Mingo tutorial

carolframa@gmail.com

August, 2023

Contents

This text is principally intended for physics students from the USC who are not familiar with databases, GitHub repositories nor object-oriented programming and who may find overwhealming to work with the mingo package. There may be mistakes along the text and certaintly a lack of rigous in the explanations provided here.

Notice that the information registered here is from a Mac's user so there may be some differences in some of the steps if you work with Windows or Linux.

What is mingo ?

Mingo is a "Python package with data analysis utilities for the MiniTrasgo project". You can find the mingo repository developed by Alfonso at `https://github.com/alfonsoSR/mingo`.

The code you find there was designed to analice the data measured with the Mingo detector. This detector was designed to register data from electron cascades. Within the code there are several files that contain data obtained from a simulation that recreates what we would measure in a Mingo detector for a certain given conditions (we will come to this later, this is just to give some context).

The final goal of the code is to use its classes and methods without having to interact with the code itself. That is, you import from the mingo package the classes or functions you want to use and you just use them. Following what is explained in the *demo* Alfonso provides[1], we would have for example a code like this:

```
from mingo import DBInput, Database, report

dbinput = DBInput("example", username= "carol")

db = Database(dbinput)

db.fill("doc/demo/sources")

report(db, "example-report.pdf")
```

In this code[2] we use the *report* function to create a report without interacting with the mingo code. This is really confortable for the user but it might be limited if you want to make specific studies with the data. Here we will try to provide some information in order to make modifying the mingo code easier.

1 Installation.

Our first step is to clone the mingo repository from GitHub. By doing this we are creating a new folder somewhere in our computer where we will have access to the Mingo code.

[1] You can find a Jupyter Notebook with an explanation of the code we provide here in his GitHub repository: `https://github.com/alfonsoSR/mingo/blob/main/doc/demo/demo.ipynb`.

[2] Notice that in this case the argument for the *fill()* method is different from the one in the *demo* because here the python script was not in the folder where the sources are.

I'll try to be very detailed in this section because for me, this was the most difficult part and I needed a lot of help to get it installed.

Before doing anything, there is a key matter: the program is not configurated to be installed with Conda or Anaconda. If you are using Spyder for programming with python, I'm afraid you will not be able to access the mingo code. Alfonso's recommendation is to use Virtual Studio Code[3] . If you are coming from Spyder you may take a little to get use to it but it will be worthy.

Another Alfonso's recommendation to make your life easier is to install Homebrew. By installing this you will be able to install and uninstall the applications or programs you want from your terminal by doing 'brew install python3' or 'brew uninstall python3'. This is really useful beacuse you avoid dealing with the installations from your internet browser[4].

Once you have Homebrew installed and in order to clone Mingo's repository, you should also install Git by doing 'brew install git'. Git is a system that provides the user the ability to track every change you do in your project files and makes it easier for a group of people to work in the same project. Once you have Mingo installed, you will be able to keep it updated if there is any change made in the GitHub repository by doing 'git pull' and you can also submit the changes you do by doing 'git push'[5].

With all this done, I think you should be able to clone the repository somewhere in your computer by doing 'git clone https://github.com/alfonsoSR/mingo.git' so you now have some folder named Mingo at some location in your computer. Now you can install the package by doing 'pip3 install -e .' being sure that you are inside the Mingo folder in your terminal[6]. To check that everything was right you can run 'pip3 list' anywhere and you should see a list of packages that contains one named mingo.

Next step is to install Postgresql@15. PostgreSQL is a database management system that allows you to organise your information in terms of databases with which you can interact. You can install it by doing 'brew install postgresql@15' and you can start a server to use the database with 'brew services start postgresql@15'.

At the moment, the version 15 is not the one installed by default, that is, if you run 'brew install postgresql' the version you are installing is version 14. This may cause some problems when using the version 15 because it is not installed in the default location and you may need to run 'brew link postgresql@15' in order to stablish a link that connects the locations where the version 15 is installed with the default location (the last one is the location where the computer is going to search in order to use Postgres, that's why you need to have your version 15 linked to that place). You can ignore this step if you don't encounter any problem.

Last step is to install all the packages used in the code, the ones you can find in the file 'requirements_dev.txt'. In order to do this step, I recommend you to create a vitual enviroment to store all the packages. A virtual enviroment is just a folder that is within your code and that contains all the

[3]Virtual Studio Code is a programming enviroment which is not linked to a certain program as Anaconda is. This means that you can use this interface to program with any lenguage you want or even for LaTeX, for example.

[4]If you come from Anaconda you should be careful about where Python is installed in your computer. You should only have Python installed once. For me, the pyhton version installed with Hombrew is at '/opt/homebrew/bin/python3' (for a M1 Mac, you can check where your Python is by running 'which python3' in the terminal). If you have several versions of Python in more than one folder, I recommend you to uninstall all of them and install Python again with Homebrew.

[5]You can find some useful information about Git and GitHub here:Git Guide and Git & GitHub cheat sheet.

[6]To navigate between the locations in your computer (that is, the folders) with your terminal you can use 'cd folder' where 'folder' is replaced by the folder you want to get in (you can see the folders that are in the location you are at the moment by doing 'ls') and 'cd ..' to get to the previous folder that contains the one you are at. With 'pwd' you can see where you are at.

packages you use in your code[7]. To create a virtual enviroment you run 'python3 -m venv mingo-venv' in the Mingo folder (instead of 'mingo-venv' you can use the name of your choice for your virtual enviroment). Once you create it you can activate it by doing 'source mingo-venv/bin/activate' (for Windows you would use 'mingo-env\Scripts\activate.bat'). Once it is activated you can install all the packages required by doing 'pip3 install -r requirements_dev.txt'. You can check that everything is correct by running the tests (see section 2.4).

Assuming you have Hombrew, Git and Python installed, the abbreviated steps would be:

1. Clone the repository in your computer.

```
git clone https://github.com/alfonsoSR/mingo.git
```

2. Install the package with pip from the `mingo` repository.

```
pip3 install -e .
```

3. Install PostgreSQL 15 and start the server.

```
brew install postgresql@15
brew services start postgresql@15
```

4. Create a virtual enviroment and activate it.

```
python3 -m venv mingo-venv'
source mingo-venv/bin/activate
```

5. Install the required packages.

```
pip3 install -r requirements_dev.txt
```

If everything went right you should now be able to edit your mingo package!

2 Mingo structure.

I'll try to explain here how the code is structured in order to make it easier to edit it. I'm assuming you might not be familiar with python classes. There are plenty of tutorials on the internet, but for me the most useful resource was this Python Programming: Object-Oriented Design course. Also you might have not worked with databases in a code, but don't panic. I found it quite intuitive since the hard work was already made by Alfonso. You may need to do some consult in order to get the data you want from the database but that's it.

[7]Jupyter Notebooks is a 'package' that you need to install in the virtual enviroment in order to use it.

2.1 *database.py*

Let's start with the script that designes the structure of the database. This is not a script that you should edit since it only structures the data from the source files. I'm just including it here in order to give some insights to make it easier to interact with the database.

I'm recovering here what I mentioned at the begining about the mingo detectors. In order to understand how the database is structured you need to know how a Mingo detector is designed. It is basically formed by 4 active detectors that may or may not have some kind of absorbent material below them (in order to increase the electron generated in the cascades). The files we work with record the information of all the electrons generated by an initial one that impacts at the central point of the first detector (set to be like that in the simulation).

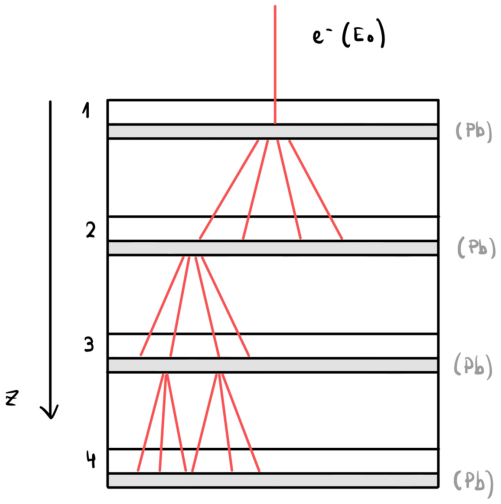

Figure 1: Mingo detector scheme.

With this in mind and with the specifications Alfonso did at the begining of each class, I think it should be relatively easy to get the idea of the database structure[8].

In the GitHub repository you can find a entity-relationship model (ER) that describes the relationships between the entities that form our database (the *er-model.drawio* file)[9]:

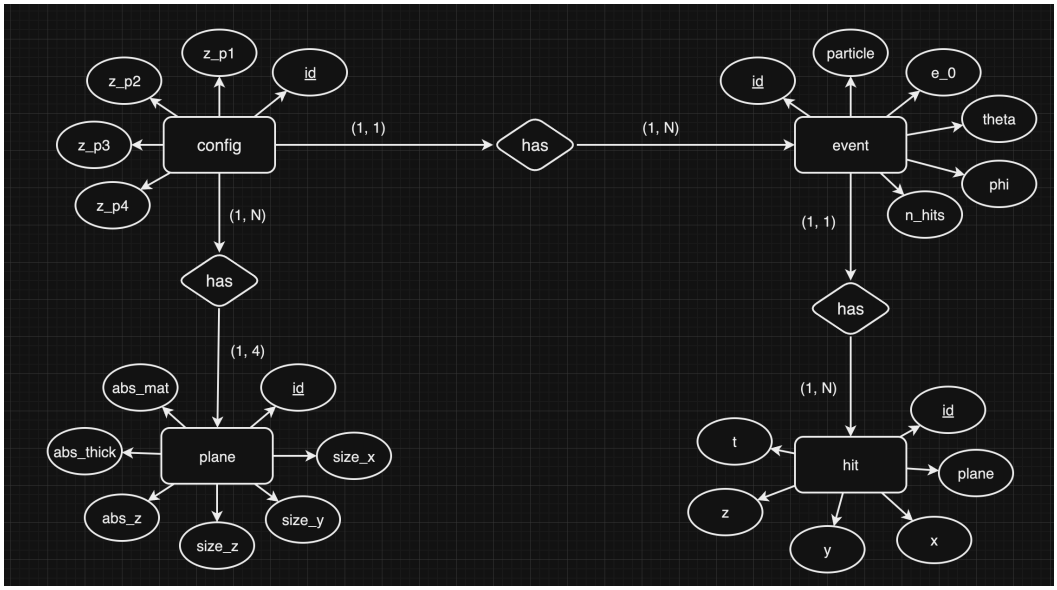

Figure 2: Mingo database structure.

[8] A simple video that explains the main idea of using databases: Database Tutorial for Beginners

[9] A square represents a table, a rhombus represents a relation and a circle represents a column.

The definition of the magnitudes presented in the previous ER can be found in the documentation of the classes defined in the *database.py* script.

I'll present now the structure of each table so you can visualice what you are doing every time you make a consult in your database. You can also access them from your terminal by connecting to PostgreSQL running 'psql "example"' where 'example' is the name of your database (this is the one we put in the code at the begining of the text, if you used Alfonso's demo, your database name is "demo").

Let's start by presenting the 'plane' and the 'config' tables:

```
example=# SELECT * FROM plane;
 id | size_x | size_y | size_z | abs_z | abs_mat | abs_thick
----+--------+--------+--------+-------+---------+-----------
  1 |    999 |    999 |     22 |       |         |
  2 |    999 |    999 |     22 |    22 | Pb      |      10.4
  4 |    999 |    999 |     22 |   222 | Pb      |      16.2
(3 filas)

example=# SELECT * FROM config;
 id | fk_p1 | fk_p2 | fk_p3 | fk_p4 | z_p1 | z_p2 | z_p3 | z_p4
----+-------+-------+-------+-------+------+------+------+------
  1 |     1 |     2 |     1 |     4 |    0 |  100 |  200 |  400
(1 fila)

example=#
```

Figure 3: Plane and config tables, respectively.

As you can see in the 'plane' table we associate an 'id' to each of the detector's planes, in this case we only need 3 different planes to describe our detector, each of which has a different amount of lead below.

All of the data is measured with the same detector since we only have one row in the 'config' table. The planes associated to this detector are the ones described in the 'plane' table.

Let's take a look now to the 'event' table:

```
example=# SELECT * FROM event WHERE id < 15;
 id | fk_config | particle | e_0  | theta | phi | n_hits
----+-----------+----------+------+-------+-----+--------
  1 |         1 | electron | 3200 |     0 |   0 |     99
  2 |         1 | electron | 3200 |     0 |   0 |     61
  3 |         1 | electron | 3200 |     0 |   0 |    115
  4 |         1 | electron | 3200 |     0 |   0 |     51
  5 |         1 | electron | 3200 |     0 |   0 |     68
  6 |         1 | electron | 3200 |     0 |   0 |     88
  7 |         1 | electron | 3200 |     0 |   0 |     73
  8 |         1 | electron | 3200 |     0 |   0 |    119
  9 |         1 | electron | 3200 |     0 |   0 |     48
 10 |         1 | electron | 3200 |     0 |   0 |     80
 11 |         1 | electron | 3200 |     0 |   0 |     67
 12 |         1 | electron | 3200 |     0 |   0 |     92
 13 |         1 | electron | 3200 |     0 |   0 |     36
 14 |         1 | electron | 3200 |     0 |   0 |    104
(14 filas)
```

Figure 4: Event table.

Each event corresponds with an initial electron impacting on the first plane and generating an electron cascade of *n_hits* electrons. As you can see *theta* and *phi* are always 0 because the simulation is made so that the electron impacts vertically.

The last table is the 'hit' table:

```
example=# SELECT * FROM hit WHERE id < 15;
 id | fk_event | plane |   x    |    y    |   z   |   t
----+----------+-------+--------+---------+-------+--------
  1 |        1 |     1 |      0 |       0 |     0 |      0
  2 |        1 |     2 | 0.8035 | -0.6089 |   100 | 0.3669
  3 |        1 |     3 |  6.846 |   5.975 |   200 |  0.702
  4 |        1 |     4 | -18.42 |  -12.11 |   400 |  1.375
  5 |        1 |     3 |  3.003 |  -1.928 | 219.6 |  0.766
  6 |        1 |     3 |  3.003 |  -1.928 | 219.6 |  0.766
  7 |        1 |     3 |  3.003 |  -1.928 | 219.6 |  0.766
  8 |        1 |     2 |  0.795 | -0.4428 | 101.7 | 0.3726
  9 |        1 |     2 |  0.795 | -0.4428 | 101.7 | 0.3726
 10 |        1 |     3 |  45.27 |    20.3 |   200 | 0.7421
 11 |        1 |     3 |  42.82 |  -73.94 | 212.2 |  1.663
 12 |        1 |     3 |  37.73 |  -59.87 | 209.2 |  1.612
 13 |        1 |     2 |  84.23 |   -33.4 | 102.6 |  1.214
 14 |        1 |     4 | -207.8 |   328.9 |   406 |  2.403
(14 filas)
```

Figure 5: Hit table.

In this case what we are looking at is the information of all of the electrons that the detector registers. The first electron of the table is the one that generates the cascade and the following ones are the generated electrons. As you can see in in the 'event' table we have 99 electrons (*n_hits* in the first row of the 'event' table) associated to the first electron that impacted in the detector (the one we associate to the variable *fk_event = 1*).

The files we work with all have a number of events equal to 10000 and there are now 8 files so the event table has 80000 rows, each of them associated to *n_hits* number of hits.

2.2 *analysis.py*

This script is where all the classes used to make the analysis are stored. If you want to study a new variable or make a new different statistical study this is the script you have to modify (or you can also create a new one if you feel more confortable).

I'm focusing the explanation provided here in creating a new variable of any kind to expand the report file you obtain when you call to the *report* function. If your analysis goes beyond this, you should probably create a hole new different class appart from the ones Alfonso uses.

You can find in each class a short explanation on what is being done so you can follow the code easily. The main idea is to use the *Base* class as a parent class for all the variables you want to study in your report (*Hit_distribution*, *Shower_depth*...), all of them inherit from the *Base* class.

By doing this there are some common methods that you declare within the *Base* class and that all the subclasses share (such as *Plot_distribution*, which plots the graphic). This makes it quite easy to create

a new variable because you only need to edit the methods which make the SQL consult to access to the data stored in the database in order to recall the information you need to define the variable.

This means that in order to create a new variable, you have to create a new subclass of the *Base* class. The methods you need to define in your subclass appart from the *__intit__()* method are *_dist_tmp()*, *_dist_stmt()* and *_stats_tmp()*. This three methods are the ones that contain the SQL consults where you select the information from the tables of the previous section and you define the variables you want to study.

By changing this methods and the consults you do you can create a new variable. Once you do this you should remmember to edit the *report* function in order to include the new varibale and edit the *__intit__.py* script (see section 2.3) to be able to import the new variable class if you want to use its methods without genereting the report:

```
from mingo import DBInput, Database, Hit_distribution
import matplotlib.pyplot as plt

dbinput = DBInput("example", username= "carol")

db = Database(dbinput)

hit_dist = Hit_distribution(db)

id = 1 ; label = 'Hit distribution'

hit_dist(id,label)

hit_dist.plot_distribution()
plt.show()
```

2.3 *__init__.py*

This file serves as the entry point for the package and allows you to define what should be exposed and accessible when the package is imported elsewhere. If you want a class or a function to be accessible for the user by importing it, you have to include it here.

2.4 Tests.

The idea of python's tests is to ensure that your code is doing what you want it to do[10]. If you want to check that your code is correctly installed or if the changes you made didn't break any of the code you can run this tests to see if everything is correct. In Visual Studio Code is quite simple, you can do it from the testing section at the side bar (the one with the flask form).

The code you find at GitHub has Alfonso as username in the file *mock_data.py*, you should change this to your username name so you don't encounter any errors when running the tests.

[10] A video I found quite useful to get an idea about python tests is: Unit Tests in Python