting up a local branch and syncing with the remote repository

If you did not set up your python directory using command line, it is necessary to initiate a repository on your local machine. If you installed scobra using the command line, skip step two and simply sync with the remote repository.

```
>>>  cd path/to/scobra/directory
>>>  git init
>>>  git add .
>>>  git remote add origin https://github.com/mauriceccy/scobra
>>>  git remote pull origin master
```

This will prompt a merge between your local repository and the remote repository, keeping you up to date with the master branch online.

### 5.6.2 Syncing with git

In rare cases, it may be necessary to add files created on your local machine to the remote repository. In this case, there are two options: you may either commit directly to the master branch, updating the whole git project, or you may commit to a remote branch, which can be merged with the master branch at a later time.

To create a new branch on your local machine, use

```
>>>  git checkout -b localbranchname
>>>  git remote add remotebranchname
```

Now you can push to git. To push to your own branch, replace master with your local branch name, and origin with your remote branch name.

```
>>>  git add .
>>>  git commit -m "Your commit message"
>>>  git push origin master
```