**GIC\_BAM**

Software to compute Geomagnetically Induced Currents (GIC) in power networks, based on the Bus Admittance Matrix method (BAM).

See scientific publication at:

<https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2021SW002984>

and specifically, the publication’s Supporting Information for detailed explanations and variable definitions.

There is a Matlab and a Python version for the code in the GitHub. Below is some guide and basics for the Python version:

**Guide – GIC BAM programs - Python**

* **How to install the environment GIC\_BAM.**

To install the environment first you need to have Anaconda installed in your computer. You can search it on google as “Anaconda Python” or use the link below:

<https://www.anaconda.com/products/distribution>

Once it has been installed correctly, we need to download the “GIC\_BAM.yml” file and put it at our USER file:

C:\Users\USER

Now you need to open the Anaconda Prompt, that’s a terminal that we’re going to use to create and manage our environment.

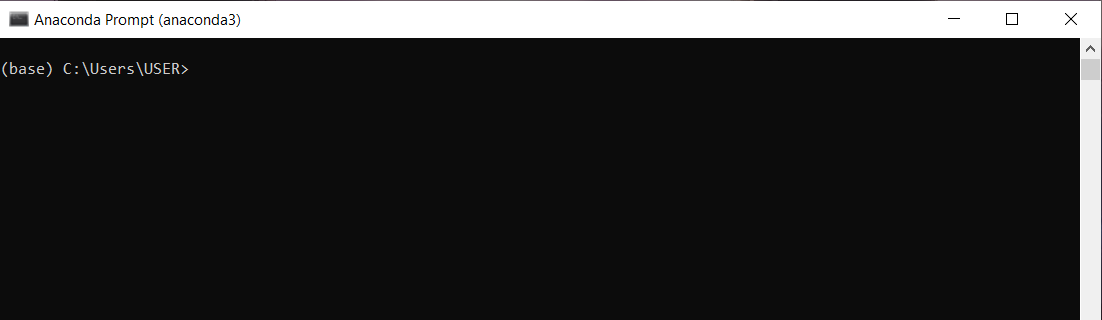


Image of the Anaconda Prompt once it’s open

To create the environment, we are going to write in the command line:

conda env create -f GIC\_BAM.yml

And we are going to let it execute it. This process will download all the libraries that we are going to need so that our program can execute itself correctly. It will also install the program Spyder that’s going to be specific for our environment, so every time we want to execute the script, we are going to open the program Spyder (GIC\_BAM) that contains the environment and all the libraries we need.

If we want to modify this environment, we need to activate our environment because when you open Anaconda Prompt you are always going to be at the base environment, the one that comes when you install Anaconda.

conda activate GIC\_BAM

So, if you want to install a new library or update the environment, first we need to activate it.

* **GIC\_BAM code commented**

This code is separated in different cells that are executed one after the other. Each of them has a specific task. The next sections we are going to talk about the different variables and different ways in which the code is presented.

**Variables**

**Filename**: this variable serves as the name of the excel file that the code will use to get the information of the power grid. It’s very important that the excel file has the same information and in the same format as the excel files that come with the code. To use your own file, you just need to overwrite the string with the name of the file and its extension.

**WriteEqNetXLS**: this variable can have different values but the most important if it’s equal to 1, in this case the code will write two new pages on the excel file.

**E\_homo\_hetero**: this variable serves to choose the type of electric field we are considering. If we want to consider a homogenous field, we need to put this variable at 1 and specify the E field manually in the user section. If we want to use an heterogenous field we need to put this variable at 2 (or any other value).

If we want to use a heterogeneous field, we enter the electric field as ‘.txt’ file. This part of the code differs from the homogeneous one. Here, you need to enter the time variation of the north and the east component of the electric field as two different files that the code joins to generate a 3-dimensional array. You also need to enter a file that corresponds to the latitude and the longitude of the grid points where the E field is known.

**Reading an excel file**

With the library pandas we can read and write excel files. With the next line of code we can read an excel sheet, the output is a pandas dataframe (a way to store data in a matrix form) with the names of the columns the same as the first row of the file and with each row numerated.

Xsubstations = pd.read\_excel(filename,sheet\_name='Substations')

**Working with pandas dataframe**

Since each column has its unique name determined by the excel, we use pandas functions iloc() to refer to the columns based on each number and not on the name.

The Pandas function isnull() returns us the places where the values are NaN. Combining this with the tile command ~ we get an array with all the real components in that column, eliminating the NaN components.

Xsubstations[~Xsubstations.iloc[:,0].isnull()].iloc[:,0]

We can transform a pandas dataframe to a numpy array adding .to\_numpy() at the end of the dataframe.

With the command .index() we get the index that refer to each value from the dataframe.

**Raising Errors**

To stop the code if there’s an error, we use the next format:

if condition:

raise ValueError(‘message’)

If a condition is met, such as the dataframe having more columns that it should, we raise an error, and display a message on the console.

**Working with arrays**

To join two arrays of the same length that have one dimension, we use np.dstack() to join the arrays along the second dimension. The resulting array has the same length but, in each row, with two components.

To generate an array full of NaN’s we use np.full() that generates an array of the given size with the value we want. In the code we have used this expression multiple times to generate empty arrays.

np.full((m,1),np.nan)

To obtain all the values from a multidimensional array we use the command .flatten() which sorts the array in just one dimension. To rearrange the values in the opposite order we use np.flipud() function from the numpy library.

With np.argwhere() we obtain the positions of an array where a condition is satisfied.

To join two arrays horizontally, we join them through the columns, we use np.hstack(). To repeat an array, we use np.tile() where it repeats an input several times equal to a repetition number. We first join the latitude and longitude of the substation with the resistance using hstack while later we repeat this combination a total of L+1 times, making a 3-dimensional array with length equal to L+1 times the original hstack array.

There are multiple ways of flattening an array, a common method is using the command ‘A’ where we flatten the array through the rows.

To generate an empty list, we multiply a list with a None component by a given length.

To select a specific part of an array we use the two-dot notation where: “start : end : step” usually the step is not specified since most of the times is 1.

If we have an array and we want to check if a certain condition is somewhere true in all the array any(), which returns a True if the answer is True at any position of the array. The command all() only returns True if all the components are True.

We check if an array doesn’t have any finite values with np.isfinite() in combination with the ~ command, where we obtain all the places where we have not finite numbers.

**Dividing by 0**

Python prints a warning when you try to divide a number by 0. To avoid this warning, we check if the denominator is equal to 0, in this case we directly put that the result is infinite, otherwise we let python do the operation.

**Writing an excel file**

To overwrite the excel file we first transform the arrays to pandas dataframe by using:

pd.DataFrame(array, columns = [])

In columns we put a list of strings. When we write the excel page the first row will contain this string list.

Since we want to write an excel file with pages with the same names as the ones we are going to write we use the function load\_workbook from the openpyxl library (1). To write the excel we use the pandas function pd.ExcelWriter() and we specify the engine to be openpyxl (2). Then we specify that the book obtained from the workbook and the sheets we have are the same as in the original one, (3) and (4). To write each sheet we use the form in line (5) where we need to specify the sheet name. Finally, we save the excel (6).

|  |  |
| --- | --- |
| book = load\_workbook(filename)  writer = pd.ExcelWriter(filename, engine='openpyxl')  writer.book = book  writer.sheets = dict((ws.title, ws) for ws in book.worksheets)  NodesEqTxt.to\_excel(writer,sheet\_name = 'NodesEq',index = False)  writer.save() | (1)  (2)  (3)  (4)  (5)  (6) |

**Defined Functions**

* **cart2sph():** we transform the cartesian components x,y and z to azimuth and elevation. To do so we use the numpy functions np.hypot() which retuns the hypotheses of two arrays, and np.arctan2() to compute the arctangent of two arrays in the correct quadrant.
* **inter\_has\_nan():** we use an interpolation in a 2-dimensional grid applying nearest method for the points that fall outside the grid, then we use np.where() to substitute the values that has NaN’s for the original linear interpolation, points that are outside the interpolation.

**Working with matrices**

To multiply two matrices, we use ‘@’ where we multiply the matrices along a dot product.

To work with big matrices, we use sparse techniques so that the program runs faster. The library scipy.sparse contains different functions that serve to work with sparse matrices. To transform a matrix to a sparse matrix we use

scipy.sparse.csc\_matrix()

And to transform a sparse matrix to an array we use the function .toarray().

To get the diagonal of a matrix we use the numpy function np.diag(), if the matrix is a sparse matrix we use the function .diagonal().

We have different ways of multiplying two matrices. To apply a dot product we use the numpy function np.dot().

To apply a matrix decomposition to a sparse matrix we use the library scipy.sparse.linalg. In this code we have used the LU decomposition to solve linear systems: scipy.sparse.linalg.splu(mat). To find the solution to a linear system we apply LUdecomp.solve(mat1) where LUdecomp corresponds to the matrix decomposition and mat1 corresponds to the matrix we want to solve.

With the variable E\_homo\_hetero we have two different cases for the computation of the GIC’s. In the homogeneous case we need to check the dimension of the electric field. We consider two different cases, when we only have one value for each component of the electric field and when we have multiple values. To do so we check the dimension of the array since if we only have one measure for each component then python treats the array as one dimension rather than with two. We use the function .ndim.

If we want to transpose an array along different dimensions, we need to specify them by adding them at the parenthesis.

Mat.transpose(dim1,dim2)

**Working with trigonometric functions**

Numpy has different trigonometric functions installed, and they work with radians and not degrees, since the set of data we have are in degrees we use np.deg2rad() to change units.

**Performing 2D Interpolation**

In python to do an interpolation we need to use the interpolate library from scipy. This library contains different 2-dimensional interpolation but with some limitations regarding the interpolations we want to do. We want to do a linear interpolation with the data inside the grid we have, but with the function presented in line (1) we can’t interpolate with the data that’s outside the grid. To be able to interpolate these points we use the nearest interpolation, where the result is computed by using the value of the nearest external point that is contained in the grid, line (2).

|  |  |
| --- | --- |
| scipy.interpolate.LinearNDInterpolator scipy.interpolate.NearestNDInterpolator. | (1)  (2) |

**How to generate figures**

In this program we have generated different figures, to do this we use the matplotlib.pyplot library.

First we need to generate a new image by using plt.figure(), we can specify the title of the figure by adding a string to the parenthesis that corresponds to the title.

If we want to plot a graph we use plt.plot(). Different plots can appear in the same figure, we just need to write different lines of code, one for each plot. If we want to generate different bars instead of a plot we use plt.bar().

The axis of the figure are generated automatically and usually adapt to the data range we have, but if want to limit the plots we can use plt.xlim() and plt.ylim(). To specify the labels for each axis we use plt.xlabel() for the X axis and plt.ylabel() for the Y axis. To have both axis with the same proportion we use plt.axis(‘equal’), for example we can transform and ellipsis to a circle with this command.

If we want to separate the plots into two sub-images we can use different subplots, which correspond to smaller images that are generated in the same figure. With plt.subplot( nrows, ncol, index ) we specify each subplot for each image. The index value can go from 1 to nrows\*ncol. To set a title for the subplots we use plt.title() where in the parenthesis we specify a string.

If we want to draw different circles in a plot we first we need to get the current figure, we use fig = plt.gcf(). Then we need to get the current axis of the image with ax=fig.gca(). We can draw each circle by using plt.Circle() but to add them to the figure we need to add the circle as a patch. A patch is a class from matplotlib that is a 2D artist that serves to draw different figures. To add a patch we use ax.add\_patch().

Finally to show a figure we use plt.show().

**User input during execution**

If we want the user to add a variable we can use the input() command where the user specifies a keyboard input to use at the code. We want the user to specify which transmission line he wants to plot the GIC that flows along a transmission line. First, we check if the input is a number and by using the isfloat function that we have defined. In this function we use the command try to check if the variable can be transformed into a float number. If the user has entered a keyboard argument such as a letter or a symbol and not a number this will raise an error, we use the raise ValueError command to finish the code here. If the user has entered a number we check if this number is between our transmission lines maximum and minimum value, meaning we only plot the figure if the transmission line exists.

**Creating a Video**

To create a video, we use the OpenCV library. We generate each individual frame by creating different matplotlib figures and adding them to a frame list. We don’t show each figure by using plt.close().

Each frame is considered a matplotlib figure so to transform it into an OpenCV figure we first need to pass from matplotlib to numpy. To do so we use line (1) where we change from a drawn figure to an array. In line (2) we change from. Finally, with (3) we generate a OpenCV figure, we change from the RGB colour space that matplotlib works to the BGR that OpenCV works with.

|  |  |
| --- | --- |
| data = np.frombuffer(fig.canvas.tostring\_rgb(), dtype=np.uint8)  data = data.reshape(fig.canvas.get\_width\_height()[::-1] + (3,))  data = cv2.cvtColor(data,cv2.COLOR\_RGB2BGR) | (1)  (2)  (3) |

Once the frame list has been created, to generate a video we use the cv2.VideoWriter class, line (1), which needs different inputs to work with: file name, the fourcc (4-character code of codec used to compress the frames), number of frames per second, size of the frames. In the code we have used 5 frames per second and cv2.VideoWriter\_fourcc(\*'DIVX') as the fourcc.

With a for loop, we go through each frame, and we save each individual image (2-3).

|  |  |
| --- | --- |
| v=cv2.VideoWriter( SaveVideoAs, cv2.VideoWriter\_fourcc(\*'DIVX'),5,  (width,height))  for im in frames:  v.write(im) | (1)  (2)  (3) |

Jordi Gómez Jodar and Santiago Marsal Vinadé,

Dec. 2022