Mechanical Engineering
Industrial Process and
Energy Systems Engineering

Rosmose

by

Michel Lopez

updated on 2022-09-08

Professor: Prof. François Maréchal

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/osmose/rosmose-setup.R", local = knitr::knit_glob
```
```

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/MIX* 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**.

Osmose 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  |
```

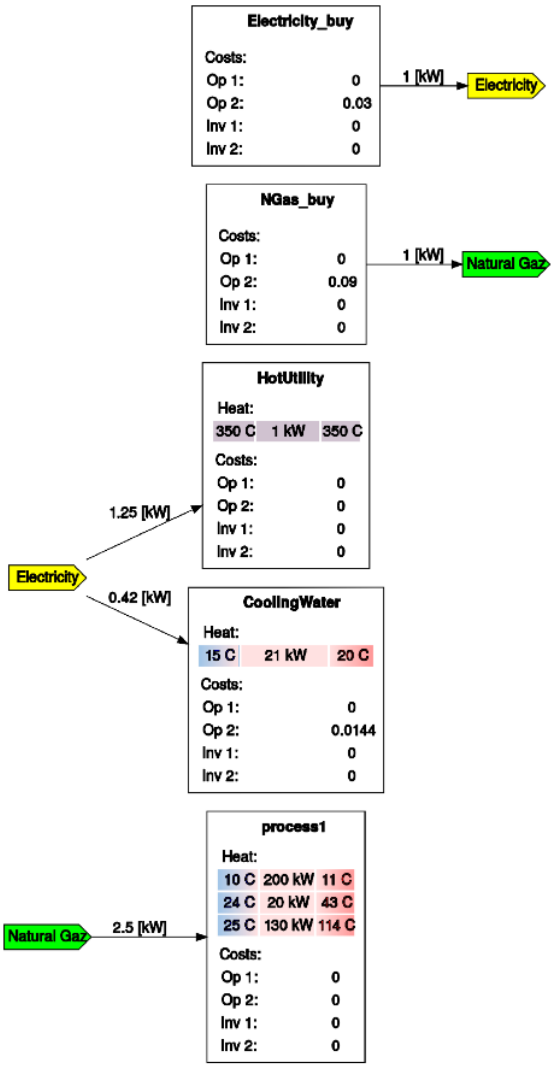
NOTES: you can put tags value in Tin, Tout, Hin, Hout, DT min/2 and alpha column with the %tag_name% percentage style

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:



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.

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.

````
```{r}`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.

````
```{r}`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

Bibliography search

Bibiliography Data Management

Researchers typically collect information in literature. This proceeds by different steps :

1. use your prefered search engine (typically `http://scholar.google.com`) and define search keywords.
2. Once you have identified a paper of interest, you add it in your Zotero library. Zotero is a bibliography management system that allows you to organise your citations
3. For your research you will then create a sub-collection with the list of papers you have identified.

In IPESE, we have a zotero group that you can access if your are in EPFL/IPESE. In this case the papers you read and comment are also shared with your colleagues.

How to get the data in the bibliography in a Rmarkdown project

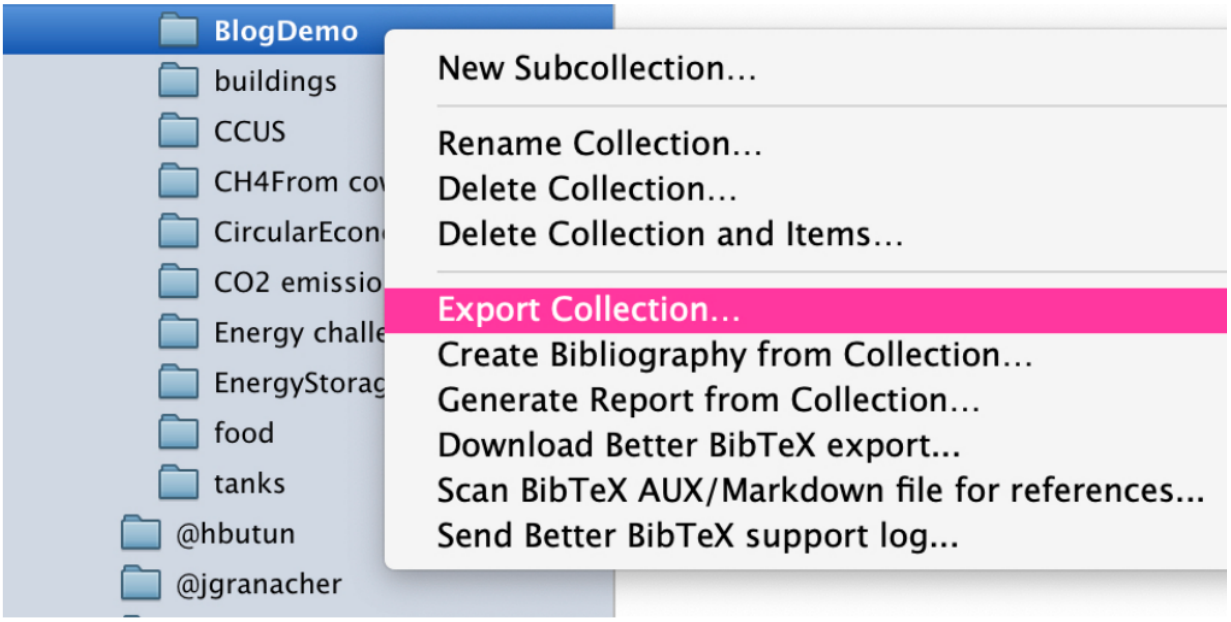
Rosmose contains a tool that can be used in rmakdown but also in Jupyter to collect the useful information in the bibliography you are generating.

Step 1 : create a sub collection and export it as a bib file

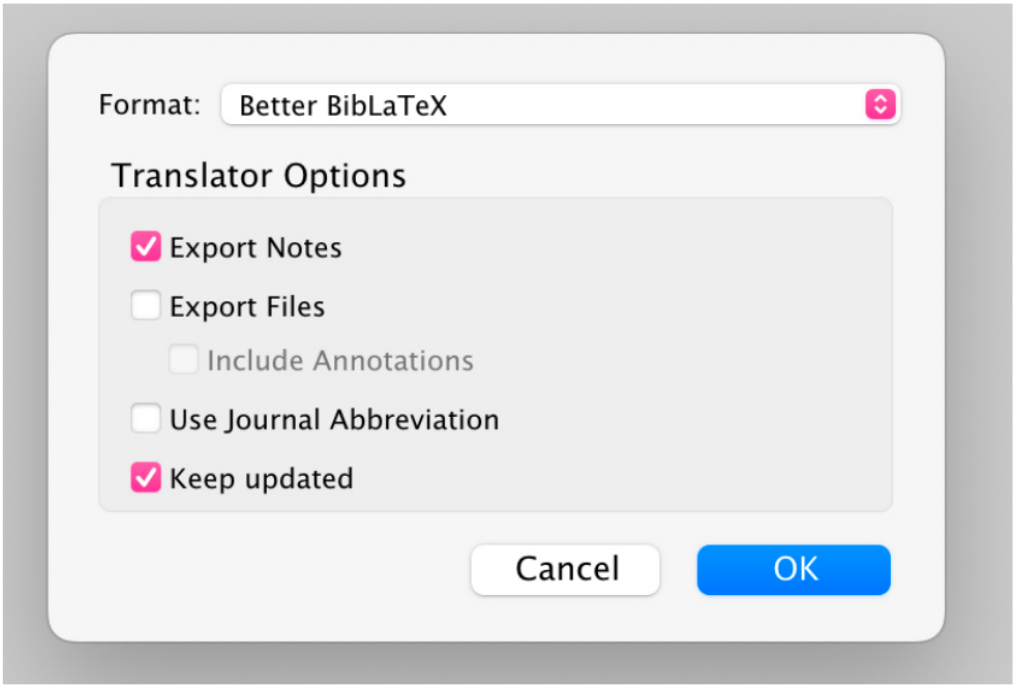
The first step is to make a bibliography search looking to get some values in the literature. In the IPESE group Zotero, we create a sub-collection for your rmarkdown project. In Zotero, the next step is to create an automatic export of the sub-collection in the folder where you write your report. For the purpose of this post I have created a file `BlogDemo.bib` in the same folder as this post.

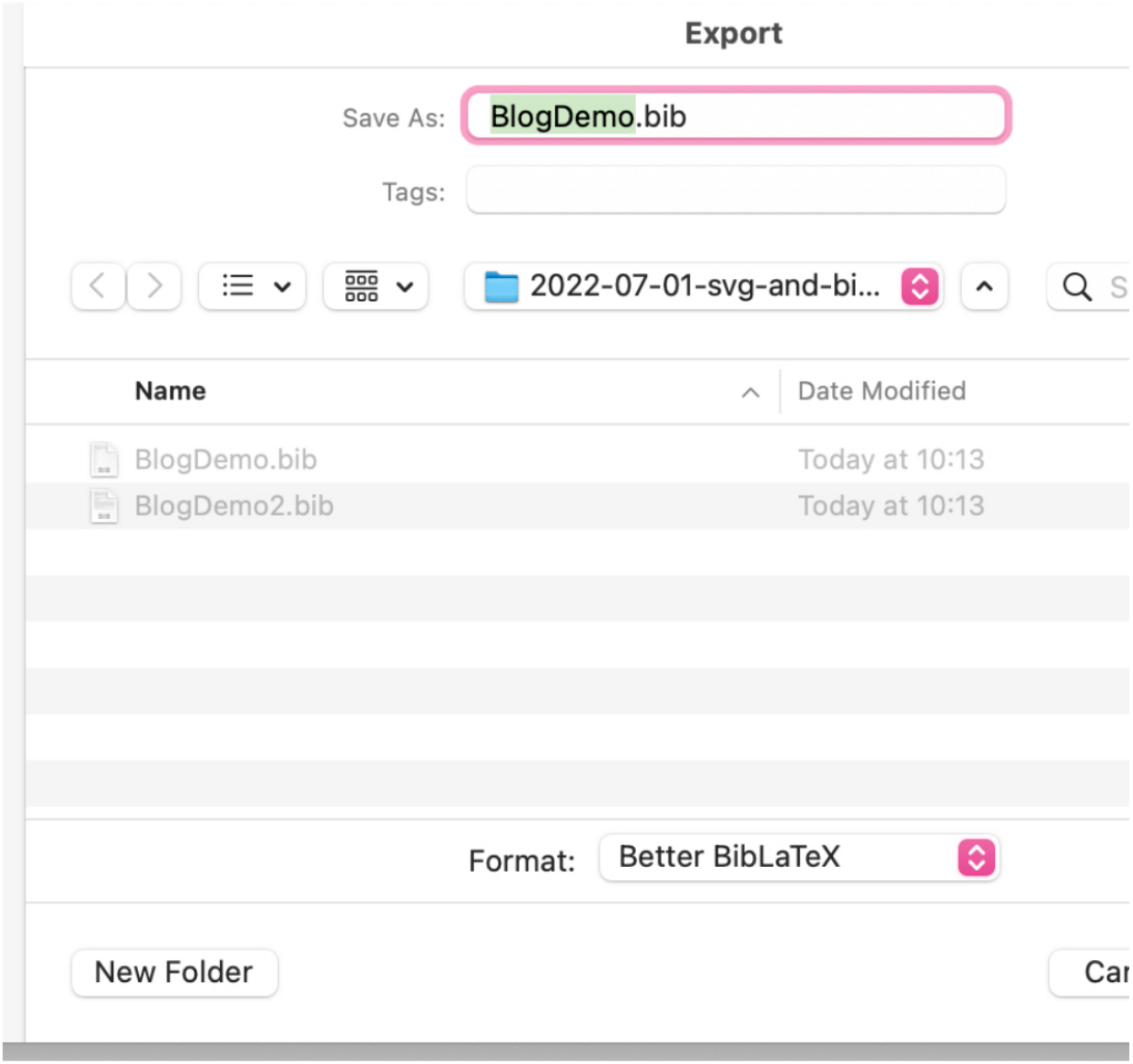
Here is the procedure :

1. export the bib file :

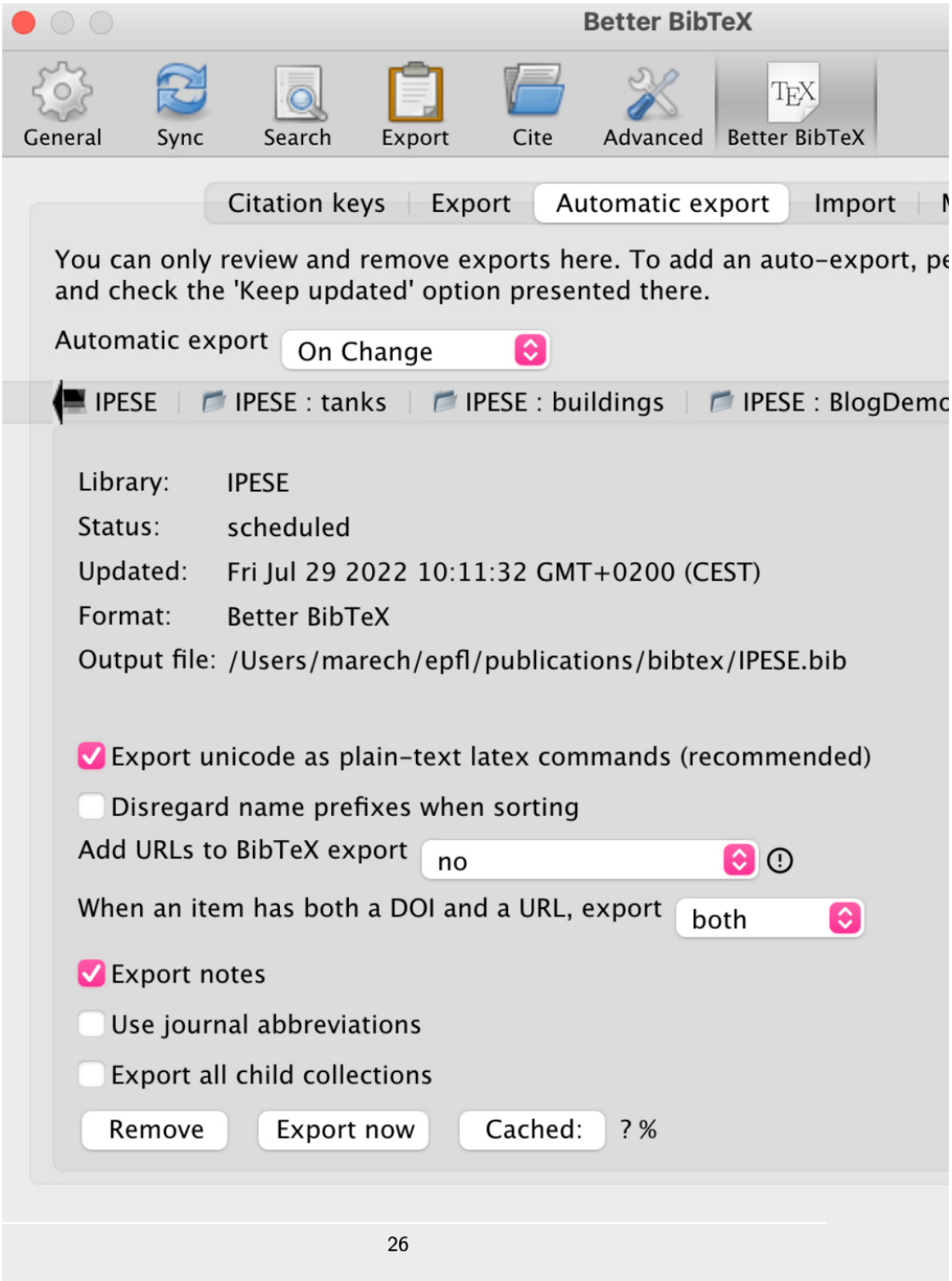


2. make it automatic and export the notes and save it inthe folder of the post.





3. update in case you need fine tuning : go to preferences of zotero, go to export -> automatic export -> tab with your bibfile.


Better BibTeX
General
Sync
Search
Export
Cite
Advanced
Better BibTeX
Citation keys
Export
Automatic export
Import
You can only review and remove exports here. To add an auto-export, pe
and check the 'Keep updated' option presented there.
Automatic export
On Change
IPESE
IPESE : tanks
IPESE : buildings
IPESE : BlogDemo
Library: IPESE
Status: scheduled
Updated: Fri Jul 29 2022 10:11:32 GMT+0200 (CEST)
Format: Better BibTeX
Output file: /Users/marech/epfl/publications/bibtex/IPESE.bib
Export unicode as plain-text latex commands (recommended)
Disregard name prefixes when sorting
Add URLs to BibTeX export
no
When an item has both a DOI and a URL, export
both
Export notes
Use journal abbreviations
Export all child collections
Remove
Export now
Cached:
? %

4. declare the bibliography in the header of the post by adding the following lines in the yaml

```
---
title: My title
author: Me
bibliography: BlogDemo.bib
biblio-style: apalike
---
```

Now each time you add a reference in the sub-collection, the bib file in the folder of your project will be updated. If you want to cite one of the paper in the bib file, just add the following instruction `[@li_carbon_2020]` and a reference to the paper with citation key li_carbon_2020 will be added in your report [@li_carbon_2020].

Step 2 : define the data you are going to look for in the bib file

When you perform a bibliography search, you are supposed to know what you are looking for. Let us for example consider that we are looking for the information on an entity for which we are looking for prroperties. We have therefore to describe what are are going to search by giving the following information :

- a **name** to an entity with a **short description of its meaning**
- a list of **properties** that characterise the information (*value*) we are looking for the entity **name**.
- Each of those values needs to be defined with the appropriate **name**, **physical units**, **ranges** and **description** that is needs to defined before starting the search.

When using readbibdata, you will have access to a simplified markup language that will define the data you are looking for. The language is described here below :

```
+- EntityName # Entity Name : This is the data entity I'm looking for which I'm looking
property1=10.0 [m^3] # Property 1 : this is the description of property 1
property2=15. [kg] # Property 2 : this is the property2 of the entity Name with a defaul
+-
```

It translates as *I'm looking at* **property1** *and* **property2** *of the entity that has the data name* **EntityName**, *that is displayed with the short name* **Entity Name** *and has the meaning of* **This is the data entity I'm looking for which I'm looking for**. *The property with name* **property1** *has a default value of 10 and the values of property1 will be displayed with a format of "xx.x" (one digit) and has the physical unit* m^3. *It will be displayed in table with the short name* **Property 1** *and has the following meaning :* **this is the description of property 1** *the property2 has a default value of 15.0 kg and is defined as being* **this is the property2 of the entity Name with a default value of 15 in kg** *that will be displayed with the short text* **Property 2**.

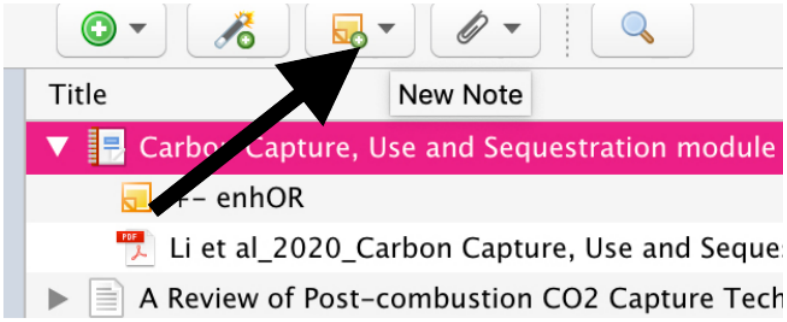
You can consider that this is like having defined a header with a list of cololumns with names **Property 1**, **Property 2** to a table with a title **EntityName**, the text after the fist # is the caption of the table, each column of the collumn corresponding to one property and each line corresponding to an instance of the entity found in the literature search.

Step 3 : note the values of the data in the bibliography file

In Zotero, values of the properties will be found by reading the papers. In order to record the value found you will add a note to the corresponding annotation of the paper in Zotero.

In order to create the instances, you are making a literature search. In each paper and for each instance of the properties found for the corresponding entity, you are going to write the values you found in the annotations of the bibliography item of your zotero data base. Just paste the description above in the corresponding annotation and replace the value, perhaps the physical units and the comments as they will be associated with the value.

This can be done either



- by adding a note via the Zotero panel
- by edding note via the edition of the pdf in zotero. In this case it will be necessary to update the notes of the pdf annotation before being able to use the annotations saved in the bibliogrphy file.

To do so you will first copy and paste the following code in the note

```
+- EntityName # Entity Name : This is the data entity I'm looking for which I'm looking
property1=10.0 [m^3] # Property 1 : this is the description of property 1
property2=15. [kg] # Property 2 : this is the property2 of the entity Name with a defaul
+-
```

and update the values and comments to the value found in the paper , ie. your note will look like this :

```
+- EntityName # V1 : [reference] : This is the set found on page 5 corresponding to vers
property1=12.0 [m^3] # Converted from the value reported in kg assuming a density of 0.8
property2=13. [kg] # This is an assumption
+-
```

Note the changes : **V1** is the name of the instance of the entity as observed in the related paper. the values in brakets identify the set of data the instance belongs to after the ":", we have a textual description of the instance of the entity found. The value of property1 has been updated with the value "12.0" found in the paper, the same for the value "13" of property2. For each property what follows the "#" is used to comment the data collected.

In a paper you can have more then one instance of EntityName. If the nstance name is not given it will be assigned a unique ID.

Step 4 : load the data from the bib datafile

Once you have collected data, the data are saved in the annotation of the papers in the bib file of Zotero. The next step consist in reading the bib file and extracting the data that can later be processed in the Rmarkdown file as a R dataframe.

The example below shows how to create a dataframe **EntityName** in R by parsing the bib file defined in the bibliography statement.

```
```{r}
EntityName <- readbibdata(
                    "
+- EntityName # Entity Name : This is the data entity I'm looking for which I'm looking
property1=10.0 [m^3] # Property 1 : this is the description of property 1
property2=15. [kg] # Property 2 : this is the property2 of the entity Name with a defaul
+-
")
```
```

Note that the text of the readbibdata call is the description of the entity you are looking for. Only this entityname will be searched for even if there are a lot of other entities in the bib file.

the dataframe obtained has the following content.

For each intance "x" of the EntityName we will obtain the following information :

- EntityName[x]$property1$value : value of the property 1
- EntityName[x]$property1$unit : physical unit of the property 1
- EntityName[x]$property1$shortname : shortname of property 1
- EntityName[x]$property1$description : description of property 1
- EntityName[x]$property1$comment : comments related to the specific value of property 1
- EntityName[x]$property1$min : minmum value expected for property 1
- EntityName[x]$property1$max : maximum value expected of the property 1
- EntityName[x]$property1$default : physical unit of the property 1
- EntityName[x]$property1$year : year of the publication
- EntityName[x]$property1$citationkey : citation key of the paper
- EntityName[x]$property1$Authors : authors of the paper
- EntityName[x]$property1$journal : as from the bib fields
- EntityName[x]$property1$set : name of the sets the value belongs to

Here is for example the results of the call from the demonstration BlogDemo.bib