Mechanical Engineering
Industrial Process and Energy Systems Engineering

Rosmose

by

IPESE

2024-09-17

Professor: Prof. François Maréchal

Table of contents

Rosmose

ROSMOSE is a meta-language created to generate energetical optimization problems. This tool is based on the RMarkdown ecosystem, and allows users to turn their work to a powerfull report or presentation. The idea of this ecosystem is to execute computer code chunks embedded in Markdown to latter render output format such as PDF, HTML, LaTeX and so on.

Rmarkdown is a report writing tool that has been developed to make science reproducible.

Compared with conventional reporting in which data are treated as characters and figures as images, an Rmarkdown report includes in the same report document, the text, the raw data, the data processing (calculation made with the data) and the plotting instructions to generate the figures. The report, the figures and the calculation results are therefore generated on the basis of the raw data considered for your investigation.

Rmarkdown projects can have different forms like :

- A short notice report
- A website like the one you are reading here
- A book compiled using the bookdown package that is built by assembling rmarkdown files

- A scientific paper to be built using the rticles R package with pre-formated paper format from various publishers
- A blog post website.

Any Rmarkdown report can be built by assembling collection of child documents (use the rchild command). This allows to have in the same folder and in a collection of document the content of the report and all the data needed to generate it.

Rosmose engine

Rosmose technology allows users to:

- define problems
- define energy technologies
- call external energetical software
- solve optimisation problems

The idea here was to create a meta-language also called *engines* to base the computer code into energetic related jobs.

ROSMOSE is a package witten in python that can be found on `https://gitlab.epfl.ch/ipese/osmose/tools/pyxosmose` and use with reticulate in the RMarkdown ecosystem.

> NOTE : This documentation is valid for the rosmose (pyxosmose) library with the version 1.12.7

PART

I

Introduction

Installation

To use the rosmose library, you can create a RMarkdown file and load

```
```{r}
source("https://ipese-internal.epfl.ch/rscripts/rosmose-setup.R", local = knitr::knit_gl
```
```

That's it.

Basics

There is 3 main categories in **rosmose**.

1. **MODEL** that allows you to communicate with external software like (vali, aspen, ...)
2. **OSMOSE** that design the energy technology with layers, streams, units and solve the optimization
3. **TAGS** that is the concept of energetics variables (every model input or output is a tag)

Before going in deep with those categories, let's try to understand how **rosmose** works
To call **rosmose** functionalities you will need to use special chunks name

````
```{rosmose}

```
````

The first line will explain what you want to do to **ROSMOSE**. The first line looks like following:

````
```{rosmose}
: MODEL INPUTS mymodel
```
````

Here we can see 4 elements in the first line:

1. The **CHUNK MANAGER**, represented by the *:* sign
2. The **CATEGORY**, in this example, the *MODEL* category
3. The **FUNCTIONALITY**, here we tells rosmose to define *INPUTS*
4. **ARGUMENTS**, in this case *mymodel* that is the model name defined.

Chunk manager

The chunk manager tells the system what to do with the information in the chunk. It's define by a special character

char	Description
:	Execute the functionality and report the content
!	Execute the functionality without reporting it
#	Comment the entire chunk

Category

The category is defined in the second position of the first line and uppercase

Category	Description
MODEL	Tells rosmose to use external software functionalities
OSMOSE	Tells rosmose to use osmose functionalities
TAGS	Tells rosmose to use TAGS functionalities

FUNCTIONALITY and ARGUMENTS

The functionalities tells rosmose what to do with the information bellow the first line. As functionalies can need arguments, we also have arguments starting from there but not mendatory. As the functionalities depend on the category, everything will be detailed in the categories functionalities section, with the possible arguments.

PART

II

Categories functionalities and usage

Tags functionalities

TAGS is the variable concept of rosmose. As `rosmose` is a meta-language for generate energetic systems, a variable is not a simple value but an object containing at least:

- a name
- a value
- a physical unit
- a description

for example a energetic value can not be `10` but has to be

`temperature_input 10 C with the description: temperature input for heat pump.`

Create Tags

As TAGS is the main concept and calculation language in rosmose, you dont need to put any special *first line instruction* to create TAGS.

You can create **TAGS** with the structure

`tag_name = tag_value [tag_physical_unit] # tag_description`

Let's create a tag

```
```{rosmose}
my_variable1 = 2503 [C] # that's my third variable
```
```

Currently we need to create tags that are the result of a calculation.

```
```{rosmose}
my_variable2 = (20 + 10) / (100 + 2)*2
```
```

Those calculation can also be done with the value of another **TAG**. To do this, we can specify the value of the TAGS by surround the TAG name between percentage signs like: `%tag_name%`. That means that we can create more complex tags as following:

```
```{rosmose}
my_variable3 = (10 + %my_variable1% * %my_variable2% + 4 /2 *100) / 2000
```
```

Display Tags

As explained earlier, **ROSMOSE** and RMarkdown ecosystem allows user to not only execute code chunks, but also generate a report. That's why Tags also have a functionality to present **TAGS** in your report.

To display tags, we call the category `TAGS` with the functionality `DISPLAY_TAGS` and give the table value as argument. This argument is not mendatory and its written with the `[nvuc]` style. That means display the tags in a table with the columns:

- n (name)
- v (value)
- u (physical unit)
- c or d (comment/description)

```
```{rosmose}
: TAGS DISPLAY_TAGS [nvD]
```
```

It's possible that you will have a lot of **TAGS** and you only want to display only a few of them. You can do so by adding the names of the desired tags in the chunk content as following:

```
```{rosmose}
: TAGS DISPLAY_TAGS [nvD]

my_variable1
my_variable2
```
```

Save Tags

It's possible to persist tags in order to reload them without recreating them. To save tags, we call the category `TAGS` with the functionality `SAVE`

```
```{rosmose}
! TAGS SAVE
```
```

Load Tags

You can reload Tags that have been saved by calling the category `TAGS` with the functionality `LOAD` as following

```{rosmose}
! TAGS LOAD
```

Model functionalities

As explained earlier, **MODEL** is the category used to call external software. The usage of **MODEL** works as following:

1. Define the model by giving a name, a software and a path to this model
2. Define inputs and outputs.
3. Solve the model

The model will be solved with the values of the inputs defined and the outputs that you want to retrieve after the solve.

Every inputs and outputs are **TAGS** and can be reused as that.

Let's check how to create the model.

Create Model

For creating a model we dont need to add a special functionaly in the first line, only a model name. It is follow by a markdown table containing a column software, a column location and a column for comments. - The software column is needed to tell rosmose which software to use for the model name. - The location column tells the software model path - The comment column allows user to add comments.

It looks like this:

```
```{rosmose}
: MODEL myModelName

|Software|Location| Comment|
|:---|:---|:----|
|ASPEN|model/myNREL_DAP.bkp||
```
```

In this example, we create a model with the name `myModelName` that will use `ASPEN` as external software and use the model `myNREL_DAP.bkp` file that is stored in the model folder.

Available Software :

- ASPEN
- VALI

Define Model inputs

To define the model inputs, we call the category `MODEL` with the functionality `INPUTS` and give the model name as argument.

The first line is followed by a markdown table containing:

- the tag name of the model (the model value that you want to update)
- the value of the tag name that will be updated
- the physical unit of the input
- a comment to explain what is this tag

In the case of a `MODEL` using **ASPEN**, we need to give a path that define where the tag is placed in the model.

```
```{rosmose}
: MODEL INPUTS myModelName

| Name        | Path                                      |   Value | Units  | Comments |
|:------------|:------------------------------------------|--------:|:-------|:---------|
| Total_flow  | /Data/Streams/516/Input/TOTFLOW/MIXED     |   36340 | kg/hr  |          |
| Pressure    | /Data/Streams/516/Input/PRES/MIXED        |     6.1 | atm    |          |
| Temperature | /Data/Streams/516/Input/TEMP/MIXED        |     114 | C      |          |
```
```

In the case you are using another software you can just remove the path column

```
```{rosmose}
: MODEL INPUTS myNotAspenModel

| Name     |   Value | Units  | Comments |
|:---------|--------:|:-------|:---------|
| value_1  |   24846 | kg/hr  |          |
| value_2  |     204 | C      |          |
| value_3  |     1.4 | C      |          |
```
```

> NOTE: Every time you will rerun a chunk that creates input for a model, it will update the inputs with the same name and create new inputs for the none existing ones.

Define Model outputs

Outputs are the values that you want to retrieve after solving a `MODEL`. That works as the `INPUTS` but without a value column as we don't have this information.

For defining outputs, we call the category `MODEL` with the functionality `OUTPUTS` and give the model name as argument.

```{rosmose}
: MODEL OUTPUTS myModelName

| Name       | Path                                                                  |  Units
|:-----------|:----------------------------------------------------------------------|-------:
| dryfeed_in | /Data/Streams/105/Output/RES_MASSFLOW                                 |  kg/hr
| acid_in    | /Data/Blocks/A200/Data/Streams/232S/Output/MASSFLOW/MIXED/H2SO4 |  kg/hr
| w1         | /Data/Blocks/A200/Data/Streams/211/Output/MASSFLOW/MIXED/H2O    |  kg/hr
| w5         | /Data/Blocks/A200/Data/Streams/274/Output/MASSFLOW/MIXED/H2O    |  kg/hr
| w2         | /Data/Blocks/A200/Data/Streams/516/Output/MASSFLOW/MIXED/H2O    |  kg/hr
| w3         | /Data/Blocks/A200/Data/Streams/215/Output/MASSFLOW/MIXED/H2O    |  kg/hr
| w4         | /Data/Blocks/A200/Data/Streams/216/Output/MASSFLOW/MIXED/H2O    |  kg/hr
```

If you use a model different to ASPEN, you can remove the column path

```{rosmose}
: MODEL OUTPUTS myNotAspenModel

| Name     | Units  | Comments |
|:---------|:-------|:---------|
| value_4  | kg/hr  |          |
| value_5  | C      |          |
| value_6  | C      |          |
```

Define Model interfaces

The **interfaces** concept allows user to define inputs and outputs using a special language.

The inputs are define with the >> sign and outputs with the << sign. Interfaces are define as following

Inputs interfaces are defined like this

```
tag_name >> 505 [C] # comment
```

and outputs like

```
tag_name << [C] # comment
```

In the case of a model that is using **ASPEN**, you need to define inputs and outputs interfaces with the path as

```
tag_name >> /Data/Streams/516/Input/TOTFLOW/MIXED = 101 [kg/hr] # this is a comment
tag_output << /Data/Blocks/A200/Data/Streams/216/Output/MASSFLOW/MIXED/H2O [kg/hr] #
```

that can be translated as put the *tag_name* variable in the path */Data/Streams/516/Input/TOTFLOW/MIXE* with the value *101*, with the physical unit *kg/hr* and a comment *this is a comment*.

Example :

```
```{rosmose}
! MODEL INTERFACES myModelName

Total_flow >> /Data/Streams/516/Input/TOTFLOW/MIXED = 36340 [kg/hr] # that's a comment
Pressure >> /Data/Streams/516/Input/PRES/MIXED = 6.1 [atm]
Temperature >> /Data/Streams/516/Input/TEMP/MIXED = 114 [C]

dryfeed_in << /Data/Streams/105/Output/RES_MASSFLOW [kg/hr] #
acid_in << /Data/Blocks/A200/Data/Streams/232S/Output/MASSFLOW/MIXED/H2SO4 [kg/hr] #
w1 << /Data/Blocks/A200/Data/Streams/211/Output/MASSFLOW/MIXED/H2O [kg/hr] #
w5 << /Data/Blocks/A200/Data/Streams/274/Output/MASSFLOW/MIXED/H2O [kg/hr] #
w2 << /Data/Blocks/A200/Data/Streams/516/Output/MASSFLOW/MIXED/H2O [kg/hr] #
w3 << /Data/Blocks/A200/Data/Streams/215/Output/MASSFLOW/MIXED/H2O [kg/hr] #
w4 << /Data/Blocks/A200/Data/Streams/216/Output/MASSFLOW/MIXED/H2O [kg/hr] #
```
```

Reset Inputs

As said in a NOTE before, everytime you rerun a chunk that creates input for a model, it will update the inputs with the same name and create new inputs for the none existing ones. Sometimes we want to reset the inputs in order to redefine all the inputs of our model. In this case, you can use the `RESET-INPUTS` functionality for recreating new inputs after. This functionality is written as following:

```
```{rosmose}
! MODEL RESET-INPUTS myModelName
```
```

> NOTE: This functionality also exists for outputs with the `RESET-OUTPUTS` functionality calls

Display Model inputs

There is a functionality that allows you to display the inputs of a specific model. This functionality can be useful when you define inputs and outputs with the interfaces language as it is used as a programming language.

To display model inputs, we call the category `MODEL` with the functionality `DISPLAY_INPUTS` and give the model name as argument. You can give a second argument that define which value you want to display.

```
```{rosmose}
: MODEL DISPLAY_INPUTS myModelName [nvd]
```
```

Display Model outputs

There is also a **MODEL** functionality that allows you to display all the model outputs. To do this, we call the category `MODEL` with the functionality `DISPLAY_INPUTS` and give the model name as argument. As for displaying **MODEL INPUTS** You can give a second argument that define which value you want to display with the `[nvuc]` type.

```
```{rosmose}
: MODEL DISPLAY_OUTPUTS myModelName [nvd]
```
```

As this will display the model outputs, and the values will only be calculated after solving the model, this functionality is useful after the solve functionality. let's check that.

Solve Model

For solving the model, we call the category `MODEL` with the functionality `SOLVE` and give the model name as argument.

```
```{rosmose}
! MODEL SOLVE myModelName
```
```

This functionality will call the external software with the model file, the inputs data and the defined outputs data. This call will be done by calling external servers.

> [!] NOTES: those servers are in the epfl network and can only be called when you are in this network (epfl network or vpn).

Solving models can take time and it's interesting to know that, as every inputs and outputs of models are **TAGS**, you can use the *TAGS* **SAVE** and **LOAD** functionality to avoid having to restart the calculation every time.

It is also possible to solve the **MODEL** locally, of course, if the software is installed on the user computer. To use this functionality, you can use the `SOLVE-LOCAL` keyword instead of the normal `SOLVE` one.

Solve vali model locally with an input file

In some cases, the input of vali can be given or shared as a `.txt` / `.mea` file. **Rosmose** can run vali model with an input file by calling the functionality `SOLVE-LOCAL` with the following arguments:

- model name

- input file path

1. Create the model object

```
```{rosmose}
: MODEL myValiModel

|Software|Location| Comment|
|:---|:---|:----|
|VALI|model/model.bls||
```
```

2. Define the outputs if you dont want all

```
```{rosmose}
: MODEL OUTPUTS myValiModel

| Name     | Units  | Comments |
|:---------|:-------|:---------|
| value_4  | kg/hr  |          |
| value_5  | C      |          |
| value_6  | C      |          |
```
```

3. Solve the model with the input path as parameter

```
```{rosmose}
! MODEL SOLVE-LOCAL myValiModel model/my_inputs_file.txt
```
```

You can now use the outputs tags for further calculation or usage.

[!] NOTES: This functionality is at the moment only available for **VALI** with the `SOLVE-LOCAL` functionality. Also consider that the inputs are not saved as **TAGS**.

ROsmose functionalities

Create an Energy Technology (ET)

```{rosmose}
: OSMOSE ET myET

| Property            | value |
|:--------------------|:------|
| capex_weight_factor | 0.4   |
| co2_tax             | 0     |
```

NOTES: you can put tags value in the value column with the %tag_name% percentage style

Add Layers to ET

```{rosmose}
: OSMOSE LAYERS myET

| Layer       | Display name | shortname | Unit | Color  |
|:------------|:-------------|:----------|:-----|:-------|
| NATURAL_GAS | Natural Gaz  | ng        | kW   | green  |
| ELECTRICITY | Electricity  | elec      | kW   | yellow |
```

Add units to ET

```{rosmose}
: OSMOSE UNIT myET

| unit name      |type   |
|:---------------|:------|
```

```
| myProcessUnit  |Process|
| myUtilityUnit  |Utility|
```

Add parameters to units of ET

```{rosmose}
: OSMOSE UNIT_PARAM myProcessUnit

| cost1 | cost2 | cinv1 | cinv2 | imp1 | imp2 | fmin | fmax |
|:------|:------|:------|:------|:-----|:-----|:-----|:-----|
| 0     | 0     | 0     | 0     | 0    | 0    | 1    | 1    |
```

NOTES: you can put tags value every column with the %tag_name% percentage style

Add Resource Streams to Unit

```{rosmose}
: OSMOSE RESOURCE_STREAMS myProcessUnit

|layer      |direction|value|
|:----------|:--------|:----|
|NATURAL_GAS| in      |2.5  |
```

NOTES: you can put tags value in the value column with the %tag_name% percentage style

Add Heat Streams to Unit

```{rosmose}
: OSMOSE HEAT_STREAMS myProcessUnit

| name | Tin  |Tout| Hin | Hout | DT min/2 |alpha|
|:-----|:-----|:---|:----|:-----|:---------|:----|
| c1   | 10   | 11 | 0   | 200  | 1        | 1   |
| c2   | 24   | 43 | 0   | 20   | 12       | 1   |
| c3   | 25   |165 | 0   | 130  | 2        |0.5  |
```

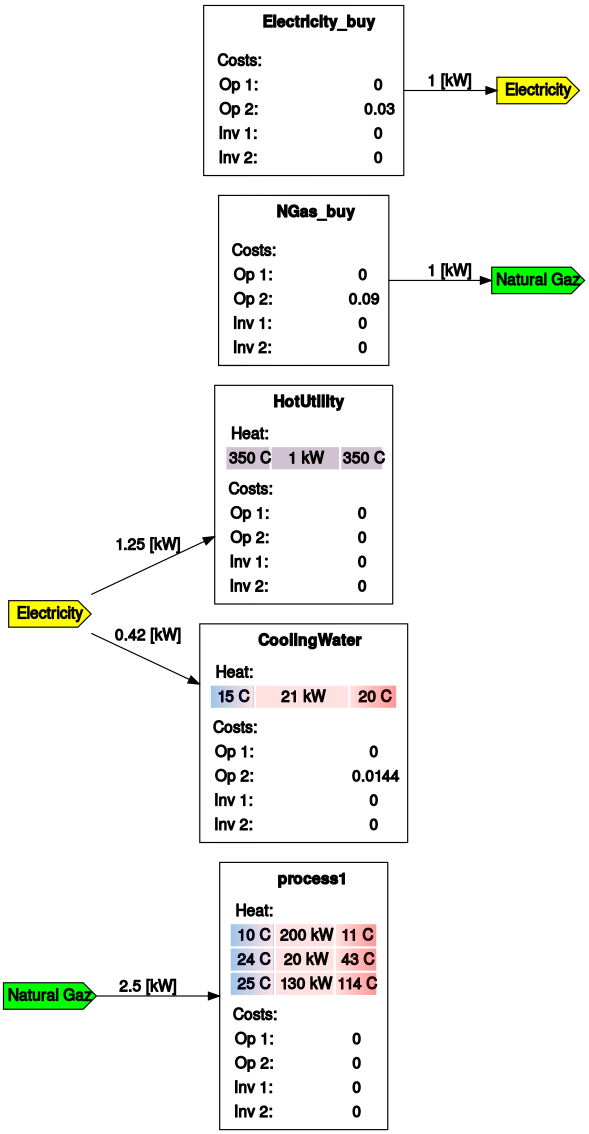
NOTES: you can put tags value in Tin, Tout, Hin, Hout, DT min/2 and alpha column with the %tag_name% percentage style

Display ET

You can create a graphical representation of every units of the Energy Technology by calling the functionality **DISPLAY_ET** of the category **OSMOSE**. This takes the name of the et as arguments.

```
```{rosmose}
: OSMOSE DISPLAY_ET myET
```
```

this function will create an svg file for displaying it in html output and pdf for displaying it in pdf file. The output will looks like this:



Superstructure Heat Pump

```{rosmose}
! OSMOSE SUPERSTRUCTURE HEATPUMP heatpump_ss
```

```
: OSMOSE FLUIDS refrigerator_ss

|Fluid        |
|:------------|
|IsoButane    |
|Methane      |
|Ethylene     |
|water        |
|Ammonia      |
|n-Propane    |
|R1234yf      |
|Propylene    |
|R32          |
|Ethane       |
|CarbonDioxide|
|R245fa       |
|R1233zd(E)   |
|R1234ze(Z)   |
|R1234ze(E)   |
|R365MFC      |
|n-Pentane    |
|Isopentane   |
|n-Butane     |
|R134a        |
|R152a        |
```

```{rosmose}
: OSMOSE TEMPERATURES heatpump_ss

|Parameter    |T1   |T2   |T3   |Unit |Comment                                    |
|:------------|:----|:----|:----|:----|:------------------------------------------|
|Temperatures |50   |40   |30   | C   |Evaporation and condensation temperatures|
|SuperheatDT  |2    |1    |2    | C   |Superheating temperature difference      |
|SubcoolingDT |2    |1    |2    | C   |Minimum temperature difference contrib   |
|CompressorDT |2    |1    |2    | C   |Superheating temperature difference      |
|DT           |2    |1    |2    | C   |Minimum temperature difference (dTmin/2) |
|MixForceUse  |1    |0    |0    | -   |Sensible heat contained                  |
```

```{rosmose}
: OSMOSE LAYER heatpump_ss

|Balance_type     |LayerOfElec  |Supercritical|
|:----------------|:------------|:------------|
|ResourceBalance  |Elec         | no          |
```

```{rosmose}
: OSMOSE COMPRESSORS_PARAMS1 heatpump_ss

|Fmin |Fmax |Inv1  |Inv2  |
|:----|:----|:-----|:-----|
|0    |100  |26143 |2172.7|
```

```{rosmose}
: OSMOSE COMPRESSORS_QUANTITY heatpump_ss

|Per_fluid |Per_model |Per_cluster |
|:---------|:---------|:-----------|
|2         |4         |4           |
```

```{rosmose}
: OSMOSE COMPRESSORS_EFFICIENCY heatpump_ss

|Efficiency|
|:---------|
|0.6       |
```

```{rosmose}
: OSMOSE COMPRESSORS_SIZING heatpump_ss

|Param   |Min     |Max     |Unit| Comment          |
|:-------|:-------|:-------|:---|:-----------------|
|size    | 7      |500     |kW  |Compressor size   |
|pressure| 1.5    | 20     |bar |Range cost linear |
|ratio   | 1.2    | 7      |-   |Pressure ratio    |
```

```{rosmose}
: OSMOSE COMPRESSORS_SELECTION heatpump_ss

select >> 0 [-] # Activate selection
linearize >> {10,100} [kW] # Linearize cost between this range
f_corr >> 1 [-] # Cost corr factor
ctype >> {1,2,5} [-] # Type of compressor: {1-liquid_ring, 2-mono_screw, 3-twin_screw, 4
mstages >> 0 [-] # Apply max number of stages per casing: 1-yes, 0-no
```

```{rosmose}
: OSMOSE HEX_PARAMS1 heatpump_ss

|Component  |Fmin  |Fmax  |Inv1  |Inv2  |
|:----------|:-----|:-----|:-----|:-----|
|Evaporator |0     |100   |     1|     2|
|Condenser  |0     |100   |     3|     4|
```

```{rosmose}
: OSMOSE HEX_PARAMS2 heatpump_ss

|Param    |Value   |Unit   | Comment                                                    |
|:--------|:-------|:------|:-----------------------------------------------------------|
|U        | 1      | W/m2K | Heat transfer coefficient                                  |
|dT       | 10     | K     | Minimum temperature difference                             |
|a        | 500    | Euro  | Cost multiplication coefficient                            |
|b        | 0.8    | -     | Cost power coefficient                                     |
|Min      | 100    | kW    | Minimum size of heat exchangers                            |
|Max      | 1000   | kW    | Maximum size of heat exchangers                            |
|force    | 0      | -     | Binary {0,1} to force the sizing of HEX                    |
|DSH      | 0.2    | -     | Percent use of desuperheating % of a condensation level    |
```

```{rosmose}
: OSMOSE VALVES_PARAMS heatpump_ss

|Min  |Max  |Unit |Comment              |
|:----|:----|:----|:--------------------|
|0.5  |20   |bard |differential pressure|
```

```{rosmose}
: OSMOSE VESSELS_PARAMS heatpump_ss
```

```
|Vessel |Fmin    |Fmax    |Comment      |
|:------|:-------|:-------|:------------|
|FlaD   |0       |100     |Flash Drum   |
|MixS   |0       |100     |Mix separator|
|DSHQ   |0       |100     |Desuperheater|
```

Superstructure Steam Network

```
```{rosmose}
! OSMOSE SUPERSTRUCTURE STEAMNETWORK steamnetwork_ss
```
```

```
```{rosmose}
: OSMOSE FLUID steamnetwork_ss

fluid >> R365mfc [-] # Fluid selected from the fluid list
```
```

```
```{rosmose}
: OSMOSE LEVELS steamnetwork_ss

|Parameter      |L1     |L2     |L3     |L4     |L5     |Unit |Comment
|:--------------|:------|:------|:------|:------|:------|:----|:-----------------------
|Pressure       |1.95632|1.00081|0.56420|0.56420|0.56420| bar |Pressure levels defined f
|layerofpressure|p1     |p2     |p3     |p4     |p5     | -   |Layer of pressure
|Temperature    |60     |40     |25     |25     |25     | C   |Temperature level, only i
|isturbine      |1      |0      |0      |0      |0      | -   |Activate turbine at the r
|issteam        |1      |0      |0      |0      |0      | -   |Activate steam generation
|superheatdT    |20     |4      |3      |2      |1      | K   |Superheating temperature
|layerofdrawoff |droffp1|droffp2|droffp3|droffp4|droffp5| -   |Layer of draw off for ste
```
```

```
```{rosmose}
: OSMOSE DT steam_network_name

|Parameter  |DT   |Unit  |Comment
|:----------|:----|:-----|:-------------------------------------------------------------
|gas        |15   |K     | Minimum temperature difference contribution related to the ga
|liquid     |5    |K     | Minimum temperature difference contribution related to the li
|phasechange|2    |K     | Minimum temperature difference contribution related to the ph
|global     |10   |K     | Minimum temperature difference contribution for the global sc
```
```

```{rosmose}
: OSMOSE HTCOEFF steamnetwork_ss

|Type           |htc         |Unit   |
|:--------------|:-----------|:------|
|gas            |0.06        |kW/m2K |
|liquid         |0.56        |kW/m2K |
|condensing     |1.6         |kW/m2K |
|vaporising     |3.6         |kW/m2K |
```

```{rosmose}
: OSMOSE EFFICIENCY steamnetwork_ss

|Efficiency     |Value     |
|:--------------|:---------|
|eff_backpr_turb|0.90      |
|eff_cond_turb  |0.95      |
|eff_pump       |0.80      |
```

```{rosmose}
: OSMOSE SIZING steamnetwork_ss

|Equipment      |Fmin |Fmax |
|:--------------|:----|:----|
|header         |0    |10000|
|drawoff        |0    |10000|
|turbine_drawoff|0    |10000|
|turbine_ext    |0    |10000|
|pump           |0    |10000|
```

```{rosmose}
: OSMOSE COST steamnetwork_ss

|Equipment  |Inv1  |Inv2  |
|:----------|:-----|:-----|
|turbine    |150   |1500  |
|pump       |50    |100   |
```

```{rosmose}
: OSMOSE PARAMS steamnetwork_ss
```

```
layerofelec >> Elec [-] # Layer of electricity
layerofheat >> DefaultHeatCascade [-] # Layer of heat
layerofmakeup >> Makeupfluid [-] # Layer of makeup fluid
layerofmakeupqual >> {1,1} [-] # Quality of the layer of makeup
subcooldT >> -1.1 [K] # Subcooling temperature difference in the condenser, must be <= 0
add_ext_turbine >> 0 [-] # Binary variable to add extraction turbine
```

```{rosmose}
: OSMOSE LAYERTYPE steamnetwork_ss

|LayerType  |BalanceType            |
|:----------|:----------------------|
|electricity|ResourceBalance        |
|pressure   |ResourceBalanceQuality |
|drawoff    |ResourceBalance        |
|condensate |ResourceBalance        |
|makeup     |ResourceBalance        |
```

Solve optimization

For solving the optimization problem, we call the category `OSMOSE` with the functionality `SOLVE` and give as arguments

- a project name
- an objective function
- a list of energy technologies

We also need to give a op_time value in a markdown table. the command we look's as following.

```{rosmose}
! OSMOSE SOLVE test-project TotalCost [myET]

| name     | default |
|:---------|:--------|
| op_time  | 5000    |
```

By solving the optimization problem, rosmose will generate osmose files and call an external server to solve the problem. This server will return a json object and be loaded as a R data.frame called `data`. That means that after the **SOLVE** you can use ***R*** chunks and get values from this data.frame as following

```
```{r}
capex <- data$results$KPIs$capex
```
```

The json return is stored in a `result` folder with a `report.Rmd` file that can be build to check the execution report of the optimization.

It is also possible to solve the osmose problem locally, of course, if osmose is installed on the user computer. To use this functionality, you can use the `SOLVE-LOCAL` keyword instead of the normal `SOLVE` one.

Solve options

```
```{rosmose}
: OSMOSE OPTIONS mathProg

| Property    | Value          | Comments |
|:------------|:---------------|:---------|
| language    | ampl           |          |
| solver      | gurobi         |          |
```
```

Generate ET Lua file from chunk

For debug/development purpose, there is a functionality that allows you to create the ET lua files without solving anything in order to control the result of the serialization. When you call the `SERIALIZE_ET` functionality you can render ET objects to lua files.

```
```{rosmose}
! OSMOSE SERIALIZE_ET [et1, et2]
```
```

This will create **et1.lua** and **et2.lua** files in the `./temp` folder.

Generate frontend Lua file from chunk

It's also possible to check the `frontend.lua` file by calling the `SERIALIZE_PROJECT` functionality. As it's necessary to create a full project object, you have to give the same arguments as you will do when you solve a project.

```
```{rosmose}
! OSMOSE SERIALIZE_PROJECT test-project TotalCost [et1, et2]

| name     | default |
|:---------|:--------|
```
```

```
| op_time  | 5000    |
```

This will create a **frontend.lua** file and an **operating_data.csv** file in the `./temp` folder

PART

III

Appendix

Calculation of thermodynamic properties in Coolprop's Python library in Rmarkdown

Coolprop library for Python in Rmd

In this course, the thermodynamic properties of the substances could be calculated either using specialized software (i.e. Aspen or DWSIM), thermodynamic tables or other libraries, such as **Coolprop**. The choice of each calculation tool is let to the students, as there is not a mandatory method, as long as the mass and energy balances are consistent.

Coolprop is a C++ library that implements pure and pseudo-pure fluid equations of state and transport properties for 122 components, with fully-featured wrappers for Python and MATLAB, among other languages. It works in 32/64-bit distributions of Windows, in Linux, OSX, Raspberry PI, etc.

Coolprop examples can be found here. More complex examples can be also found here to create web-based calculators or interactive webpages

To use Python chunks in Rmarkdown environment to calculate single thermodynamic properties of pure substances using Coolprop library, first declare the python.exe path that points to the virtual environment and activate the reticulate library of R, like this:

```
#define the path of python virtual environment to use with reticulate
path <- file.path('../venv/Scripts/python.exe') #set the python venv
library(reticulate)
use_python(path)
```

A very simple code in Python used to calculate the density D of Nitrogen at a temperature T of 298 K and a pressure P of 101325 Pa using Coolprop is shown next:

```
from CoolProp.CoolProp import PropsSI
rho = PropsSI('D', 'T', 298.15, 'P', 101325, 'Nitrogen')
```

However, even if the previous chunk could be used to determine, one by one, each thermodynamic property of a long list of substances, the verbosity ends up impairing the readability of the calculation procedure, as well as leading to syntax errors and debugging difficulties.

The State class: all the thermodynamic properties in one Python object

(a collab playground to test the State class is here)

For the sake of clarity, it has been provided a class named `State`, which can be "instantiated" by passing specific arguments to calculate a set of relevant thermodynamic properties of pure substances in just few steps. The calculated values of those properties are assigned to a friendly Python "dictionary".

The class is internally implemented; thus it just needs to be instantiated. However, in order to understand how the `State` class function works, the full code is described below. You can find the state.py file in a directory similar to `\venv\Lib\site-packages\pyxosmose` (CAUTION: do not confuse the State class with the CoolProp.State module described in here)

```
from CoolProp.CoolProp import PropsSI

class State():

    def __init__(self, pair,fluid='Water',temperature=-1.0,pressure=-1.0,density=-1.0, c
        ############ Define the relevant properties
        self.pair = pair
        self.fluid = fluid
        self.temperature = temperature
        self.pressure = pressure
        self.density = density
        self.cpmass = cpmass
        self.cvmass = cvmass
        self.enthalpy = enthalpy
        self.entropy = entropy
        self.vapfrac = vapfrac
        #self.Other = Others

    def StateCalc(self):
        if self.pair == 'TP': # Calculate only for TP
            try:
                if PropsSI('Q', 'T', self.temperature, 'P', self.pressure, self.fluid) <
                    self.vapfrac = PropsSI('Q', 'T', self.temperature, 'P', self.pressur
                    self.enthalpy = PropsSI('H', 'T', self.temperature, 'P', self.pressu
                    self.entropy = PropsSI('S', 'T', self.temperature, 'P', self.pressur
                    self.density = PropsSI('D', 'T', self.temperature, 'P', self.pressur
                    self.cpmass = PropsSI('Cpmass','T', self.temperature, 'P', self.pres
                    self.cvmass = PropsSI('Cvmass', 'T', self.temperature, 'P', self.pre
                    ########## self.Others = PropsSI('Others', 'T', self.temperature, 'P
                    ########## Other properties come here
            except ValueError:
                print('Saturation pressure and temperature are dependent. Select another
```

```
        else: # Calculate for other pair than TP but without Q in pair
            if self.pair == 'TD':
                self.pressure = PropsSI('P','T',self.temperature,'D',self.density,self.f
                self.enthalpy = PropsSI('H', 'T', self.temperature, 'D', self.density, s
            elif self.pair == 'TH':
                self.pressure = PropsSI('P','T',self.temperature,'H',self.enthalpy,self.
                self.enthalpy = PropsSI('H', 'T', self.temperature, 'H', self.enthalpy,
            elif self.pair == 'TS':
                self.pressure = PropsSI('P','T',self.temperature,'S',self.entropy,self.f
                self.enthalpy = PropsSI('H', 'T', self.temperature, 'S', self.entropy, s
            elif self.pair == 'PD':
                self.temperature = PropsSI('T', 'P', self.pressure, 'D', self.density, s
                self.enthalpy = PropsSI('H', 'P', self.pressure, 'D', self.density, self
            elif self.pair == 'PH':
                self.temperature = PropsSI('T', 'P', self.pressure, 'H', self.enthalpy,
                self.enthalpy = PropsSI('H', 'P', self.pressure, 'H', self.enthalpy, sel
            elif self.pair == 'PS':
                self.temperature = PropsSI('T', 'P', self.pressure, 'S', self.entropy, s
                self.enthalpy = PropsSI('H', 'P', self.pressure, 'S', self.entropy, self
            elif self.pair == 'DH':
                self.pressure = PropsSI('P', 'D', self.density, 'H', self.enthalpy, self
                self.temperature = PropsSI('T', 'D', self.density, 'H', self.enthalpy, s
            elif self.pair == 'DS':
                self.pressure = PropsSI('P', 'D', self.density, 'S', self.entropy, self.
                self.temperature = PropsSI('T', 'D', self.density, 'S', self.entropy, se
                self.enthalpy = PropsSI('H', 'P', self.pressure, 'S', self.entropy, self
            elif self.pair == 'HS':
                self.pressure = PropsSI('P', 'H', self.enthalpy, 'S', self.entropy, self
                self.temperature = PropsSI('T', 'H', self.enthalpy, 'S', self.entropy, s

            elif self.pair == 'TQ':  # verify that Q is in the correct range.
                if self.vapfrac < 0:
                    print('The vapor fraction cannot be negative, verify input to TQ')
                elif self.vapfrac <=1:
                    self.pressure = PropsSI('P', 'T', self.temperature, 'Q', self.vapfra
                    self.enthalpy = PropsSI('H', 'T', self.temperature, 'Q', self.vapfra
                else:
                    print('The vapor fraction must be lower or equal than 1, verify inpu

            elif self.pair == 'PQ':  # verify that Q is in the correct range.
                if self.vapfrac < 0:
                    print('The vapor fraction cannot be negative, verify input to PQ')
                elif self.vapfrac <=1:
                    self.temperature = PropsSI('T', 'P', self.pressure, 'Q', self.vapfra
```

```
                    self.enthalpy = PropsSI('H', 'P', self.pressure, 'Q', self.vapfrac,
                else:
                    print('The vapor fraction must be lower or equal than 1, verify inpu

            elif self.pair == 'DQ':  # verify that Q is in the correct range.
                if self.vapfrac < 0:
                    print('The vapor fraction cannot be negative, verify input to DQ')
                elif self.vapfrac <=1:
                    self.pressure = PropsSI('P', 'D', self.density, 'Q', self.vapfrac, s
                    self.temperature = PropsSI('T', 'D', self.density, 'Q', self.vapfrac
                    self.enthalpy = PropsSI('H', 'D', self.density, 'Q', self.vapfrac, s
                else:
                    print('The vapor fraction must be lower or equal than 1, verify inpu

            elif self.pair == 'HQ': # verify that Q is in the correct range.
                if self.vapfrac < 0:
                    print('The vapor fraction cannot be negative, verify input to HQ')
                elif self.vapfrac <=1:
                    self.pressure = PropsSI('P', 'H', self.temperature, 'Q', self.vapfra
                    self.temperature = PropsSI('T', 'H', self.temperature, 'Q', self.vap
                else:
                    print('The vapor fraction must be lower or equal than 1, verify inpu

            elif self.pair == 'SQ': # verify that Q is in the correct range.
                if self.vapfrac < 0:
                    print('The vapor fraction cannot be negative, verify input to SQ')
                elif self.vapfrac <=1:
                    self.pressure = PropsSI('P', 'S', self.entropy, 'Q', self.vapfrac, s
                    self.temperature = PropsSI('T', 'S', self.entropy, 'Q', self.vapfrac
                    self.enthalpy = PropsSI('H', 'S', self.entropy, 'Q', self.vapfrac, s
                else:
                    print('The vapor fraction must be lower or equal than 1, verify inpu

            self.entropy = PropsSI('S', 'P', self.pressure, 'H', self.enthalpy, self.flu
            self.density = PropsSI('D', 'P', self.pressure, 'H', self.enthalpy, self.flu
            self.vapfrac = PropsSI('Q', 'P', self.pressure, 'H', self.enthalpy, self.flu
            self.cpmass = PropsSI('Cpmass', 'T', self.temperature, 'P', self.pressure, s
            self.cvmass = PropsSI('Cvmass', 'T', self.temperature, 'P', self.pressure, s
```

As it can be seen, the `State` class takes few number of arguments. The first is a `pair`, i.e. two thermodynamically independent intensive properties (**tiip**) used to calculate the remaining ones. It also receives a `fluid`argument, which has a default value of 'Water'. Other arguments, such as `temperature` (K), `pressure` (Pa), 'density (kg/m3), `cpmass` (J/kgK), `cvmass` (J/kgK), `enthalpy` (J/kmol), `entropy` (J/kgK), and `vapfrac` (unitless) can be specified depending on the selected pair. Those arguments are initialized to -1 in the constructor method (`def __init__`).

Given a pair and the two **tiip** values, the remaining thermodynamic properties are calculated using the class method (`StateCalc`), which retrieves the values of all the thermodynamic properties in a Python dictionary that looks like this:

```
{'pair': 'TP', 'fluid': 'Water', 'temperature': 343.1, 'pressure': 31000, 'density': 0.1
```

Note that the vapor fraction is undefined outside the saturation region and, thus it is reported as -1 (vapor fraction falls by definition within 0 and 1). A full list of the aliases of the fluids available in Coolprop can be found here and a list of other properties not specified by the `State` class is shown here. In fact, with a little effort, any other property could be included in the 'State class (see the annotation `#self.Other = Others`) and, therefore, it could be used and reported as any other variable.

The 'instantiation' of the `State` class and the retrieval of the thermodynamic properties is as simple as executing the following code (NOTE: in the following code, `State` is a class, `State1` is a dictionary, and `Point1` is an instance/object of `State` class):

```
from pyxosmose.state import State
# First, define the thermodynamic point,
Point1 = State(pair='TP', fluid='Water', temperature=340, pressure=31000)
# Then calculate the state using the State class method StateCalc
Point1.StateCalc()
# And print the dictionary for revision
State1 = Point1.__dict__ # Whole dictionary with properties
print(State1)
```

```
{'pair': 'TP', 'fluid': 'Water', 'temperature': 340, 'pressure': 31000, 'density': 979.5
```

```
print("The enthalpy of the water is (J/kg): ", Point1.enthalpy)
```

```
The enthalpy of the water is (J/kg):  279868.6663457518
```

```
h1 = Point1.enthalpy # Point.enthalpy is equivalent to State1["enthalpy"]
```

Additional Python functions are helpful for calculating ideal mixtures of real substances based on their mass fraction:

```
from pyxosmose.state import State

def mixture(T=298, P=101325, frac_water=0.89, frac_fat=0.11):

  # call Coolprop
  state1 = State(pair='TP',fluid='Water',temperature=T,pressure=P)
  state2 = State(pair='TP',fluid='MethylLinolenate',temperature=T,pressure=P)

  state1.StateCalc()
```

```
  state2.StateCalc()

  # ideal mixture
  state_mix = {}
  state_mix['fluid'] = 'water[' + str(frac_water) + ']&fat[' + str(frac_fat) + ']'

  for key in ['density', 'cpmass', 'cvmass', 'enthalpy', 'entropy']:
    state_mix[key] = frac_water * getattr(state1, key) + frac_fat * getattr(state2, key)
  return state_mix
```

An example of using the `mixture` function is shown below. Note that it produces not an object of `State` class, but a dictionary (`State2`) with the same keys as the attributes of the `State` class:

```
from codes_01_energy_bill.coolprop_functions import mixture

State2=mixture(frac_water=0.8, frac_fat=0.2) # this is a dictionary!!
print(State2)
```

```
{'fluid': 'water[0.8]&fat[0.2]', 'density': 976.7940177228429, 'cpmass': 3768.5468846031
```

```
h2 = State2["enthalpy"]
```

Saving data into and retrieving data from a JSON file

The data gathered throughout all the calculations could be dumped into a JSON file, called e.g. `MyJsonFile.json`, using the `save_states` function. Data is stored in JSON format so that it could be used in other parts of the report. Data in the JSON file can be retrieved using other function called `load_states`:

```
import json
def save_states(states, json_name='MyJsonFileDefault'): #MyJsonFileDefault is just a def
  with open('codes_01_energy_bill/results/' + json_name + '.json', 'w') as f:
    json.dump(states, f)
  return

def load_states(path='codes_01_energy_bill/results/MyJsonFileDefault.json'):
  with open(path, 'r') as f:
    data = json.loads(f.read())
  return data
```

An example of utilization of these functions is shown below:
Saving data into the JSON

```
MyStates_list = {"MyState1": Point1.__dict__, "MyState2": State2, "enthalpy1": h1, "enth
save_states(MyStates_list, 'MyJsonFile')
```

Retrieving data from a JSON file:

```
data = load_states('codes_01_energy_bill/results/MyJsonFile.json')
print(data) # it is a dictionary of dictionaries and other key:value objects
```

```
{'MyState1': {'pair': 'TP', 'fluid': 'Water', 'temperature': 340, 'pressure': 31000, 'de
```

Using State function together with other Python libraries for solving systems of equations

The power of Python and Coolprop could be combined with other programming libraries such as `numpy` and `scipy` to solve systems of equations, which can in practice be mass and energy balances (NOTE: verify if `scipy` package is installed before using this chunk, see the last section of this tutorial to install new Python libraries in the virtual environment):

```
# Example of how to use the State class for calculating properties and solving systems o

import numpy
import scipy
from scipy.optimize import fsolve
from pyxosmose.state import State

mystate1 = State(pair='TP',fluid='air',temperature=298,pressure=101325)
mystate1.StateCalc()

#print(mystate1.__dict__)
h1 = mystate1.enthalpy
s1 = mystate1.entropy
#print('Enthalpy of state 1: ', h1, 'Entropy of state 1: ', s1)
P2s = 300000
mystate2s = State(pair='PS',fluid='air',pressure=P2s,entropy=s1)
mystate2s.StateCalc()
#print(mystate2s.__dict__)
eff = 0.8
#print('Enthalpy of state 2s: ', mystate2s.enthalpy)


def equations(vars):
    h2s, h2a = vars

    #equations
```

```
    eq1 = h2s-h1-eff*(h2a-h1)
    eq2 = h2s-mystate2s.enthalpy
    return eq1, eq2


seeds = numpy.array([100000, 100000])
[h2s, h2a] = fsolve(equations, seeds)


print('These are the solutions',[h2s, h2a])
```

```
These are the solutions [533127.757776997, 560338.4226996723]
```

```
print('Checking residuals:',equations([h2s, h2a]))
```

```
Checking residuals: (0.0, 0.0)
```

```
print('This is h1: ',h1, ' and this is h2s: ',h2s)
```

```
This is h1:  424285.09808629606  and this is h2s:  533127.757776997
```

This is another example of how to use directly Coolprop for calculating properties and solving systems of linear equations

```
from CoolProp.CoolProp import PropsSI

def equations(vars):
    #variables
    h1, s1, h2a, h2s = vars # the variables are pre-declared here, meaning that they com
    #parameters
    P1 = 101325 #bar
    T1 = 298 #K
    P2s = 300000 #bar
    eff = 0.8
    #equations
    eq1 = h1 - PropsSI('H', 'T', T1, 'P', P1, 'air')
    eq2 = s1 - PropsSI('S', 'T', T1, 'P', P1, 'air')
    eq3 = h2s - PropsSI('H', 'P', P2s, 'S', s1, 'air')
    eq4 = h2s - h1 - eff * (h2a - h1)

    return eq1, eq2, eq3, eq4

    seeds = numpy.array([0,0,0,0])
    h1, s1, h2a, h2s = fsolve(equations, seeds)
```

```
print('This are the solutions: ', [h1, s1, h2a, h2s])
```

```
This are the solutions:  [424285.09808629606, 3879.982762572455, 560338.4226996723, 5331
```

```
print('Checking residuals:', equations([h1, s1, h2a, h2s]))
```

```
Checking residuals: (0.0, 0.0, 0.0, 0.0)
```

Citation of Coolprop software

Naturally, Coolprop is the result of the intellectual effort of researchers and programmers, and thus it needs to be cited properly if used. The bibtex citation for Coolprop suggested in its website is:

```
@article{doi:10.1021/ie4033999,
    author = {Bell, Ian H. and Wronski, Jorrit and Quoilin, Sylvain and Lemort, Vincent}
    title = {Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and
             the Open-Source Thermophysical Property Library CoolProp},
    journal = {Industrial \& Engineering Chemistry Research},
    volume = {53},
    number = {6},
    pages = {2498--2508},
    year = {2014},
    doi = {10.1021/ie4033999},
    URL = {http://pubs.acs.org/doi/abs/10.1021/ie4033999},
    eprint = {http://pubs.acs.org/doi/pdf/10.1021/ie4033999}
    }
```

Other examples using Python in Rmarkdown's Reticulate

Python and R objects can be used interchangeably thanks to an R library called Reticulate:
An R variable

```
#library(reticulate)
x = 42
print(x)
```

```
[1] 42
```

In the following chunk, the value of x on the right hand side is 42, which was defined in the previous chunk.

```
x = x + 12
print(x)
```

```
[1] 54
```

This works fine and as expected.

```
x = 42 * 2
print(x)
```

```
84
```

The value of `x` in the Python session is 84. It is not the same `x` as the one in R.

```
x = x + 18
print(x)
```

```
102
```

Retrieve the value of `x` from the Python session again:

```
py$x
```

```
[1] 102
```

Assign to a variable in the Python session from R:

```
py$y = 1:5
```

See the value of `y` in the Python session:

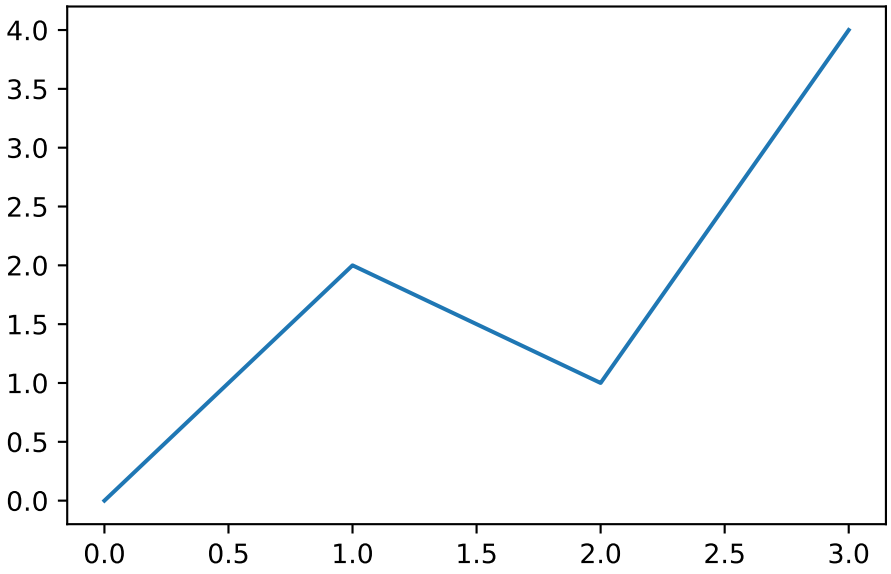
```
print(y)
```

```
[1, 2, 3, 4, 5]
```

Graphical representations using Python

You can draw plots using the **matplotlib** library in Python (NOTE: if you get an error using MatPlotLib, check this out).

```
import matplotlib.pyplot as plt
plt.plot([0, 2, 1, 4])
plt.show()
```



As it can be seen, Python is not only used to retrieve thermodynamic properties using the Coolprop library, but could be also used to plot more elaborated plots, such as the Composite Curves (CC) of the studied systems.

```
import matplotlib.pyplot as plt
import json
import numpy as np

# read json for heat integration data
file_name = open('codes_02_heat_recovery/result/brewery-1.json')
integ = json.loads(file_name.read())

# we used .keys() and json-reader to locate the data of interest (hot and cold composite
hotcc = integ["results"]["graph"][0][0][1]["data"][0]["curve"]
coldcc = integ["results"]["graph"][0][0][1]["data"][1]["curve"]

# extract the temperature (T in Kelvin) and heat (Q in kW)
T_hc = [i["T"] for i in hotcc]
Q_hc = [i["Q"] for i in hotcc]
T_cc = [i["T"] for i in coldcc]
Q_cc = [i["Q"] for i in coldcc]

# compute the shifted carnot factor (1 - T0/T) and Treference = 298.15K
Tref = 298.15
ca_hc = [(1-Tref/i) for i in T_hc]
ca_cc = [(1-Tref/i) for i in T_cc]
```

```
# compute back the original carnot factors given a deltaT_min
deltaT_min = 40
T_hc_ori = [(i+deltaT_min/2) for i in T_hc]
T_cc_ori = [(i-deltaT_min/2) for i in T_cc]
ca_hc_ori = [(1-298.15/i) for i in T_hc_ori]
ca_cc_ori = [(1-298.15/i) for i in T_cc_ori]

# prepare the second y-axis for plotting
# the minim carnot value is extracted from the original cooling. Round it to the lowest
# the maximum carnot carnot value is extracted from the original heating. Round it to th
ca_min = round(min(ca_cc_ori) - 0.05, 1)
ca_max = round(max(ca_hc_ori) + 0.05, 1)
# generate an array with the carnot list from min to max
ca_values = np.arange(ca_min, ca_max, 0.2)
# compute the corresponding temperature values in celcius
Tref_ce = 25
T_ce = [int(Tref_ce/(1-i)) for i in ca_values]

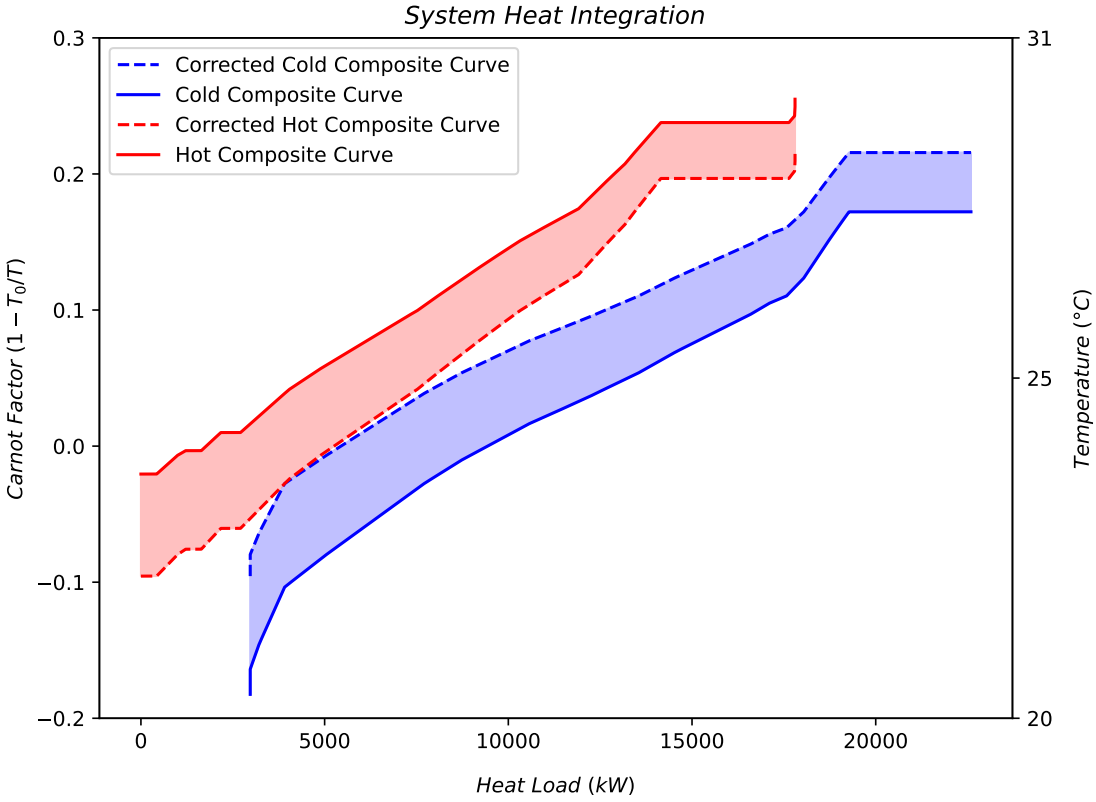
# plot the composite curves
figure, ax = plt.subplots(figsize = (8,6))

# both cold curves (blue): dashed --> shifted, solid --> original
ax.plot(Q_cc, ca_cc, "--b")
ax.plot(Q_cc, ca_cc_ori, "-b")
# both hot curves (red): dashed --> shifted, solid --> original
ax.plot(Q_hc, ca_hc, "--r")
ax.plot(Q_hc, ca_hc_ori, "-r")
# fill the area between the original and shifted curves
ax.fill_between(Q_cc, ca_cc_ori, ca_cc, color="blue", alpha=0.25)
ax.fill_between(Q_hc, ca_hc, ca_hc_ori, color="red", alpha=0.25)

# add the lables and title
ax.set_xlabel('$Heat \ Load \ (kW)$', labelpad=12)
ax.set_ylabel('$Carnot \ Factor \ (1 - T_{0}/T)$')
ax.set_title('$System \ Heat \ Integration$')
ax.legend(["Corrected Cold Composite Curve", "Cold Composite Curve", "Corrected Hot Comp
ax.set_ybound(ca_min, ca_max)

# secondary axis. In this case we don't plot anything, we simply set the secondary axis
ax2 =ax.twinx()
ax2.set_yticks(np.arange(0, len(T_ce), 1), T_ce)
ax2.set_ylabel('$Temperature \ (\u00b0C)$', labelpad=10)

plt.show()
```




```
# save the plot as a png
figure.savefig("composite_curves.png", format="png", dpi=600)
```

How to install libraries that are not yet installed in the Python virtual environment of Rmarkdown

Lastly, if Coolprop or any other library is not installed yet or needs to be updated, use the following command to avoid installation outside of the venv, i.e. to force the installation in the target directory. First, open a cmd console in the Scripts folder, where is the pip.exe. Then type the following command pointing to the `lib/site-packages` of the respective venv:

`pip install scipy -t "D:\brewery_process-main\venv\Lib"` (use your path)

This command should install the missing Python libraries in the targeted folder of the virtual environment

Logical and mathematical operators in Rosmose language

Similar to the OSMOSE Lua language, in Rosmose language, it can be also used different mathematical and logical operators:

- Sum: %foo%+2
- Substraction: %bar%-2
- Multiplication: %foo%*2
- Division: %bar%*2
- Potentiation: %foo%**%bar%

Special mathematical functions:

- exponential: eˆx = 2.7182818**%x%
- logarithm:
- sin:
- cosine:
- tan:
- sinh:
- square root: %foo%**0.5
- n root of m: %m%**%n%

Logical and boolean expressions:

- If/Else:
- For:
- Greater than:
- Lower than:
- And/Or/Not:

Transference of variables from:

- python to r: .
- r to python: $
- r to rosmose:
- romose to r:
- python to rosmose:
- rosmose to python:

Syntax errors to be avoided

When using Rosmsose language, avoid doing the following syntax errors:

- Do not use INPUTS functionality, unless it is an ASPEN model. Common inputs do not require any declaration, and it throws error:

```
```{rosmose}
MODEL INPUTS myModel
fooTag = 2 [unit] #comment
...
...
```
```

- NEVER change the names of the JSON files generated in the folder result. NEVER. For example, never put suffix like hps, furn, vend, etc. after the numbering of the files and the .json extension:

tixothermop-1hps.json tixothermop-2furn.json tixothermop-3vend.json

- Do not use # or leave spaces at the beginning of a chunk:

```
```{rosmose}
# This is a bad comment
...
...
```
```

- Be careful using tags and layer names:

Do not put spaces or dashes in the tags, only use (_). However, _ should be also avoided in the layer names

- Do not assign one tag to another tag

For instance `tag1=%tag2%` may seem helpful when willing to use the tag in other part of the code with a more easy to remember name. If for any reason this is necessary, multiply the tag by 1, i.e. `tag1=1*%tag2%`

- If using MER as objective function, it is required to solve Rosmose only loading the Process Models in the SOLVE execution.
- When defining the UNIT field, be aware that this defines the Units of the ET, thus it needs to be defined as `OSMOSE UNIT MyET`, instead of `OSMOSE UNIT MyUnit`

Common errors

When using Rosmsose language, avoid doing the following logical errors:

- **Infeasible problem**: In this case, the utilities defined in the problem cannot satisfy the demand of the process units. It could happen if they are restricted in terms of maximum size or temperature level. To tackle this issue, one must recall the process needs and make sure suitable utilities in the problem definition are provided.
- **AMPL connection not established**: In case the AMPL license is not correctly configured on the computer, ROSMOSE will not be able to connect to the specified solver. To solve this error, correct the path mapping in the environmental variables definition in the computer configuration settings so that it can adjust the AMPL solver directory.
- **Expected value near symbol %**: In this case, one of the tags has not been correctly defined in the Quarto environment. This is either due to a typo or importing data from the wrong location (path) when connected to external flowsheeting software such as Aspen Plus. To eliminate this error, revise the definition of tags throughout your problem formulation.
- **No file found with this name or location**: This occurs when a file path or name is not defined correctly in the fronted file. To address this issue, check your repository information (directory location, file names, access rights for Quarto, etc.)
- **Unexpected trend or shape of composite curve**: When defining heat streams, it is important to respect the thermodynamic rules of temperature levels and heat duty. A negative heat flow must correspond to a decrease in temperature and vice versa. Otherwise, OSMOSE Lua cannot arrive at the correct heat integration results or returns error messages stating incorrect problem definition.
- **Utility units never activated, or masses not transported correctly**: As ROSMOSE relies on layers for exchanging mass and energy flows, the correct definition of layers names and units is crucial for the seamless flow of streams through these "pipes". In case there is discrepancy between layers defined in different ETs, then certain flows will never be balanced within the units, and it hinders the optimization process due to infeasible solutions.
- **Activate the serialization feature**: In case you face a persisting error, that does not fit into the description of any of the above cases, you can use the `: OSMOSE SERIALIZE` functionality to retrieve the backend Lua files describing the project and ET files. This offers an opportunity for debugging deeper errors conveyed from ROSMOSE to the platform solving the optimization problem.