

Alexandria - User Guide

for Python and Matlab editions

Romain Legrand



Version 0.0.1

Alexandria - User Guide

© Romain Legrand 2021

All rights reserved. No parts of this book may be reproduced or modified in any form by any electronic or mechanical means (including photocopying, recording, or by any information storage and retrieval system) without permission in writing from the author.

Cover illustration: depiction of the ancient library of Alexandria, unknown artist.

To my wife, Mélanie.

To my sons, Tristan and Arnaud.

Foreword

Bayesian vector autoregressions have become the cornerstone of time-series econometrics. Despite the need for these models, there exist to these days very few software applications. On the commercial side, Eviews proposes some limited Bayesian VAR features since version 8. Only a few applications are available and implementation comes as a blackbox. On the non-commercial side, I had the privilege to develop the BEAR toolbox when I was working at the European Central Bank, between 2014 and 2018. BEAR is a Matlab-based software for Bayesian time-series econometrics proposing a wide range of state-of-the art Bayesian econometric methodologies. It was initially developed internally to meet the needs of the European Central Bank, but it quickly gained notoriety and has now become a reference tool for many central banks and financial institutions across the world. BEAR remains to these days the most comprehensive application in the field of Bayesian time-series econometrics.

Despite this success, I eventually became unsatisfied with BEAR. One reason is technical: BEAR is developed under a simple functional programming paradigm. While working with functions only is fine for a small project, this approach finds its limits for large software projects like BEAR. In this case, proper object-oriented programming becomes a necessity to maintain efficient and well-structured code. Considering this question, it eventually became apparent that the only solution was the creation of a new software from scratch, built on the object-oriented paradigm.

The second reason is economical: BEAR is developed in Matlab, a mathematical language that remains to these days dominant among economists and econometricians. Matlab is a great platform permitting fast development and efficient execution of econometric applications, but it is also a commercial product. Matlab licenses are expensive and not affordable to everyone. Using Matlab thus excludes de facto many students and researchers, especially from emerging countries. In an era of fair opportunity and open source projects, I think this situation is not an option anymore.

For these reasons, I developed a new software called Alexandria. Alexandria provides a solution to these two issues. It is developed in pure object-oriented programming, building a solid structure for long-term development. Also, Alexandria comes in two flavours: a Matlab version for veteran econometricians; and a Python version that offers a free, open source alternative that makes the product accessible to everyone, regardless of financial constraints. The two versions are strictly similar, so that the user really can pick whichever language is preferred.

A tribute to the ancient library of Alexandria, the name of the software reflects two main ideas. First, it is a library – in the computer science sense of the term – providing codes, programmes and applications for easy access to Bayesian time-series econometrics models. Second, it has the ambition of becoming an extensive collection of knowledge – as was the library of Alexandria in its time – with the hope of providing in the long run all the major models in the field of Bayesian time-series econometrics. I hope that you will enjoy using Alexandria as much as I enjoyed developing it.

Romain Legrand

Contents

1	Installing Alexandria, Python edition	1
1.1	Python: an overview	1
1.2	The Anaconda distribution	1
1.3	Anaconda installation on Windows	4
1.4	Anaconda installation on Linux/macOS	5
1.5	Alexandria: local installation	8
1.6	Alexandria: permanent installation	10
2	Installing Alexandria, Matlab edition	11
2.1	Matlab: an overview	11
2.2	Matlab installation on Windows	11
2.3	Matlab installation on Linux/macOS	12
2.4	Alexandria: local installation	13
2.5	Alexandria: permanent installation	15
3	Preparing the project	17
3.1	Creating the project folder: Python edition	17
3.2	Creating the project folder: Matlab edition	19
3.3	Creating the base data file for the model	20
3.4	Other datafiles: forecasts	23
4	Running Alexandria from the Graphical user Interface	25
4.1	Launching the interface: Python edition	25
4.2	Launching the interface: Matlab edition	26
4.3	Interface: tab 1	27
4.4	Interface: tab 2, linear regression	29
4.5	Interface: tab 3	32
4.6	Interface: tab 4	33
4.7	Interface: tab 5	34
5	Alternative ways to run Alexandria	35
5.1	Running Alexandria from the Integrated Development Environment: Python edition	35
5.2	Running Alexandria from the Integrated Development Environment: Matlab edition	36
5.3	Running Alexandria on the fly: Python edition	38
5.4	Running Alexandria on the fly: Matlab edition	40
6	Alexandria outputs	43
6.1	Console outputs	43
6.2	Graphics outputs	44
6.3	File outputs	45
7	Alexandria classes - documentation	47
7.1	Linear regression: MaximumLikelihoodRegression	48
7.2	Linear regression: SimpleBayesianRegression	51

7.3	Linear regression: HierarchicalBayesianRegression	55
7.4	Linear regression: IndependentBayesianRegression	59
7.5	Linear regression: HeteroscedasticBayesianRegressionn	63
7.6	Linear regression: AutocorrelatedBayesianRegression	67
8	Alexandria datasets	71
8.1	The Taylor rule dataset	71

CHAPTER 1

Installing Alexandria, Python edition

1.1 Python: an overview

Python is a high-level, interpreted, and general-purpose programming language. It owes its name to the BBC TV show "Monty Python's Flying Circus". It was initially designed in the late 1980's by Guido van Rossum at Centrum Wiskunde and Informatica in the Netherlands. It was first released in 1991 and developed further by the Python Software Foundation.

Relatively marginal until the 2000's, Python has been gaining in popularity since then. It represents today one of the most widely used programming languages, aside with other popular languages such as C# or Java. The Python community is estimated to more than 8 million users today.

There are several reasons which explain the success of Python. Python is powerful, yet one of the simplest languages to learn. It emphasizes readability and conciseness, making development considerably faster under Python than under alternative frameworks. It is general, meaning it can be used to develop virtually any kind of application. Thanks to these qualities, many world-class software companies have chosen Python to write their applications, including Youtube, Google, Instagram, Netflix, Reddit, Spotify or Quora, to name just a few.

Python has also become increasingly popular amongst the data science community, as major applications were developed in Python or provided API to become Python-compatible. Scikit-Learn, TensorFlow, Torch, Keras, Numpy, Pandas or Matplotlib are examples of widely used data science applications that contributed to increase the popularity of Python among data scientists. While Matlab remains to these days dominant in the econometrics community, there is no doubt that Python will also play an increasingly important role in the field, making it a first-choice language for an open source econometrics software.

1.2 The Anaconda distribution

In theory, installing Python is straightforward. On the [Python webpage](#), one can download the install file for the latest release and then proceed to the installation. In practice however, this option is not recommended. First, the basic Python installation is minimal and provides very little in terms of development tools. The only available feature is a small programme called Idle (Figure 1.1) which is hardly more than a raw Python console. Idle provides no code editor, no variable explorer and no graphic visualisation, which makes it practically unusable for development purposes.

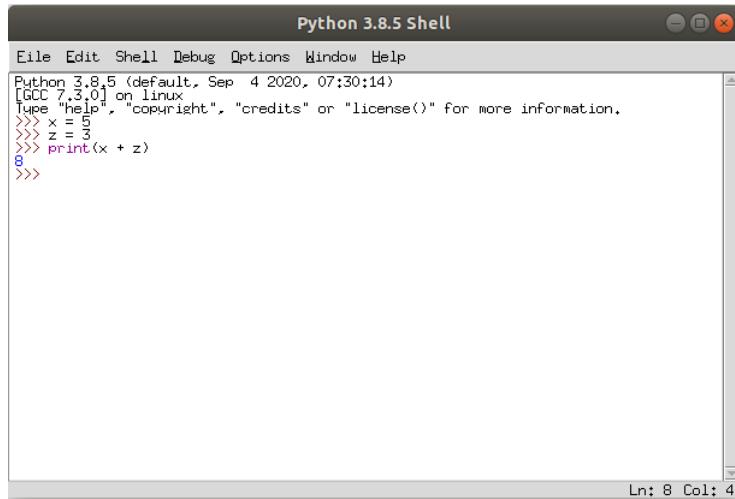


Figure 1.1: The Idle Python shell

Second, as any general programming language, base Python is extremely limited and requires additional libraries to provide suitable functionalities. With a basic installation those libraries need to be installed and maintained manually. This is impractical and creates a high risk of eventually generating conflicts between libraries, which may completely ruin the whole installation.

For these reasons it is recommended to use instead a distribution called Anaconda, which has by now become the industry standard. Anaconda is a free, open source software suite for scientific computing suitable for Windows, Linux and macOS. Anaconda makes Python more accessible thanks to a simplified Python installation and package management system.

Concretely, Anaconda does the following. First, it automatically installs Python 3 on the system, which avoids any kind of manual installation. The distribution also comes with over 250 packages automatically installed, comprising virtually any library that may be ever needed. Most importantly, it comes with its own management system that automatically handles the libraries updates and conflicts.

Second, Anaconda comes with a suite of Python development softwares, including two great applications: Spyder and Jupyter Notebook. Spyder (Figure 1.2) is a free and open-source integrated development environment (IDE) for Python. It provides a code editor (left panel), a Python console for interactive execution (bottom right panel), and a variable/figure explorer (top right panel). Spyder provides everything needed for editing, analysis, debugging and visualisation. It also comes close to Matlab and RStudio in its design, so users of these programmes will find it easy to make the move to Python.

Jupyter Notebook (Figure 1.3) is a web-based environment for creating notebook documents containing text, code, and data. It combines the features of a classical text editor with an interactive execution environment allowing to execute code and display visual elements such as tables and figures. Jupyter makes it easy to design clear and beautiful data science documents. While Spyder is primarily intended for development, Jupyter is more suitable for the execution and visualisation of existing applications. In the end however, both applications can be used to write and run Python programmes.

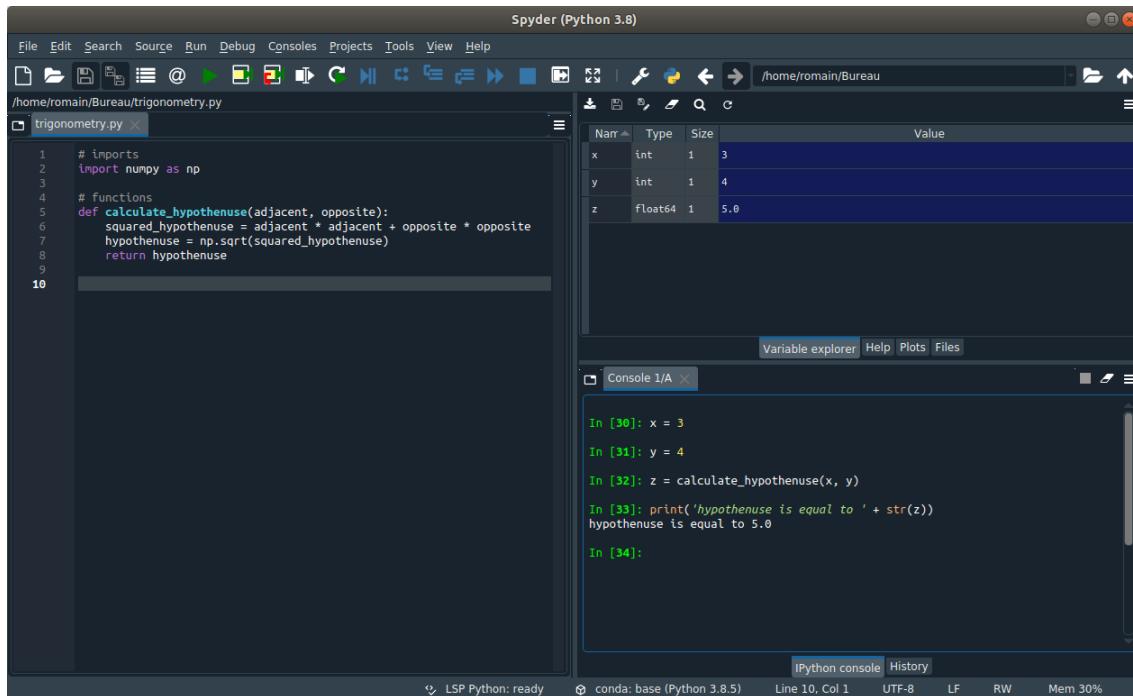


Figure 1.2: The Spyder environment

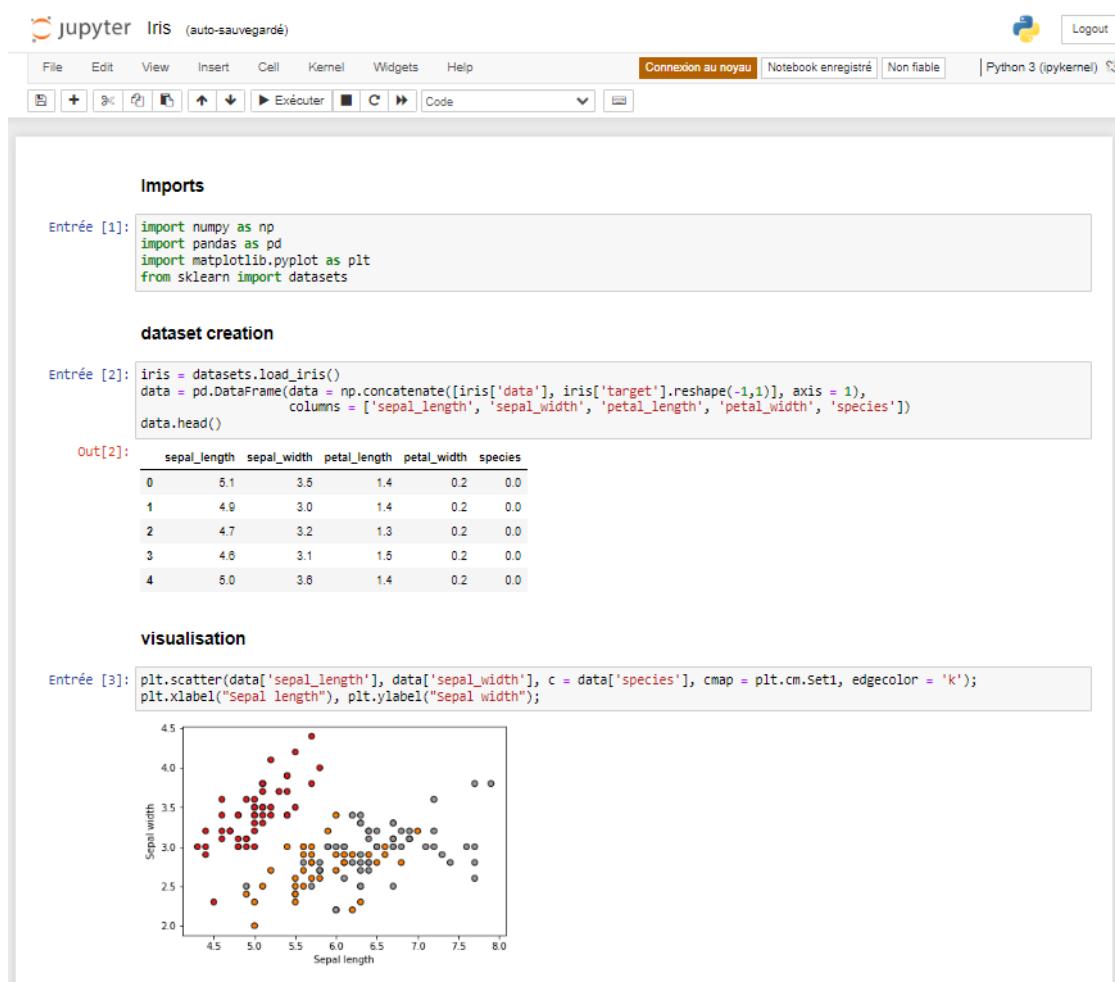


Figure 1.3: A Jupyter notebook

1.3 Anaconda installation on Windows

To install Anaconda on Windows, go to the [Anaconda webpage](https://anaconda.com/products/distribution) (Figure 1.4). The page should automatically propose to download the latest version corresponding to Windows. Click on the download button to initiate the download of the installation file. Once download is completed, navigate to the folder containing the file (it should be the Downloads folder), and double-click on the file to start installation (Figure 1.5). Follow the steps and agree when prompted to eventually complete the setup. If you experience issues in the process, you may consult the [Anaconda webpage for Windows installation](#).

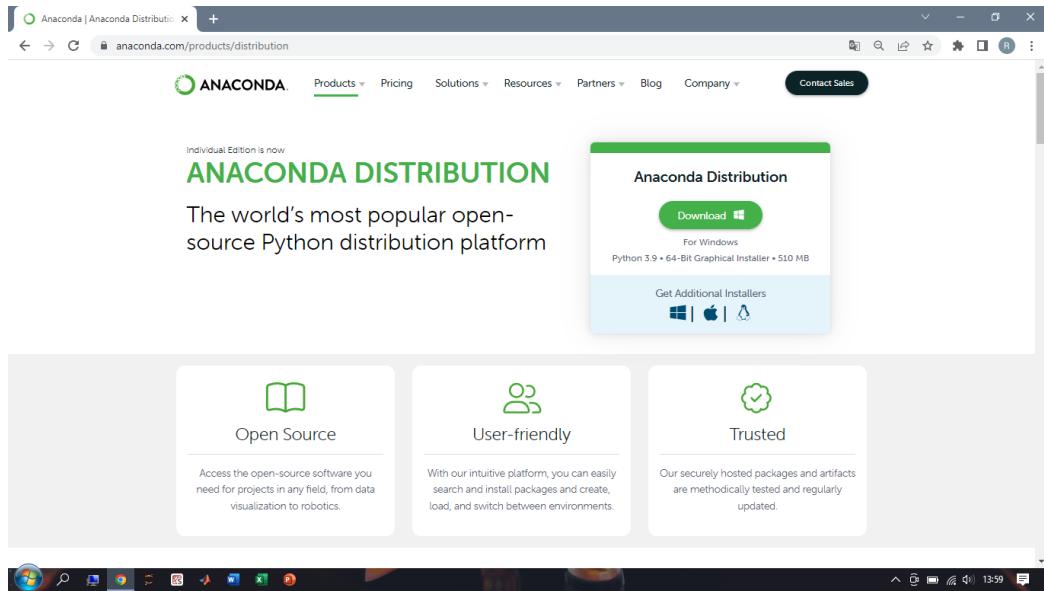


Figure 1.4: The Anaconda home page for Windows

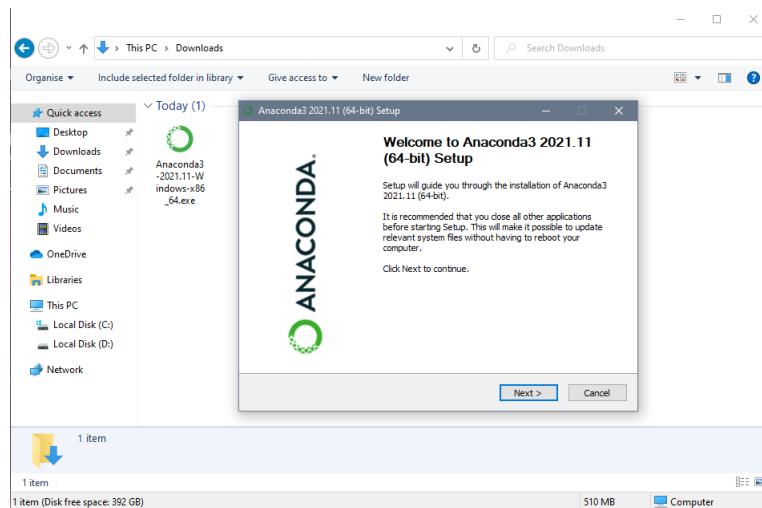


Figure 1.5: Anaconda installation on Windows

Anaconda is now installed on your computer along with Python and all the libraries you need. However, most of its material is outdated. As a final step, it is thus necessary to update the whole setup. To do so we will use the conda terminal, a facility provided by Anaconda. To open the conda terminal, search for "Anaconda prompt" in the Windows search bar (Figure 1.6, panel (a)). Then in the terminal, execute the command (Figure 1.6, panel (b)):

```
conda update --all
```

Agree when prompted and Anaconda will update Python and all the libraries, managing all possible conflicts.

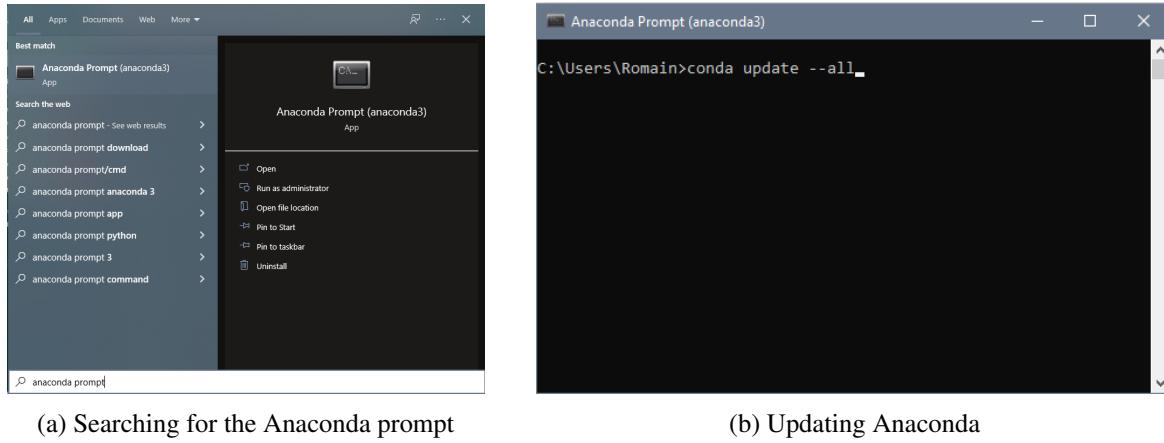


Figure 1.6: The Anaconda prompt

Finally, to open Spyder or Jupyter Notebook you can use the Anaconda navigator. To open it, search for "Anaconda navigator" in the Windows search bar (Figure 1.7, panel (a)). Then in the navigator use the "Launch" button of Jupyter or Spyder to start the application (Figure 1.7, panel (b)). Alternatively, you may create shortcuts to launch these applications directly. In the Windows search bar, type "anaconda navigator", "jupyter notebook" or "spyder", then right-click and choose "Pin to Start" or "Pin to taskbar". You may also right-click and choose "Open file location", right-click the application icon in the folder and select "Send to" > "Desktop (create shortcut)".

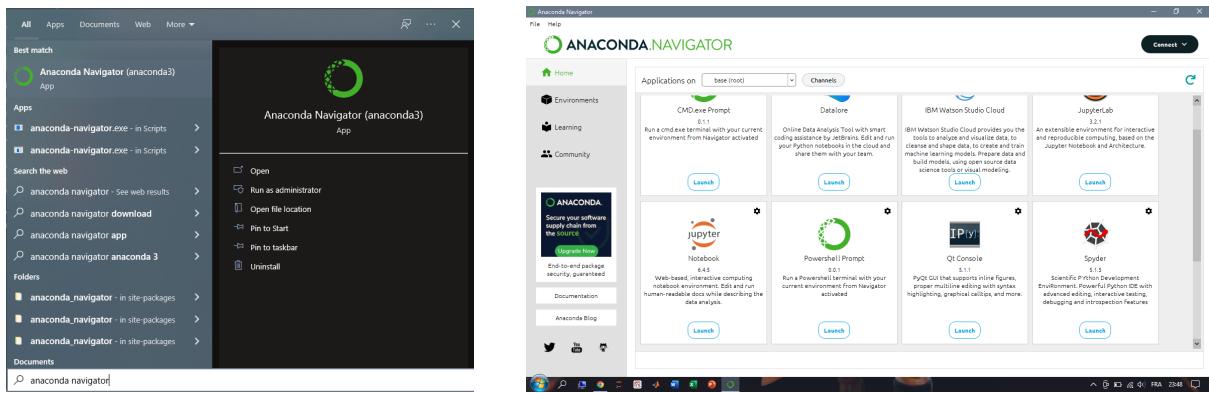


Figure 1.7: The Anaconda navigator

1.4 Anaconda installation on Linux/macOS

As Linux and macOS are both based on Unix, they follow similar installation procedures. This section outlines the steps to follow, but for more details you may consult the [Anaconda webpage for Linux installation](#), or the [Anaconda webpage for macOS installation](#).

To install Anaconda on Linux or macOS, go to the [Anaconda webpage](#) (Figure 1.8). The page should automatically propose to download the latest version corresponding to Windows. Click on the download button to initiate the download of the installation file, and wait until download is completed.

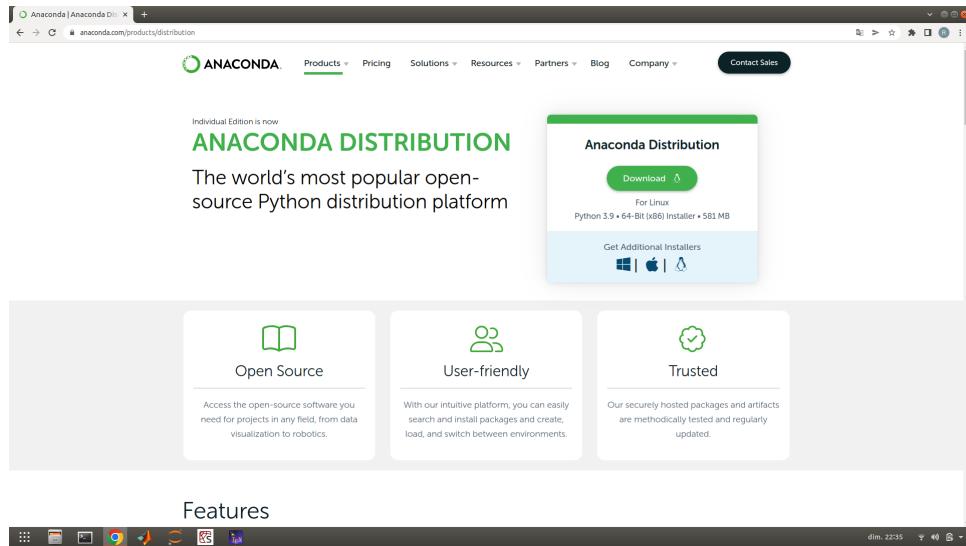


Figure 1.8: The Anaconda home page for Linux

The installation file is a bash file (a file with ".sh" extension). To execute it, we will use the terminal. If you are unfamiliar with the terminal, simply type "terminal" in the application search bar (1.9) and click the terminal icon to open a terminal.

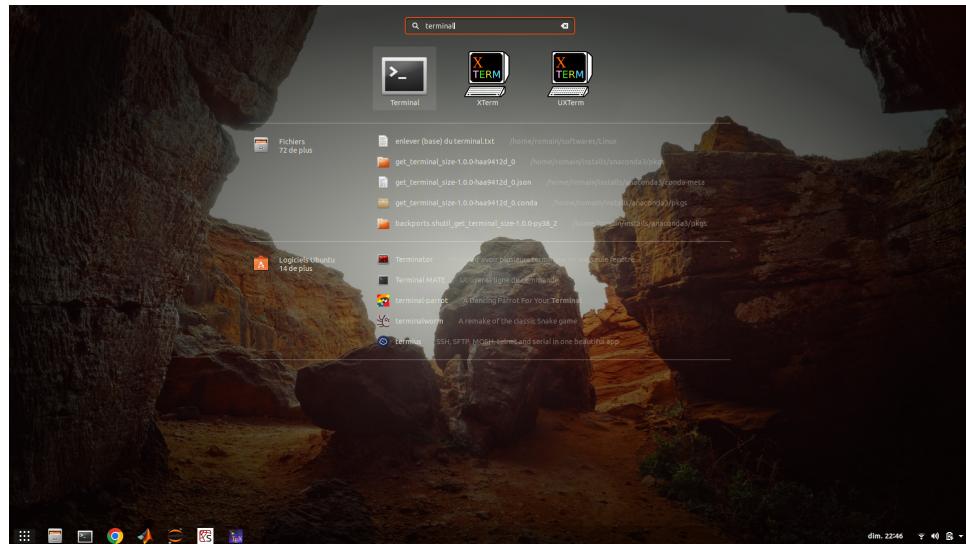


Figure 1.9: Searching for the terminal in Linux

Once the terminal is open, search for the path of the folder where the installation file has been downloaded (it should be the Downloads folder), and note the name of the installation file. Then in the terminal execute the command:

```
bash path-to-folder/name-of-file
```

So for instance if the path to the folder containing the installation file is "~/Downloads" and the installation file is "Anaconda3-2021.11-Linux-x86_64.sh", then the command to execute is:

```
bash ~/Downloads/Anaconda3-2021.11-Linux-x86_64.sh
```

Executing this command in the terminal (Figure 1.10) will start the installation, and instructions will appear in the terminal. Follow the steps, agree when prompted and keep the default options to eventually complete the setup.

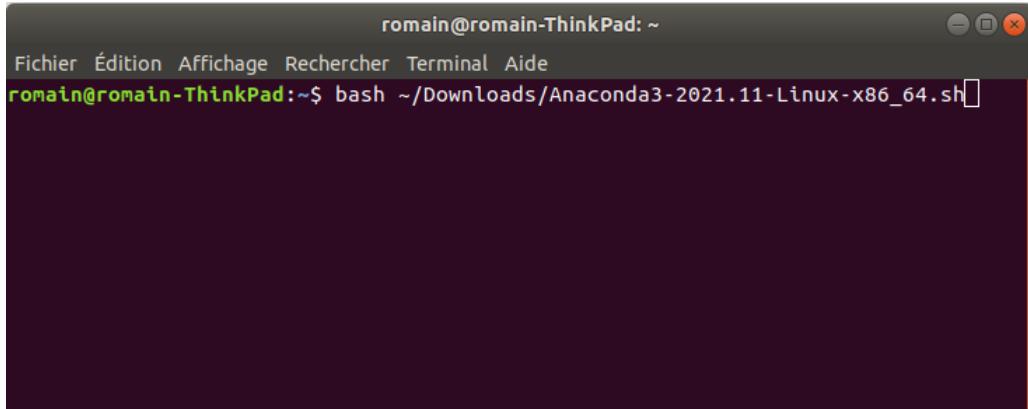


Figure 1.10: Executing the Anaconda bash install file

Anaconda is now installed on your computer along with Python and all the libraries you need. However, most of its material is outdated. As a final step, it is thus necessary to update the whole setup. To do, open a new terminal and execute the command (Figure 1.11):

```
conda update --all
```

Agree when prompted and Anaconda will update Python and all the libraries, managing all possible conflicts.

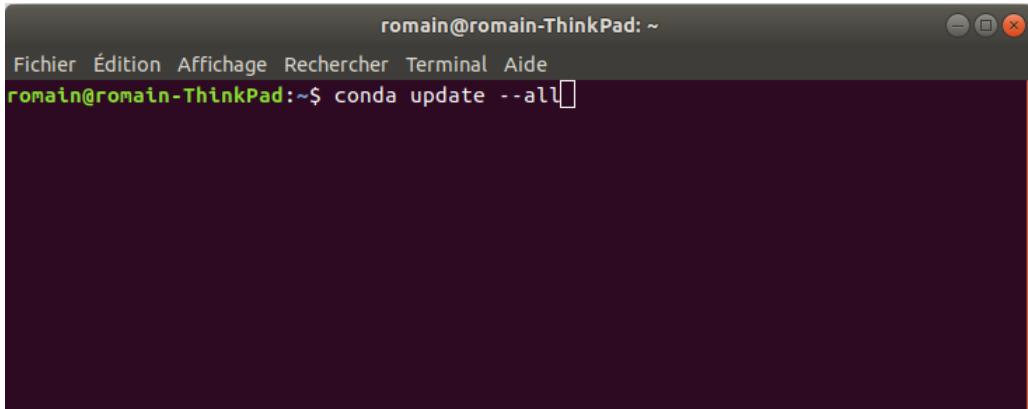


Figure 1.11: Updating Anaconda

Finally, to open Spyder or Jupyter Notebook you can use the Anaconda navigator. To access it, open a terminal and execute the command:

```
anaconda-navigator
```

This opens the navigator (Figure 1.12). You can then launch Jupyter Notebook or Spyder by pressing the corresponding Launch buttons in the navigator.

You may also directly start Jupyter or Spyder from the terminal by executing the commands:

```
jupyter-notebook
```

or:

```
spyder
```

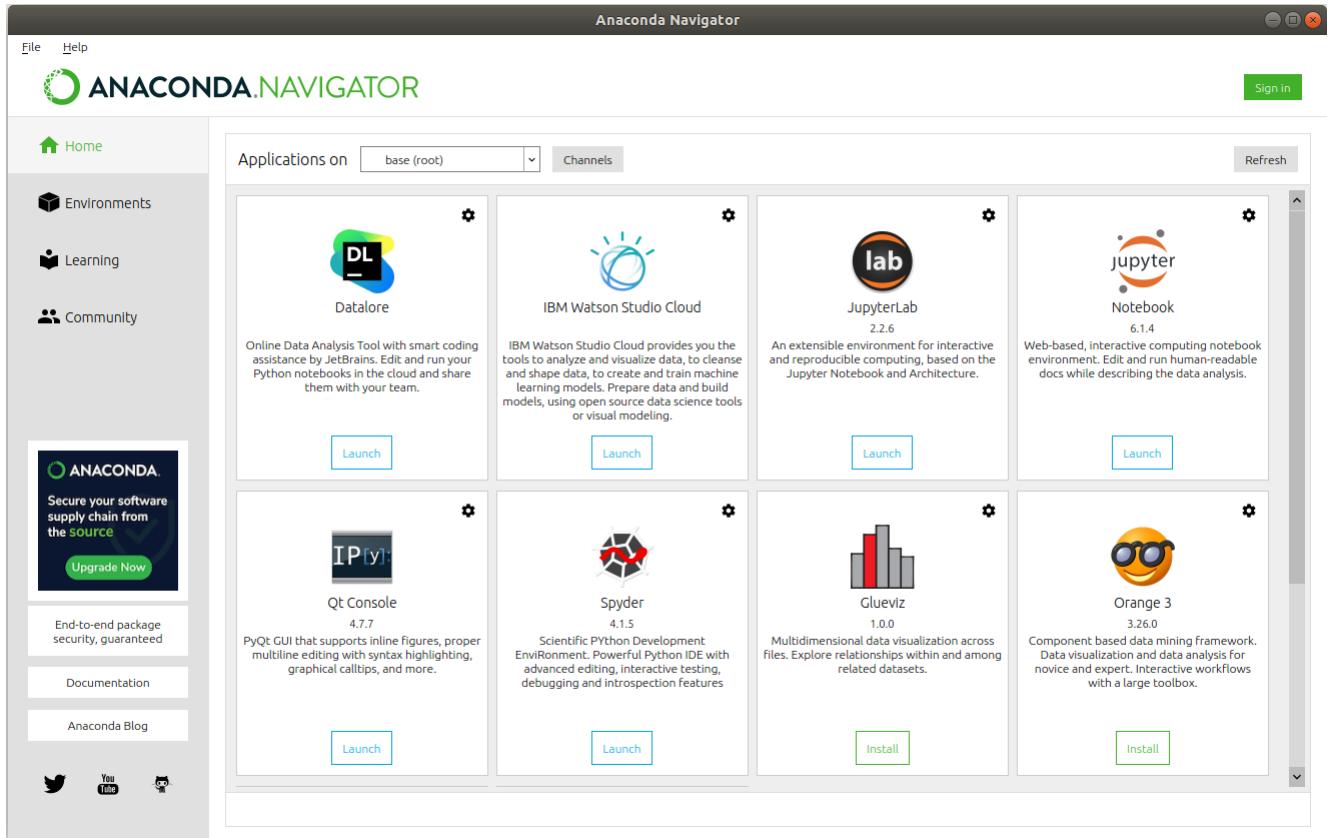


Figure 1.12: The Anaconda navigator

1.5 Alexandria: local installation

There are two options to install Alexandria: a local install that consists in creating a local copy of the folder containing the programmes for each of your projects, or a permanent install that installs the programme on your system once for all. For beginners, it may be easier to use the local installation as it is straightforward and more intuitive (one copy of the application is created for each project).

To proceed to a local install, you first need to recover the folder containing the Alexandria programmes. There are two possibilities to do so. You can go to the [Alexandria website](#), navigate to Downloads on the left menu, and in the Toolbox section click on the link for the Python edition of the software (Figure 1.13). This will download a zip file containing the toolbox programme folder.

Alternatively, you can visit the [Github page](#) of the project, click on the alexandria-python repo (it is a public repository), then click on (Figure 1.14):

Code -> Download ZIP

In both cases, unzip the ZIP archive to obtain a folder named "alexandria-python". This is your local install folder that contains all the programmes to run Alexandria. This folder also constitutes the basis of your project folder (see section 3.1), so you can rename it the way you want to match your project name and move it to any directory you wish. For instance, you may rename the folder "my_project" and place it in D:\my_project (Figure 1.15).

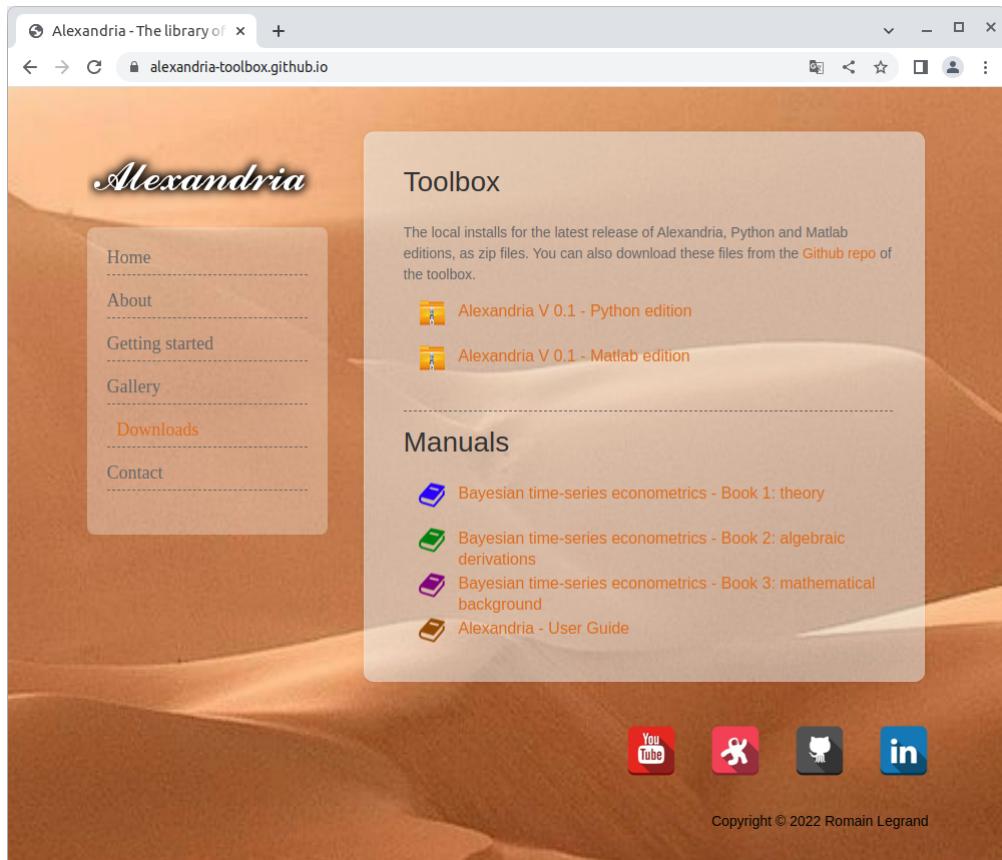


Figure 1.13: The Downloads page of the Alexandria website

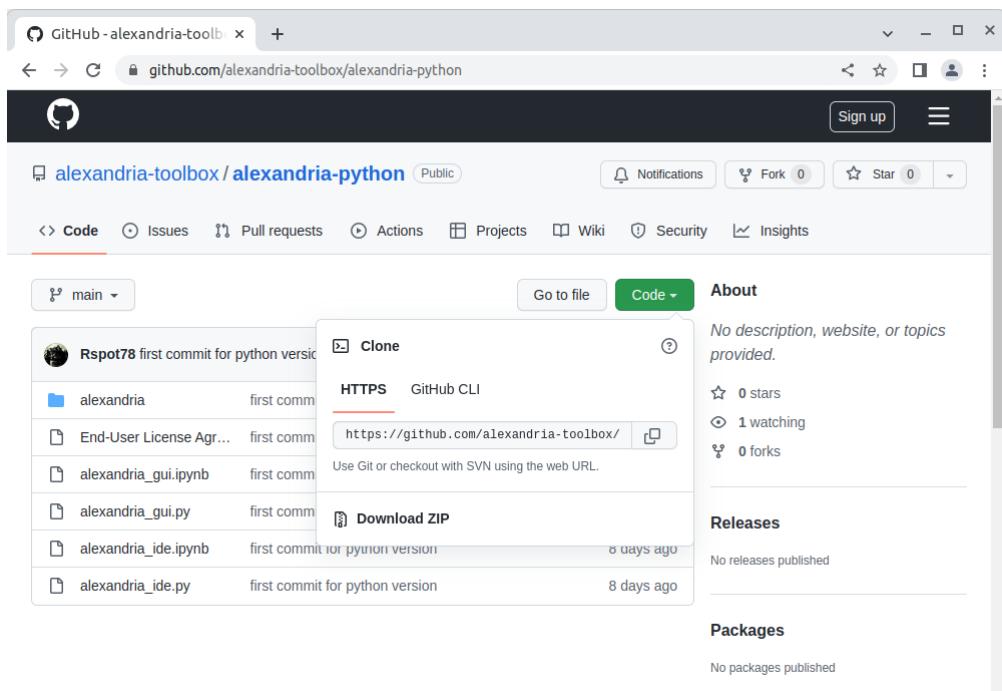


Figure 1.14: The Github repo for the Python version of the toolbox

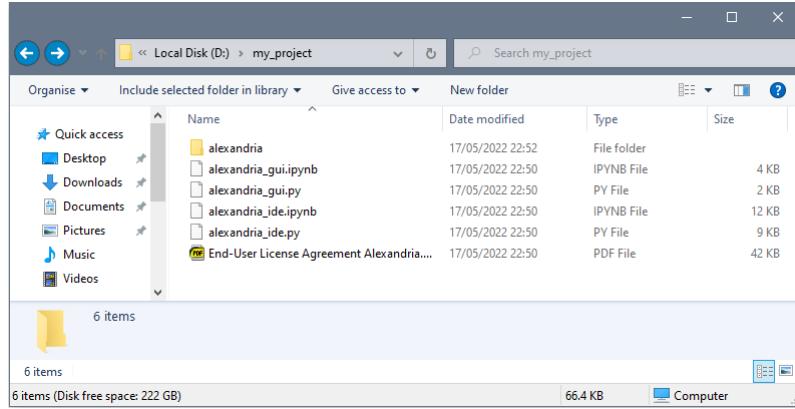


Figure 1.15: The local install folder after renaming

1.6 Alexandria: permanent installation

To install Alexandria permanently, the most convenient option consists in using PyPi. PyPi is a web repository of softwares for the Python programming language. It permits easy installation of third-party projects on personal computers. To install Alexandria from PyPi, first open a terminal. If your operating system is Windows, open the conda terminal (refer to section 1.3). If your operating system is Linux/macOS, open a regular terminal (refer to section 1.4). Then execute the command (Figure 1.16):

```
pip install alexandria-python
```

This will install alexandria on your computer. Note that you need an internet connection to proceed to the installation from PyPi. Also, at any time, you may uninstall Alexandria by using the command:

```
pip uninstall alexandria-python
```

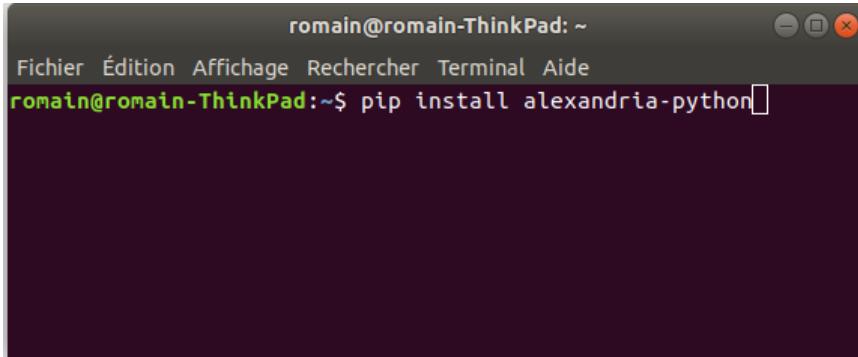


Figure 1.16: Installing Alexandria with pip

Once Alexandria is installed on your system, you can move to the creation of your project folder (section 3.1).

CHAPTER 2

Installing Alexandria, Matlab edition

2.1 Matlab: an overview

Matlab is a numerical software and programming language developed by the firm MathWorks. The name is an abbreviation of "Matrix Laboratory", stressing the primary orientation of the language towards linear algebra applications. Matlab was initially developed by Cleve Moler, a math professor at the University of New Mexico, as a hobby for his students. The programme was initially a simple matrix calculator distributed for free in universities. As popularity started growing, the programme got developed further and was first released as a commercial product in 1984. As of today, Matlab has evolved to include many features beyond linear algebra, including numerical optimization, symbolic algebra, statistical applications, algorithms and graphical visualisation. In 2020, Mathworks claims more than 4 million Matlab users worldwide, mostly from the fields of engineering, science, and economics.

There are several reasons explaining the popularity of Matlab. First, as a software specialised in mathematical applications, Matlab's syntax proves simpler than that of general languages like Python. In fact, its syntax is close to mathematical writing, making the language especially attractive for users with a scientific background. Developing and testing in Matlab is also considerably faster than with other languages, thanks to its simple and concise syntax.

Second, Matlab is powerful. It benefits from – literally – decades of development, and its routines are highly optimized. Matlab can prove several times faster than Python in numerical applications, which represents a strong asset for computationally intensive programmes.

Third, Matlab benefits from Simulink, a graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. Simulink is widely used in the scientific industry and explains much of Matlab's success within the engineering community.

For these reasons Matlab has been extensively used in diverse fields of engineering ranging from signal processing, image treatment and control systems to algorithmic finance, computational biology and econometrics. Recently, Matlab has also tried to capitalize on the recent rise of data science, but it has been facing fierce competition from open source languages such as Python.

On the downside, it should be noted that as a commercial product, Matlab requires the purchase of a license which can prove quite expensive. A standard professional license costs \$2150. For individual users, Matlab proposes a Home license for \$149. Students may benefit from the cheapest option with a student license costing \$49 (these prices may vary depending on which country you live). For more details on licensing, you may consult the [Mathworks webpage on pricing and licensing](#).

2.2 Matlab installation on Windows

Make sure you own a valid Matlab licence and have a Mathworks account activated before you initiate the installation of Matlab. Once this is done, the first step consists in downloading the Matlab installer from

the [Mathworks download webpage](#). Enter your account credentials and complete the steps to obtain the installation programme.

To initiate the installation, double-click on the setup.exe icon within the installation programme. This will open the installation navigator (Figure 2.1). Follow the steps to install Matlab, register your license and create a desktop shortcut. Once this is done, Matlab's installation should be complete. To start Matlab, simply double-click on the desktop shortcut.

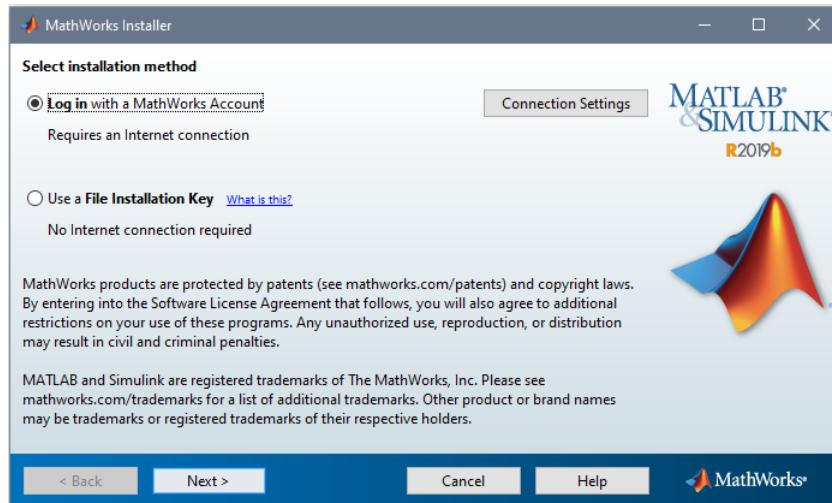


Figure 2.1: The Matlab installation navigator

If you experience issues or wish further details about the installation process, you may consult the [Mathworks help page on installation](#), or the [Mathworks video on Windows installation](#).

2.3 Matlab installation on Linux/macOS

Installing Matlab on Unix operating systems can be tricky, and unfortunately Mathworks does not provide much official installation support. For Linux, Matlab is only available for recent editions of Ubuntu, Debian, Redhat and Suse. On macOS, Matlab can be installed on macOS Catalina, macOS Mojave and macOS High Sierra. Before you start the installation, make sure you own a valid Matlab licence and have a Mathworks account activated. Then the first step consists in downloading the Matlab installer from the [Mathworks download webpage](#). Enter your account credentials and complete the steps to obtain the installation programme.

If you are on Linux, once the download is over, open a terminal (refer to section 1.4 if you are unfamiliar with the terminal) and navigate to the folder containing the download. So for instance if the path to the folder containing the installation file is "~/Downloads", execute the command:

```
cd ~/Downloads
```

Then initiate the installation by executing the command:

```
sudo ./install
```

You may be prompted to enter your username and password. This will open the installation navigator (Figure 2.1). Follow the steps to install Matlab and register your license. Once this is done, Matlab's installation should be complete. To start Matlab, you may then execute the following command in a terminal:

`matlab`

If this fails, it means that Matlab did not generate the required symbolic links during the installation. In this case, Matlab must be started by specifying the full path to the bin folder within the installation directory. For instance, if Matlab is installed in /usr/local/MATLAB/R2020b, then it should be started with the command:

`/usr/local/MATLAB/R2020b/bin/matlab`

If you experience issues during the installation process, you may find some additional information on the [Mathworks help page on installation](#), on this [Mathworks forum](#) and on this [Linux community webpage](#). If you experience issues to start Matlab, you may consult the [Mathworks help page on Matlab start for Linux](#).

If you are on macOS, once the download is over, the installation file should come as a zip archive. Once extracted, you should obtain a file called InstallForMacOSX. Double-click it to initiate the installation. This will open the installation navigator (Figure 2.1). Follow the steps to install Matlab and register your license. Once this is done, Matlab's installation should be complete. To start Matlab, you may either double-click the Matlab icon in your Matlab installation folder, or start it from the terminal by specifying the full path to the bin folder within the installation directory. For instance, if Matlab is installed in /Applications/MATLAB/R2020b, then it should be started with the terminal command:

`/Applications/MATLAB/R2020b/bin/matlab`

If you experience issues during the installation process, you may find some additional information on the [Mathworks help page on installation](#), or on this [support web page](#). If you experience issues to start Matlab, you may consult the [Mathworks help page on Matlab start for macOS](#).

2.4 Alexandria: local installation

There are two options to install Alexandria: a local install that consists in creating a local copy of the folder containing the programmes for each of your projects, or a permanent install that installs the programme on your system once for all. For beginners, it may be easier to use the local installation as it is straightforward and more intuitive (one copy of the application is created for each project).

To proceed to a local install, you first need to recover the folder containing the Alexandria programmes. There are two possibilities to do so. You can go to the [Alexandria website](#), navigate to Downloads on the left menu, and in the Toolbox section click on the link for the Matlab edition of the software (Figure 2.2). This will download a zip file containing the toolbox programme folder.

Alternatively, you can visit the [Github page](#) of the project, click on the alexandria-matlab repo (it is a public repository), then click on (Figure 2.3):

`Code -> Download ZIP`

In both cases, unzip the ZIP archive to obtain a folder named "alexandria-matlab". This is your local install folder that contains all the programmes to run Alexandria. This folder also constitutes the basis of your project folder (see section 3.2), so you can rename it the way you want to match your project name and move it to any directory you wish. For instance, you may rename the folder "my_project" and place it in D:\my_project (Figure 2.4).

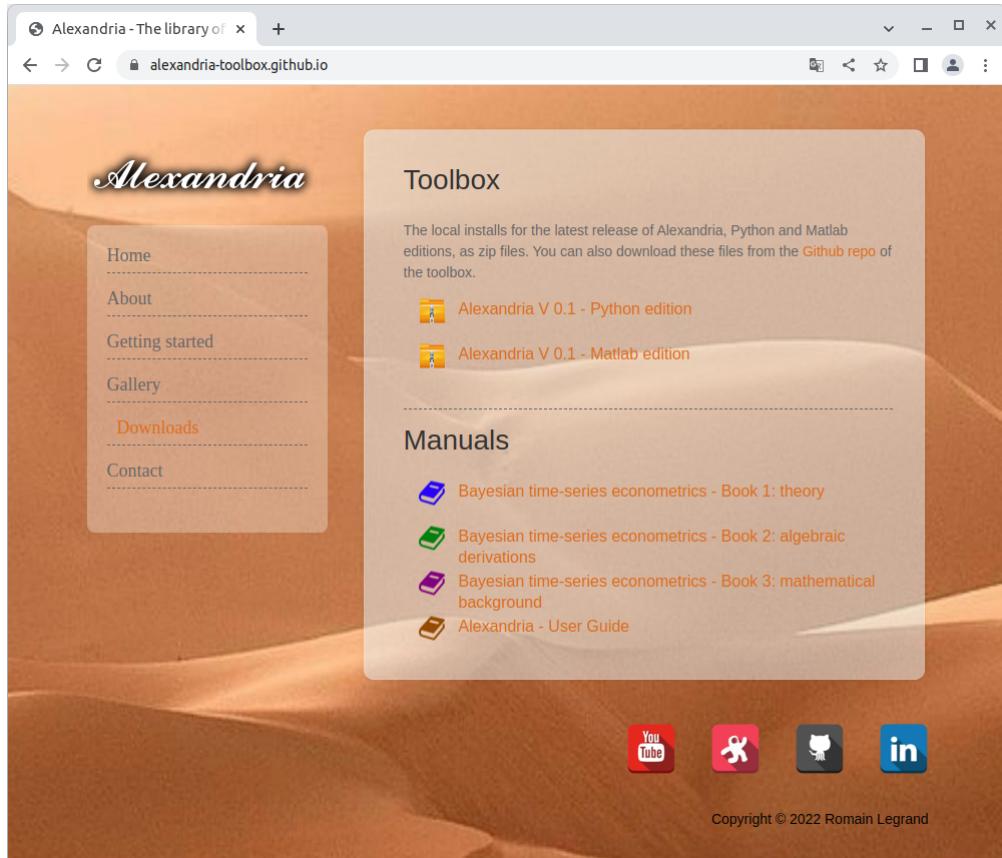


Figure 2.2: The Downloads page of the Alexandria website

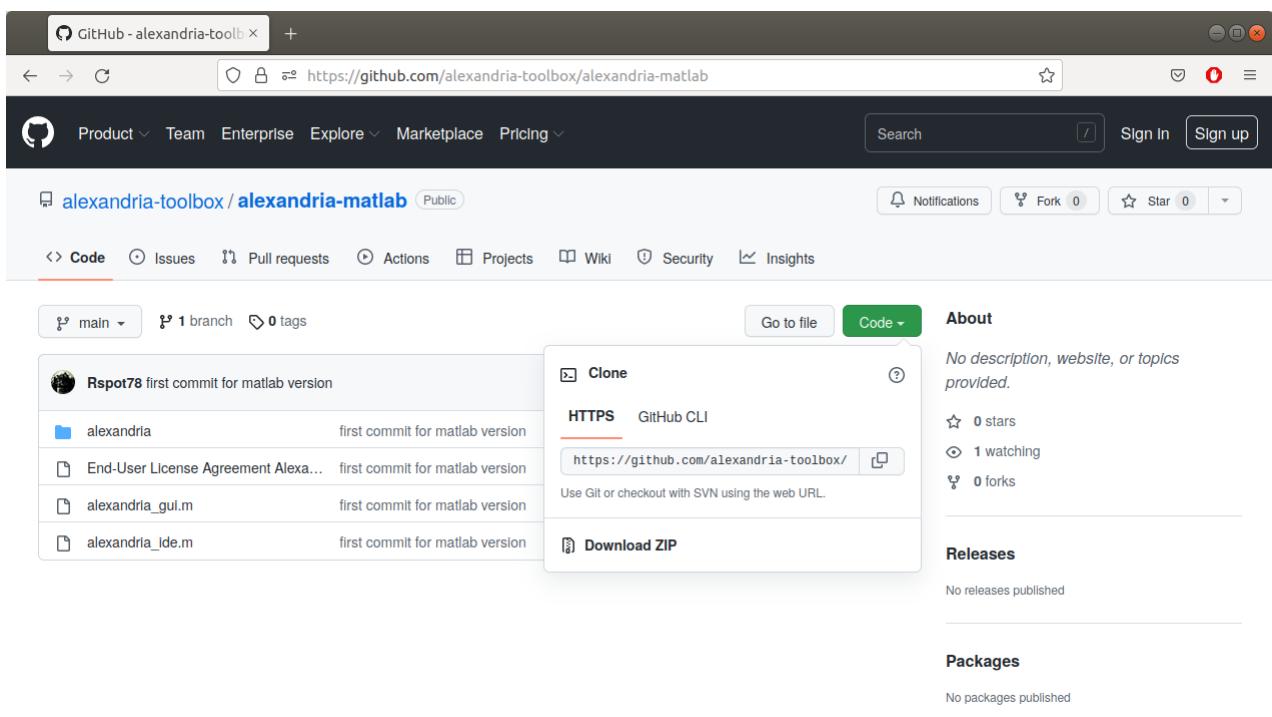


Figure 2.3: The Github repo for the Matlab version of the toolbox

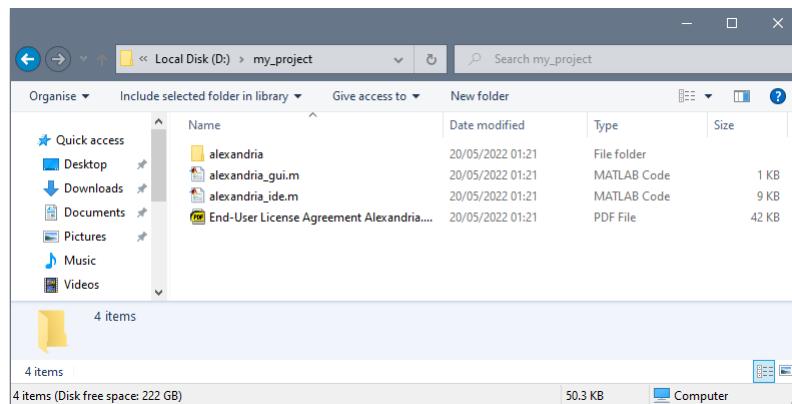


Figure 2.4: The local install folder after renaming

2.5 Alexandria: permanent installation

Installing Alexandria permanently on Matlab is easy. Start Matlab, then in the top menu bar select "Add-Ons", then "Get Add-Ons" (Figure 2.5).

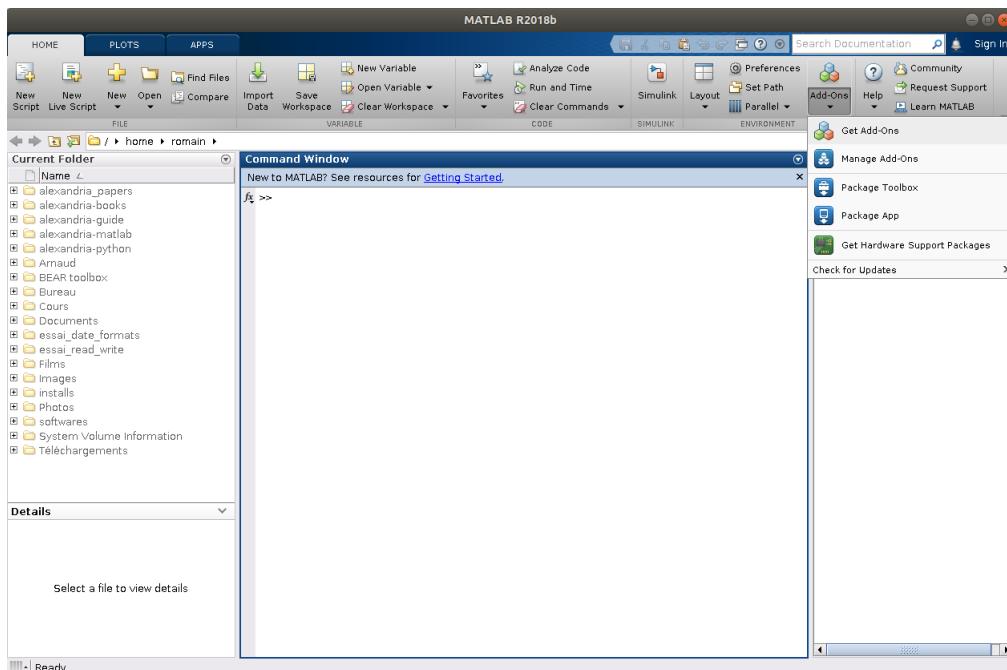


Figure 2.5: Selecting Matlab Add-Ons

This opens the Add-On Explorer. In the top search bar, search for "Alexandria". Select the first choice: Alexandria, by Romain Legrand. Finally, click on the "Add" button on the right to complete the installation (Figure 2.6). This requires a valid Mathworks account. Once the procedure is over, Alexandria is installed permanently on Matlab. At anytime you may uninstall Alexandria by selecting "Add-Ons" in the top menu bar, then "Manage Add-Ons", then select the rollmenu (triple dots) of Alexandria and select "uninstall".

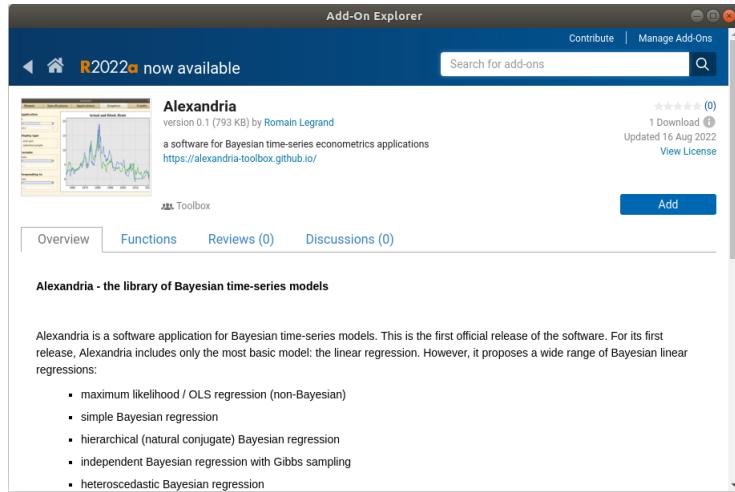


Figure 2.6: Installing the Alexandria toolbox

With Alexandria installed on your system, you can move to the creation of your project folder (section 3.2).

CHAPTER 3

Preparing the project

3.1 Creating the project folder: Python edition

If you opted for a local installation of Alexandria, your project folder is already initiated. It should look like Figure 3.1:

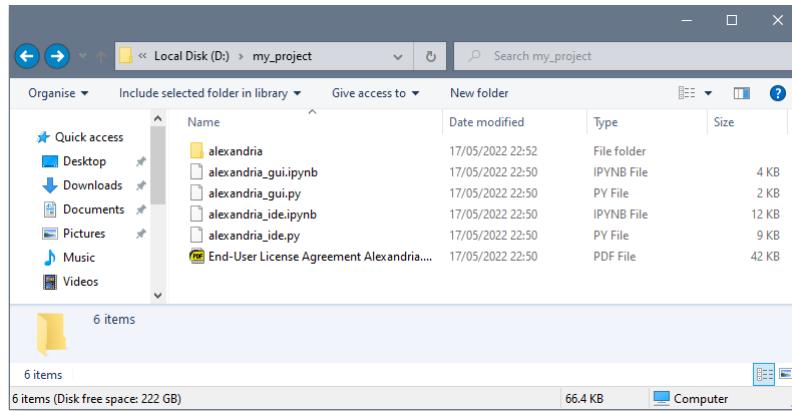


Figure 3.1: Initial project folder with local installation

Make sure you read and agree with the End-User License Agreement. You can then delete the file if you wish (note that are still bound by the Agreement if you do so). Do not delete or modify the alexandria folder and its contents as it contains the software programmes.

What to do with the four remaining files depends on how you plan to use Alexandria. Alexandria can be run either from a graphical user interface (Figure 3.2, panel (a)), or from an integrated development environment (Figure 3.2, panel (b)).

```
Imports
Entrée [ ]: import sys
import os
from warnings import filterwarnings
import matplotlib.pyplot as plt

# clear workspace and console (not to be modified)
filterwarnings('ignore')
plt.close('all')

# initiate user inputs (not to be modified)
user_inputs['tab_1'] = {}
user_inputs['tab_1'][1] = []
user_inputs['tab_1'][2] = []
user_inputs['tab_1'][3] = []

# Model
Entrée [ ]: # model choice (1 = linear regression)
user_inputs['tab_1'][1]['model'] = 1

# endogenous variables, as list of strings (e.g. 'var1', 'var2')
user_inputs['tab_1'][1]['endogenous_variables'] = ['ffrate']

# exogenous variables, as list of strings (e.g. 'var1', 'var2'); leave as empty list [] if no exogenous
user_inputs['tab_1'][1]['exogenous_variables'] = ['rgdp']

# data frequency (1: cross-sectional/undated, 2: yearly, 3: quarterly, 4: monthly, 5: weekly, 6: daily)
user_inputs['tab_1'][1]['frequency'] = 1

# data sample, start and end periods, as list of timestamps strings (e.g. ['1990-03-31', '2020-12-31']) or periods (e.g. '1990-01-01/2018-10-01')
user_inputs['tab_1'][1]['sample'] = ['1990-03-31', '2020-12-31']

# path to project folder, as string (e.g. '/home/ronalh/my_project')
user_inputs['tab_1'][1]['project_path'] = '/home/ronalh/my_project'

# name of data file, as string (e.g., 'data.csv')
user_inputs['tab_1'][1]['data_file'] = 'data.csv'
```

Figure 3.2: Graphical User Interface and Integrated Development Environment

The graphical user interface (GUI) is a simple graphical window in which the user inputs the model parameters. It is recommended for users with limited programming experience. The integrated development environment (IDE) is a python script in which the user enters manually the model information directly as code. It is recommended for users with some programming experience. Additional information on the GUI and IDE can be found in chapters 4 and 5.

If you plan to use the GUI, then keep either the file alexandria_gui.ipynb for a use in Jupyter Notebook, or the file alexandria_gui.py for a use in Spyder. If you plan instead to use the IDE, then keep either the file alexandria_ide.ipynb for a use in Jupyter Notebook, or the file alexandria_ide.py for a use in Spyder. Whatever your choice, the other three files can be deleted. So for instance if you plan to use Alexandria with the GUI in Jupyter notebook (the recommended choice for novice users), your project folder will look like Figure 3.3:

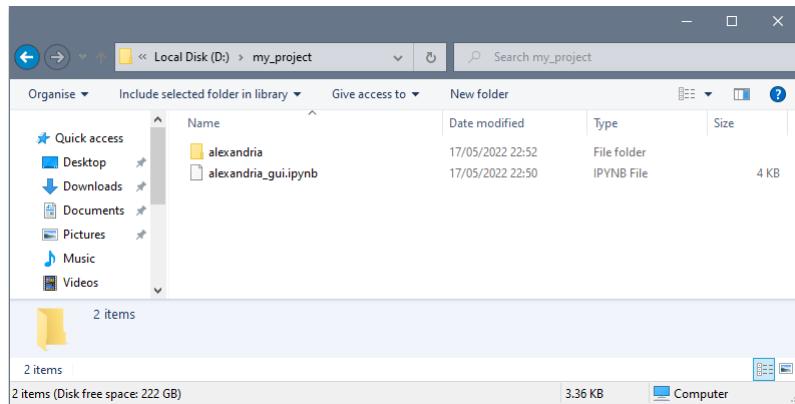


Figure 3.3: Updated project folder with local installation

If you opted for a permanent installation of Alexandria, you don't have a project folder yet. So, create a new folder that will contain your project, place it in any convenient directory and call it as you wish. It is not necessary to copy all the programmes for the software as for a local install since they are permanently installed on your system. It is however recommended for convenience that you include a copy of one of the four files alexandria_gui.ipynb, alexandria_gui.py, alexandria_ide.py or alexandria_ide.ipynb in your project folder, depending on how you plan to use Alexandria. Please refer to section 1.5 to see how you can download these files.

So assume for instance that you create a project folder called my_project and place it in D:\my_project. You then create in it a copy of the file alexandria_gui.ipynb to use Alexandria with the GUI. Your initial project folder should then look like Figure 3.4:

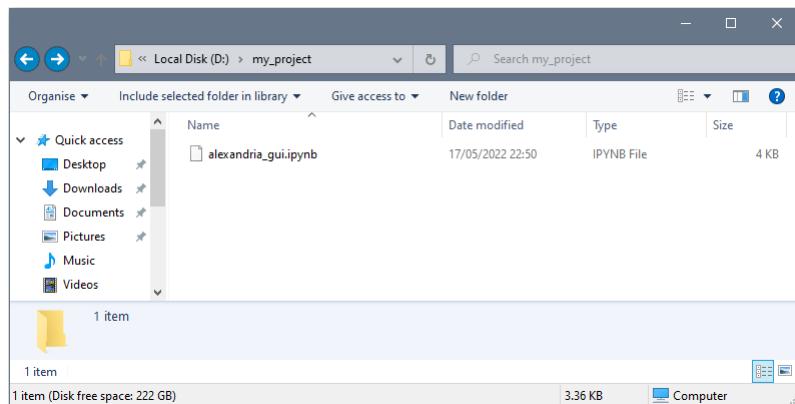


Figure 3.4: Project folder with permanent installation

3.2 Creating the project folder: Matlab edition

If you opted for a local installation of Alexandria, your project folder is already initiated. It should look like Figure 3.5:

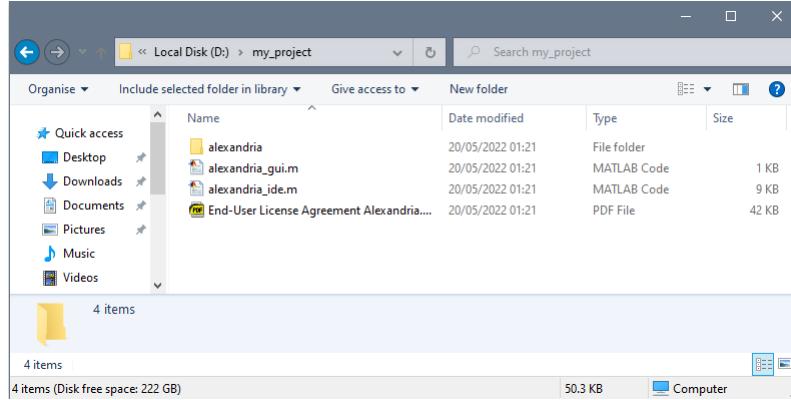
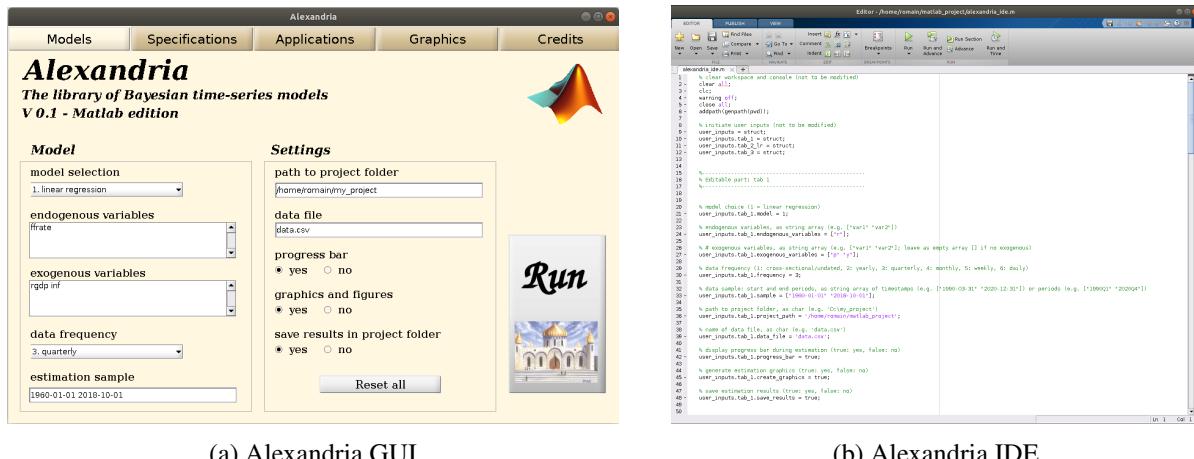


Figure 3.5: Initial project folder with local installation

Make sure you read and agree with the End-User License Agreement. You can then delete the file if you wish (note that are still bound by the Agreement if you do so). Do not delete or modify the alexandria folder and its contents as it contains the software programmes.

What to do with the two remaining files depends on how you plan to use Alexandria. Alexandria can be run either from a graphical user interface (Figure 3.2, panel (a)), or from an integrated development environment (Figure 3.2, panel (b)).



(a) Alexandria GUI

(b) Alexandria IDE

Figure 3.6: Graphical User Interface and Integrated Development Environment

The graphical user interface (GUI) is a simple graphical window in which the user inputs the model parameters. It is recommended for users with limited programming experience. The integrated development environment (IDE) is a python script in which the user enters manually the model information directly as code. It is recommended for users with some programming experience. Additional information on the GUI and IDE can be found in chapters 4 and 5.

If you plan to use the GUI, then keep the file alexandria_gui.m. If you plan instead to use the IDE, keep the file alexandria_ide.m. Whatever your choice, the other file can be deleted. So for instance if you plan

to use Alexandria with the GUI (the recommended choice for novice users), your project folder will look like Figure 3.7:

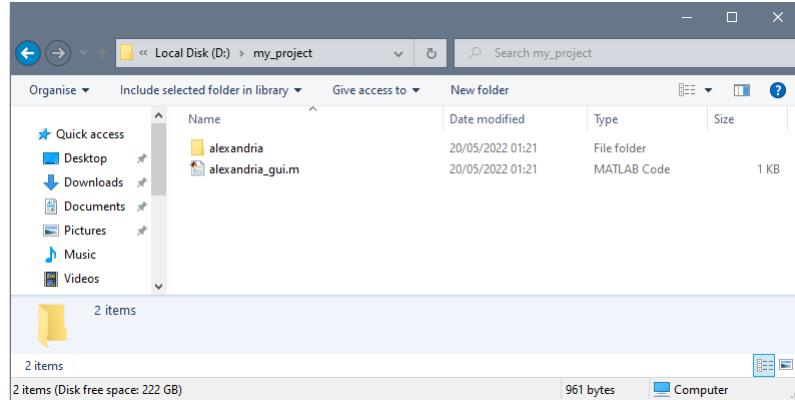


Figure 3.7: Updated project folder with local installation

If you opted for a permanent installation of Alexandria, you don't have a project folder yet. So, create a new folder that will contain your project, place it in any convenient directory and call it as you wish. It is not necessary to copy all the programmes for the software as for a local install since they are permanently installed on your system. You may however include a copy of either alexandria_gui.m or alexandria_ide.m for convenience, though there are ways to work without these files completely. Please refer to section 2.4 to see how you can download these files, and to sections 4.2 and 5.2 for precisions on how to use Alexandria without having to copy these files.

So assume for instance that you create a project folder called my_project and place it in D:\my_project. You then create in it a copy of the file alexandria_gui.m to use Alexandria with the GUI. Your initial project folder should then look like Figure 3.8:

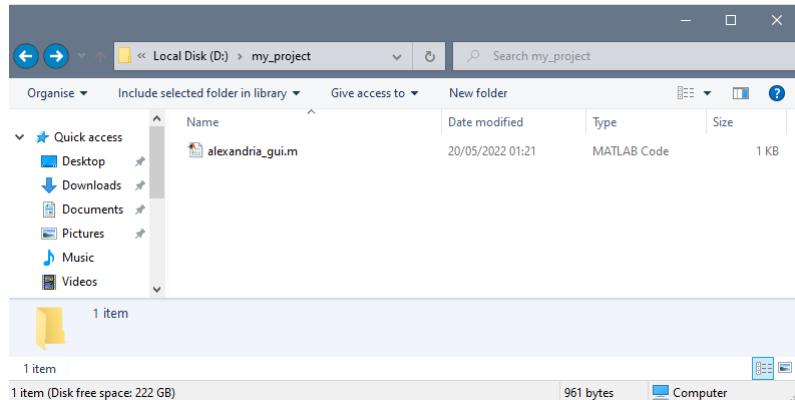


Figure 3.8: Project folder with permanent installation

3.3 Creating the base data file for the model

Once your project folder is initiated, the next step consists in creating the data file that constitutes the base input of your project. This base dataset must be placed directly in your project folder, or, if you want to avoid to have many inputs files in the project folder, in some created subfolder that you may e.g. call inputs. If you place the data file in an input subfolder, your project folder may look like Figure 3.9:

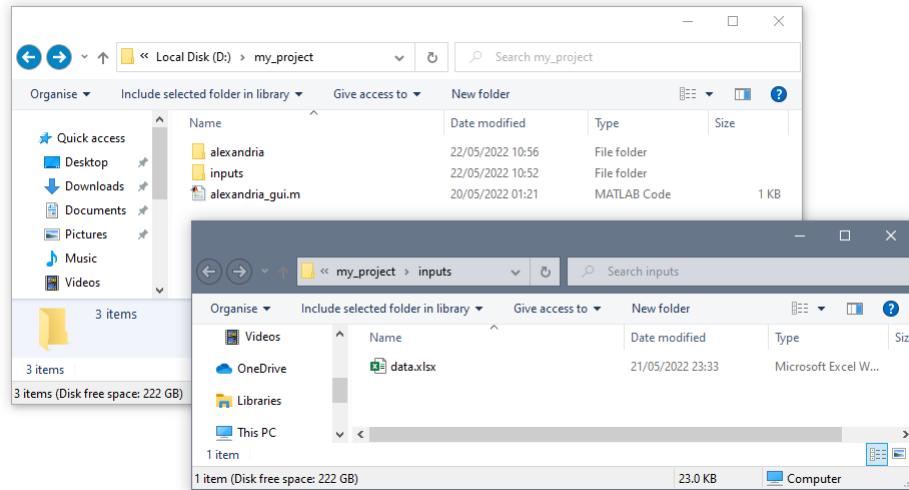


Figure 3.9: Project folder with data file

The data file must contain all the variables that are involved in the estimation of your model: endogenous, exogenous, and possibly others like heteroscedastic variables, if applicable. Do not include constants or trends in the data file, as Alexandra handles these automatically (see section 4.4).

Alexandria accepts three types of files: Excel spreadsheets with either the .xls or .xlsx file extension, and also .csv files which can be easily obtained from any open source spreadsheet such as LibreOffice Calc. A typical dataset will look like Figure 3.10:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1955-01-01	3.50	inf	rgdp											
2	1955-04-01	1.640	-0.520	2.112											
3	1955-07-01	2.180	0.149	3.839											
4	1955-10-01	2.480	0.374	3.784											
5	1956-01-01	2.500	0.373	2.702											
6	1956-04-01	2.710	1.647	2.882											
7	1956-07-01	2.950	1.862	2.098											
8	1956-10-01	2.940	2.828	3.047											
9	1957-01-01	2.960	3.607	2.920											
10	1957-04-01	3.000	3.536	1.870											
11	1957-07-01	3.470	3.547	2.012											
12	1957-10-01	2.980	3.040	0.084											
13	1958-01-01	1.200	3.625	-3.358											
14	1958-04-01	0.930	2.846	-3.590											
15	1958-07-01	1.760	2.083	-2.245											
16	1958-10-01	2.420	1.756	-0.894											
17	1959-01-01	2.800	0.346	0.060											
18	1959-04-01	3.390	0.692	1.333											
19	1959-07-01	3.760	1.176	0.386											
20	1959-10-01	3.990	1.519	-0.336											
21	1960-01-01	5.840	1.519	0.867											
22	1960-04-01	3.320	1.718	-0.678											
23	1960-07-01	2.600	1.231	-1.183											
24	1960-10-01	1.980	1.360	-3.403											
25	1961-01-01	2.020	1.462	-3.703											
26	1961-04-01	1.730	0.777	-2.994											
27	1961-07-01	1.880	1.250	-2.099											
28	1961-10-01	2.330	0.671	-1.154											

Figure 3.10: Project spreadsheet in Excel format

The spreadsheet is organised in a simple way: the first column contains the dates labels, while the first row is used for the names of the variables. The rest of the spreadsheet contains the numerical values. Consider now these elements in details.

- **Variable names**

The first row of the spreadsheet contains the labels of the variables. It must start in cell B1 and expand on the right, as in Figure 3.9. Names can contain letters, digits and underscores, but no spaces or special characters. They cannot start with a digit. Also, it is recommended to keep names short (less than 15 characters) to avoid minor display issues with the toolbox.

- **Date labels**

The first column of the spreadsheet contains the date labels. It must start in cell A2 and develop downward, as in Figure 3.9. Alexandria accepts six different date formats: annual, quarterly, monthly, weekly, daily, and if none of these applies, a cross-section/undated format. Except for the undated format, the dates can be expressed either in international date format (as in Figure 3.9), or in periodic format. Precisely, the different formats must be specified as follows:

annual data: dates can be specified either in international date format of the form yyyy-mm-dd, or in periodic format of the form yyyy. For instance:

international date format: 2000-12-31, 2001-12-31, 2002-12-31...

periodic format: 2000, 2001, 2002...

It does not matter which day is chosen within the year for the international date format, as long as there is only one observation per year.

quarterly data: dates can be specified either in international date format of the form yyyy-mm-dd, or in periodic format of the form yyyy + Q + quarter. For instance:

international date format: 2000-03-31, 2000-06-31, 2000-09-31...

periodic format: 2000Q1, 2000Q2, 2000Q3...

The format is case-sensitive so that 2000Q1 is valid, while 2000q1 isn't. It does not matter which day is chosen within the quarter for the international date format, as long as there is only one observation per quarter.

monthly data: dates can be specified either in international date format of the form yyyy-mm-dd, or in periodic format of the form yyyy + M + month. For instance:

international date format: 2000-01-31, 2000-02-28, 2000-03-31...

periodic format: 2000M1, 2000M2, 2000M3...

The format is case-sensitive so that 2000M1 is valid, while 2000m1 isn't. It does not matter which day is chosen within the month for the international date format, as long as there is only one observation per month.

weekly data: dates can be specified either in international date format of the form yyyy-mm-dd, or in periodic format of the form yyyy + W + week. For instance:

international date format: 2000-01-07, 2000-01-14, 2000-01-21...

periodic format: 2000W1, 2000W2, 2000W3...

The format is case-sensitive so that 2000W1 is valid, while 2000w1 isn't. It does not matter which day is chosen within the week for the international date format, as long as there is only one observation per week. Also, any number of weeks is acceptable for each year as long as it is at most 53.

daily data: dates can be specified either in international date format of the form yyyy-mm-dd, or in periodic format of the form yyyy + D + day. For instance:

international date format: 2000-01-03, 2000-01-04, 2000-01-05...

periodic format: 2000D3, 2000D4, 2000D5...

The format is case-sensitive so that 2000D1 is valid, while 2000d1 isn't. Alexandria uses calendar days, not business days. hence December 31, 2000 is 2000D366, not 2000D261. Your dataset doesn't need to include all the days in the year, and any day can be missing. This will typically happen for instance if you consider business days only so that weekends are ignored.

cross-sectional/undated data: observations must be labelled as integer values. For instance:

1, 2, 3, 4 ...

The list does not have to start at 1, though that is probably the option that makes most sense.

For consistency reasons Alexandria always works with end-of-period dates. If the dataset does not follow this convention, Alexandria will automatically offset the dates to match this format. So for instance if you use quarterly data and uses the date 1960-01-01 for the first quarter of 1960 (as in Figure 3.10), Alexandria will shift it to 1960-03-31. This mostly impacts the spreadsheet/graphical outputs of the toolbox and has no consequence on estimation.

● Data

The remainder of the spreadsheet contains the data itself. The dataset is a rectangular array corresponding to the dates for the rows and to the variables for the columns, as shown in Figure 3.10. The data has to be numerical and may not include missing values.

3.4 Other datafiles: forecasts

If you run forecasts as an application for your model, you need to provide additional data in a separate file. To do so, create a new spreadsheet and give it the name you want (e.g. data_forecast). Similarly to the base data file, the prediction data file can be of type .xlsx, .xls, or .csv. Also, you may place the file either in the project folder directly, or in subfolder if convenient. Assuming you place the file in a subfolder called inputs, your project folder looks like Figure 3.11:

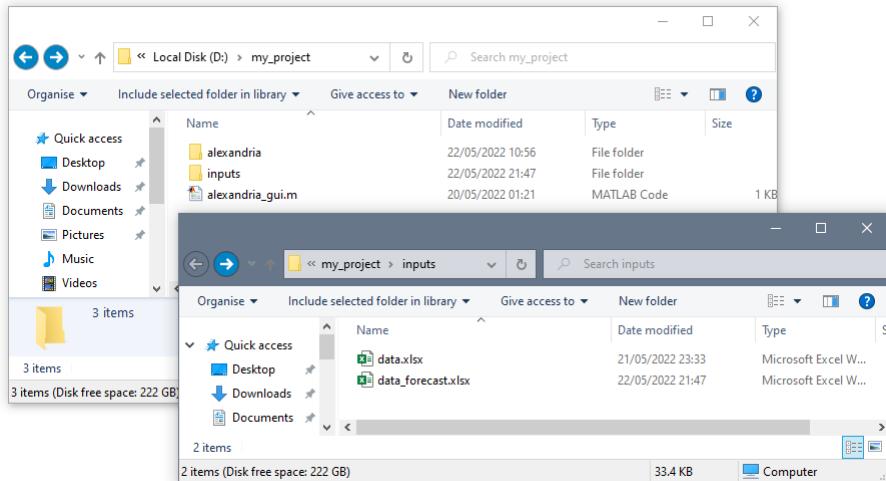


Figure 3.11: Project spreadsheet in Excel format

A typical forecast data file will look like Figure 3.12:

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		ffrate	inf	rgdp									
2	1	1.35	-0.51987	2.112337									
3	2	1.64	-0.85375	3.105948									
4	3	2.18	0.149198	3.839413									
5	4	2.48	0.373552	3.78376									
6	5	2.5	0.373274	2.702129									
7	6	2.71	1.647323	2.881877									
8	7	2.95	1.862197	2.097771									
9	8	2.94	2.828433	3.046828									
10	9	2.96	3.607289	2.919538									
11	10	3	3.535912	1.870385									
12	11	3.47	3.546618	2.012335									
13	12	2.98	3.040174	0.08377									
14	13	1.2	3.625269	-3.35793									
15	14	0.93	2.845962	-3.59013									
16	15	1.76	2.083333	-2.24496									
17	16	2.42	1.756235	-0.89351									
18	17	2.8	0.34638	0.060262									
19	18	3.39	0.691802	1.333279									
20	19	3.76	1.176064	0.385715									
21													
22													

Figure 3.12: Project spreadsheet in Excel format

The file is overall similar to the base datafile introduced in section 3.3, with a few differences. First, no dates need to be specified on the left columns. This is because when producing forecasts, Alexandria will automatically generate prediction dates as the follow-up of in-sample dates. It is therefore irrelevant how dates are specified in the prediction files as they will be ignored. For simplicity, it is recommended to use simple integer values, as in Figure 3.12.

Second, not all the model variables need to be provided in the forecast file. Exogenous variables need to be provided since they are by definition exogenously supplied. The only exceptions are constants and trends for which no values are needed since they are handled automatically. For any other exogenous variable, not providing value will result in an error. Endogenous variables may or may not be provided, depending on whether forecast evaluation criteria are selected (see section 4.6 for more details on forecast evaluation criteria). Providing values for endogenous variables will produce counterfactuals that will be used for the computation of forecast evaluation criteria. Selecting forecast evaluation criteria but not providing endogenous variable values will result in an error. If forecast evaluation criteria are not selected, endogenous variable need not be provided and can be omitted in the forecast data file.

Running Alexandria from the Graphical user Interface

4.1 Launching the interface: Python edition

If you have opted for a local installation of Alexandria, your project folder must contain either the file alexandria_gui.ipynb (for a use in Jupyter Notebook) or the file alexandria_gui.py (for a use in Spyder). If you want to use Alexandria in Jupyter, start Jupyter Notebook, and in the explorer navigate to the project folder, then click on the file alexandria_gui.ipynb. This will open the notebook in a web browser. To start the Graphical Interface, move to the top menu bar (Figure 4.1) and click on:

Kernel -> Restart & Run All

This will start the Graphical User Interface.

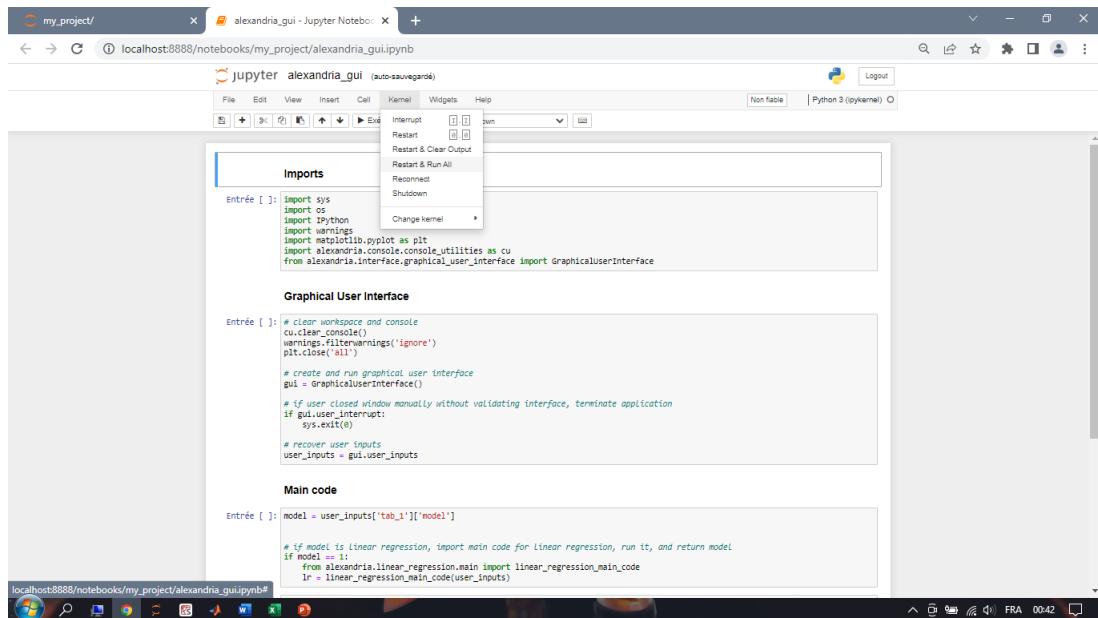


Figure 4.1: Starting the Graphical Interface from Jupyter Notebook

If instead you prefer to start Alexandria from a simple python script, open Spyder, then on the top menu bar click:

File -> Open

This will open a navigator. Navigate to your project folder and click the file alexandria_gui.py. This opens the python script in your Spyder window. To start the Graphical Interface, go again for the top menu bar of Spyder (Figure 4.2) and click:

Run -> Run

This will start the Graphical User Interface.

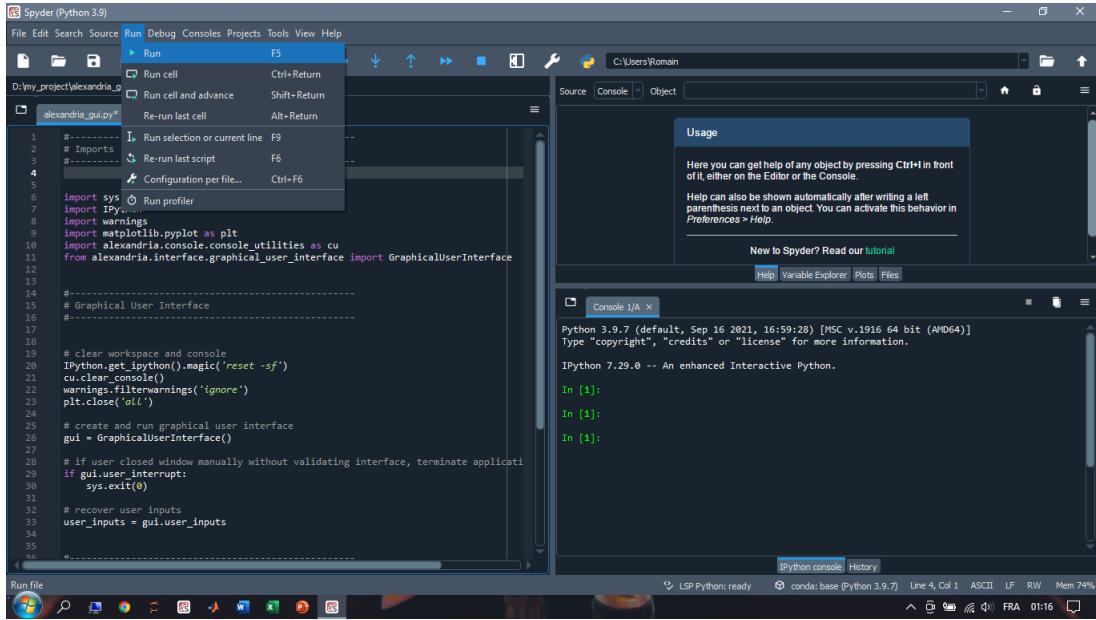


Figure 4.2: Starting the Graphical Interface from Spyder

If you opted for a permanent installation, it is recommended that you proceed the same way. However, in this case there exists an alternative way to start the Graphical Interface without using any of the files. You may just execute the following command in Jupyter or Spyder:

```
from alexandria import alexandria_gui
```

Importing the module will also execute it, and so the command is equivalent to running the script `alexandria_gui.py`. Note that this way is considered somewhat hacky and is not recommended.

4.2 Launching the interface: Matlab edition

If you have opted for a local installation of Alexandria, your project folder should normally contain the file `alexandria_gui.m`. If that is the case, start Matlab, then on the current folder explorer (usually located on the left side of the Matlab window) navigate to your project file and double-click on `alexandria_gui.m`. This should open the file in a new window. To start the Graphical Interface, move to the top menu bar (Figure 4.3) and click on:

Run -> Run: `alexandria_gui`

This will start the Graphical User Interface. You may also simply click on the green triangle arrow above the Run button to run the script directly.

If you opted for a permanent installation, you can proceed the same way. However, it is also possible to start the interface trivially and without any script in this case. Simply execute the following command in the Matlab console:

`alexandria_gui`

This will launch the Graphical User Interface.

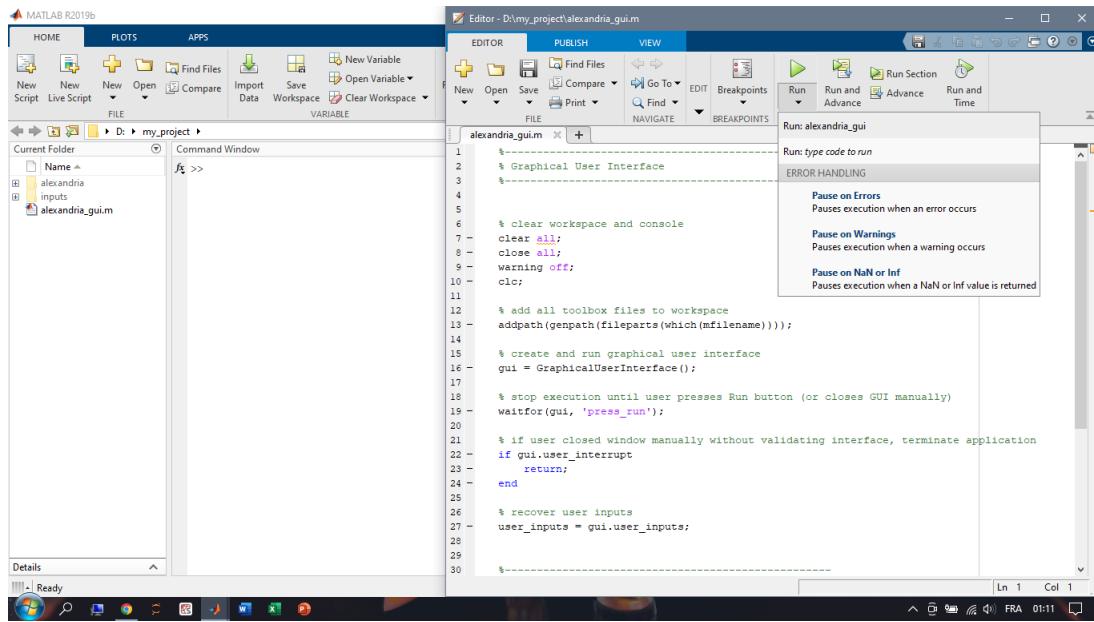


Figure 4.3: Starting the Graphical Interface from Matlab

4.3 Interface: tab 1

The first tab of the Graphical User Interface is depicted in Figure 4.4. The figure shows the interface for the Python edition of Alexandria, but the Matlab version is absolutely similar. Tab 1 is used for general model specification, data source and estimation options.

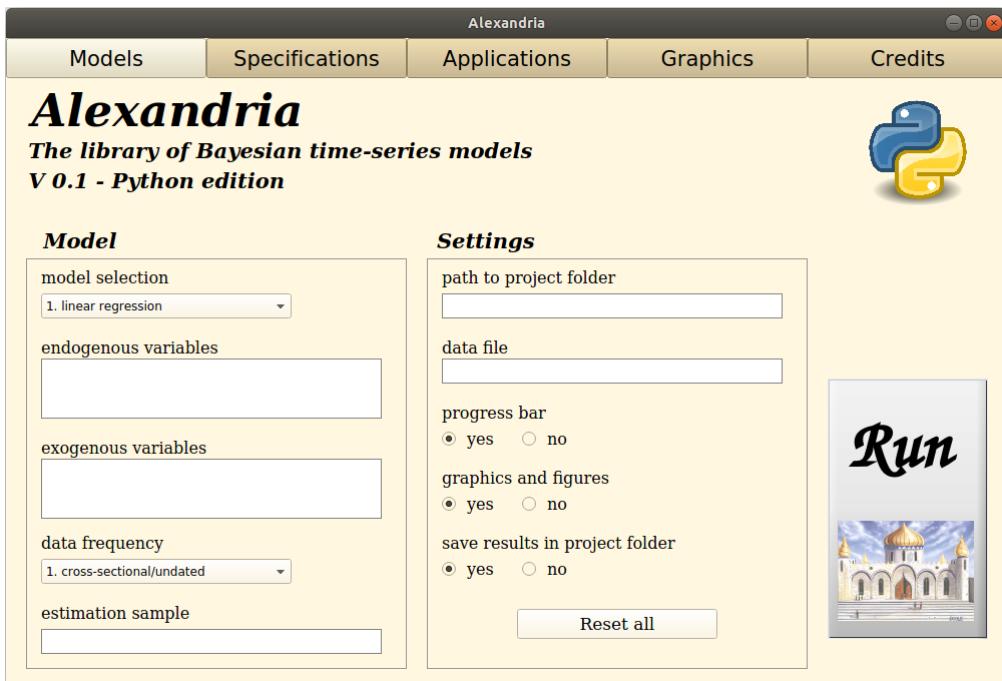


Figure 4.4: Tab 1 of the Graphical User Interface

Tab 1 specifies the following elements:

- **Model, model selection:** the general model you want to estimate. So far Alexandria only proposes the linear regression, but other models will be available as the toolbox develops.
- **Model, endogenous variables:** the list of endogenous variables for the model, separated by a space.
for instance: endo1 endo2 endo3
- **Model, exogenous variables:** the list of exogenous variables for the model, separated by a space.
for instance: exo1 exo2 exo3
- **Model, data frequency:** the dataset frequency. It must correspond to the frequency of the data in the base data file (see section 3.3). The six frequencies available are: annual, quarterly, monthly, weekly, daily, and cross-sectional/undated.
- **Model, estimation sample:** the start and end dates of the estimation sample, separated by a space. The date format must be consistent with that of the the base data file (see section 3.3).
for instance, for quarterly data in international date format: 1960-01-01 2018-10-01
for instance, for quarterly data in periodic format: 1960Q1 2018Q4
- **Settings, path to project folder:** the path to your project folder. The path must be consistent with your OS, and thus use the correct separator (slash or backslash).
for instance, on Windows: D:\my_project
for instance, on Linux: /home/user/my_project
- **Settings, data file:** name of the base datafile in your project folder, with the relevant extension (xls, xlsx or csv).
for instance: data.xls
tip: for a better organisation of the project folder, it is possible to create a subfolder that will contain the base data file, possibly along with other input files. In this case, simply specify the data file as a path with the subfolder containing the file.
for instance, with a subfolder called "inputs": inputs\data.xls
- **Settings, progress bar:** if yes, a progress bar is displayed as the model is being estimated. The progress bar is a useful indicator, but it increases the computational cost of the programme.
- **Settings, graphics and figures:** if yes, estimation plots are produced after estimation is complete. The graphics are saved in a subfolder called "graphics", and displayed in tab 4 of the interface. Please refer to section 6.3 for additional details. The plots constitute very useful outputs, but can take time to be produced for large models.
- **Settings, save results in project folder:** if yes, a number of estimation outputs are saved in text and csv format in a subfolder called "results". The files contain information about the model settings, coefficients and applications. Please refer to section 6.2 for additional details.
- **Settings, Reset all:** by default, Alexandria saves your interface inputs and propose them again in subsequent runs. Pressing the Reset all buttons deletes all previous inputs and resets the interface to default values.
- **Run button:** press this button to start the estimation. This should not be pressed if tabs 2 and 3 have not been completed. It can however constitute a useful shortcut if for instance you run the same model again and don't modify your previous settings.

4.4 Interface: tab 2, linear regression

Figure 4.5 depicts tab 2 of the Graphical User Interface when the selected model is the linear regression.

Figure 4.5: Tab 2 of the Graphical User Interface, linear regression

Tab 2 deals with model-specific information and prior. It contains the following elements:

- **Regression type:** the specific linear regression model you want to estimate. Alexandria proposes six different linear regression models: maximum likelihood, simple Bayesian, hierarchical, independent, heteroscedastic and autocorrelated. Please consult the textbook, sections 9.1 to 9.6, for additional details on the different models.
- **Estimation, iterations:** the number of iterations for the MCMC algorithms, after the burn-in iterations are completed. Applies only to models using MCMC methods.
for instance: 5000
- **Estimation, burn-in:** the number of burn-in iterations for the MCMC algorithms. Applies only to models using MCMC methods.
for instance: 2000
- **Estimation, credibility level:** the credibility level used for the calculation of credibility intervals for the model estimates. Must be comprised between 0 and 1.
for instance: 0.95
- **Hyperparameters, \mathbf{b} :** the prior mean on the regression coefficients β . Can be a single value, in which case the same prior mean applies to all coefficients. Else, must be a list of values separated by a space, one value for each exogenous variable. No values should be provided for potential constants and trends which are handled separately (see below).
for instance: 0, or: 0.2 0.4
- **Hyperparameters, \mathbf{V} :** the prior variance on the regression coefficients β . Can be a single value, in which case the same prior variance applies to all coefficients. Else, must be a list of values separated by

a space, one value for each exogenous variable. No values should be provided for potential constants and trends which are handled separately (see below).

for instance: 1, or: 0.5 1

■ **Hyperparameters, α :** the prior shape on the residual variance σ . Must be a positive scalar.

for instance: 0.0001

■ **Hyperparameters, δ :** the prior scale on the residual variance σ . Must be a positive scalar.

for instance: 0.0001

■ **Hyperparameters, g :** the prior mean on the heteroscedastic coefficients γ . Can be a single value, in which case the same prior mean applies to all coefficients. Else, must be a list of values separated by a space, one value for each heteroscedastic variable.

for instance: 0, or 0 0

■ **Hyperparameters, Q :** the prior variance on the heteroscedastic coefficients γ . Can be a single value, in which case the same prior variance applies to all coefficients. Else, must be a list of values separated by a space, one value for each heteroscedastic variable.

for instance: 100, or 100 100

■ **Hyperparameters, τ :** the variance of the random walk kernel for the Metropolis-Hastings algorithm. Should be set (by trial and error) to generate an acceptance rate of 20%-30%.

for instance: 0.001

■ **Hyperparameters, thinning:** if selected, thinning is applied to the Metropolis-Hastings algorithm, which helps to reduce the number of repeated values and produces a finer distribution. This is useful but generates additional calculations (see next parameter).

■ **Hyperparameters, frequency:** thinning frequency: if set to n , only 1 out of n iterations will be retained, the other iterations being discarded. This multiplies by n the total number of iterations in order to maintain constant the final number of draws. It can then prove very costly for large n .

for instance: 10

■ **Hyperparameters, Z variables:** the list of heteroscedastic variables. These regressors must be found in the same base data file as the base model regressors. Can be the same variables as the model exogenous variables, but other variables can be used if relevant.

for instance: htr1 htr2 htr3

■ **Hyperparameters, q:** the order of autocorrelation for the autocorrelated regression. Must be some integer equal of larger than 1.

for instance: 2

■ **Hyperparameters, p:** the prior mean on the autocorrelated coefficients ϕ . Can be a single value, in which case the same prior mean applies to all coefficients. Else, must be a list of values separated by a space, one value for each lag of autocorrelation.

for instance: 0 or 0.9 0

■ **Hyperparameters, H:** the prior variance on the autocorrelated coefficients γ . Can be a single value, in which case the same prior variance applies to all coefficients. Else, must be a list of values separated by a space, one value for each lag of autocorrelation.

for instance: 100 or 100 100

■ **Exogenous, constant:** if selected, adds a constant to the linear regression.

■ **Exogenous, constant, b:** the prior mean on the constant. Must be a scalar.

for instance: 0

- **Exogenous, constant, V:** the prior variance on the constant. Must be a positive scalar.
for instance: 100
- **Exogenous, linear trend:** if selected, adds a linear trend to the linear regression.
- **Exogenous, linear trend, b:** the prior mean on the linear trend. Must be a scalar.
for instance: 0
- **Exogenous, linear trend, V:** the prior variance on the linear trend. Must be a positive scalar.
for instance: 100
- **Exogenous, quadratic trend:** if selected, adds a quadratic trend to the linear regression.
- **Exogenous, quadratic trend, b:** the prior mean on the quadratic trend. Must be a scalar.
for instance: 0
- **Exogenous, quadratic trend, V:** the prior variance on the quadratic trend. Must be a positive scalar.
for instance: 100
- **Options, in-sample fit:** if selected, calculates the in-sample fit of the model, saves the fit estimates in the results folder, and produces fit plots.
- **Options, marginal likelihood:** if selected, calculates the marginal likelihood for the model and displays the value in the console output of the model estimation.
- **Options, hyperparameter optimization:** if selected, runs a numerical optimization procedure to find the hyperparameter values V and δ that maximize the marginal likelihood, and use these values for the priors. Applicable only to the simple Bayesian and hierarchical regressions.
- **Options, optimization type:** if simple, a common prior variance is assumed for all the β coefficients, and the optimizer will search for a single optimal scalar value for V . If full, a different prior variance is assumed for the β coefficients, so that the optimizer will try to find a different optimal value for each V coefficient.

4.5 Interface: tab 3

Tab 3 deals with the applications associated to the estimated model. It is depicted in Figure 4.6. So far Alexandria only proposes the linear regression model, so most applications are unavailable since they are associated to VAR models. However, certain applications remain available for the linear regression.

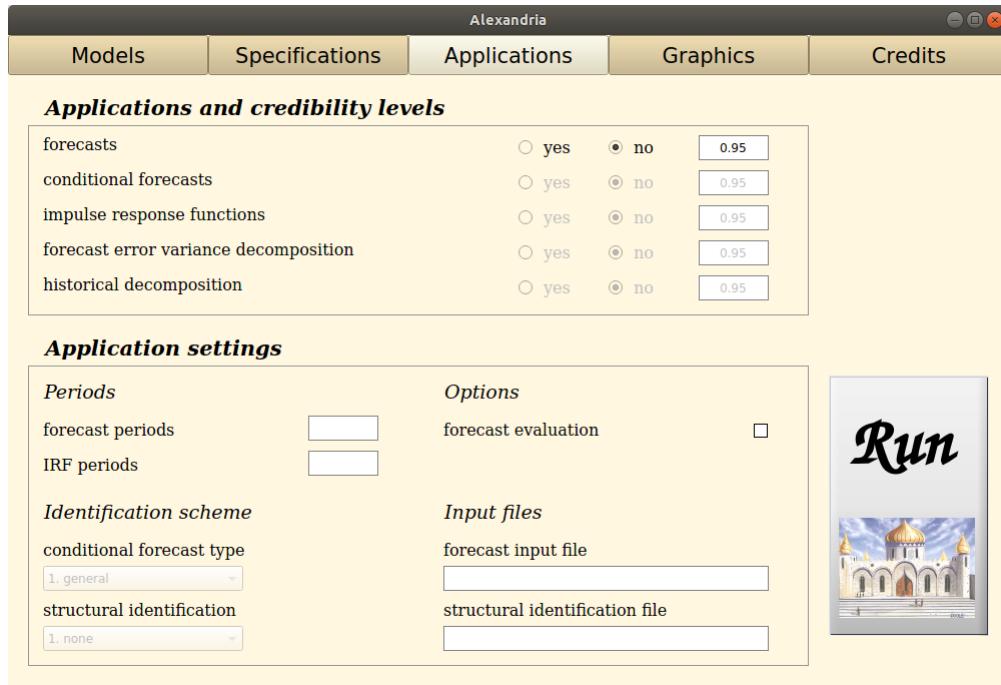


Figure 4.6: Tab 3 of the Graphical User Interface

So far, the following applications are available for the linear regression:

- **Application and credibility levels, forecasts:** if yes, forecasts are estimated for the model. This requires a valid forecast input file to supply exogenous predictors. See section 3.4 for additional details.
- **Application and credibility levels, forecasts, credibility level:** credibility level for the computation of credibility bands around the forecasts. Must be a scalar between 0 and 1.
for instance: 0.95
- **Application settings, forecast evaluation:** if selected, calculates forecast evaluation criteria for the model predictions and displays them in the estimation output. Requires valid counterfactual values in the forecast data file (see section 3.4 for additional details).
- **Application settings, forecast input file:** name of the forecast datafile in your project folder, with the relevant extension (xls, xlsx or csv).
for instance: data_forecast.xls
tip: similar to the base data file, it is possible to place the forecast datafile in a subfolder inside your project folder, and specify the name as a path to the subfolder.
for instance, with a subfolder called "inputs": inputs\data_forecast.xls
- **Run button:** press this button to start the estimation. This should be pressed only when tabs 1, 2 and 3 have all been completed.

4.6 Interface: tab 4

Tab 4 deals with the graphical outputs of the toolbox. It is depicted in Figure 4.7. Prior to model estimation, no graphics are available and thus the interface only displays the image "No graphic to display yet". This tab becomes useful only once model estimation is completed. Please refer to section 6.2 for additional details on how to navigate this tab once graphics are produced.

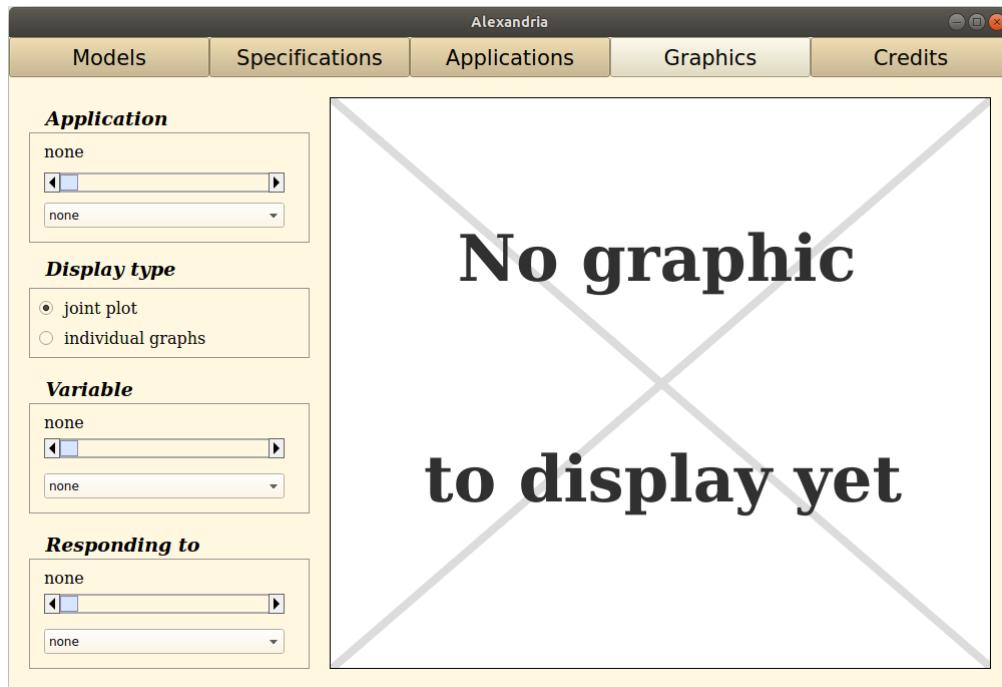


Figure 4.7: Tab 4 of the Graphical User Interface

4.7 Interface: tab 5

Tab 5 of the Graphical User Interface is depicted in Figure 4.8. It does not offer any interaction but offers general information about the toolbox and related social media.



Figure 4.8: Tab 5 of the Graphical User Interface

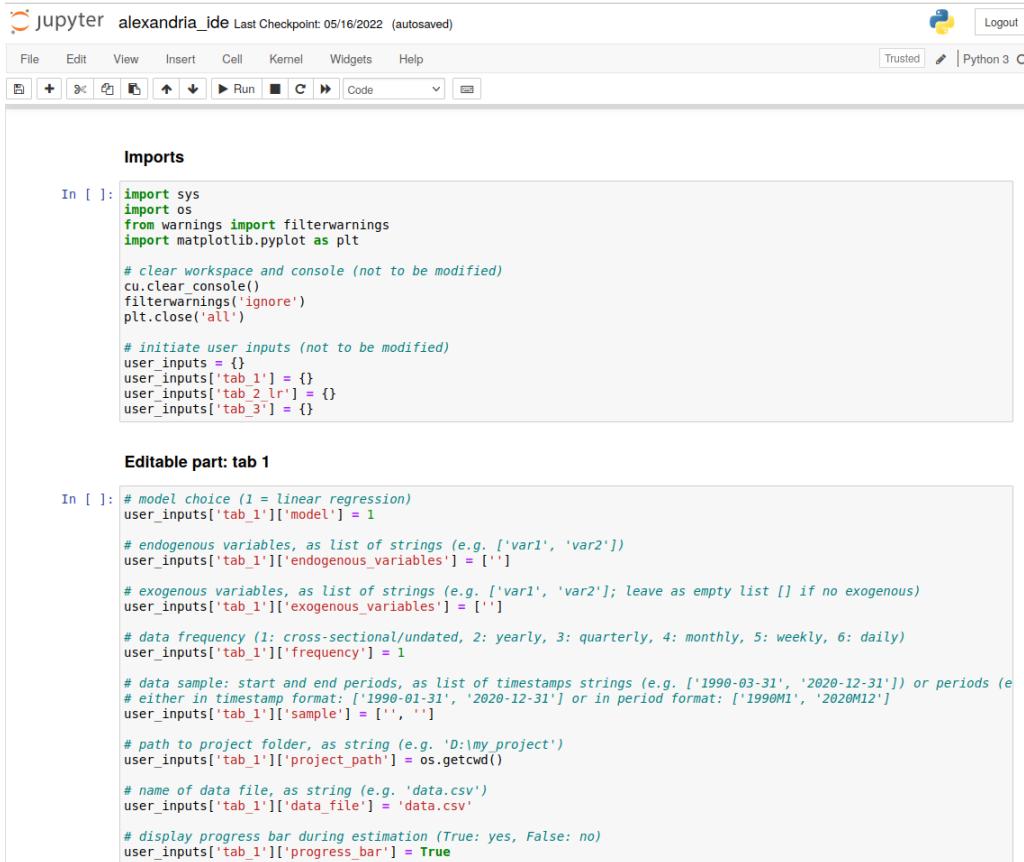
CHAPTER 5

Alternative ways to run Alexandria

5.1 Running Alexandria from the Integrated Development Environment: Python edition

If you don't want to navigate the Graphical User Interface every time you re-estimate a model, you can use a more straightforward way to run the toolbox: the Integrated Development Environment (IDE). Fundamentally, the IDE is a simple Python script that is equivalent to the Graphical user Interface but replaces the interface navigation with direct Python code. Running Alexandria from the IDE then only requires to run the script from Jupyter or Spyder, which can prove much faster than using the Graphical User Interface.

For the Python edition of Alexandria, using the IDE requires that you place in your project folder either the file alexandria_ide.ipynb (for a use in Jupyter Notebook) or the file alexandria_ide.py (for a use in Spyder). It is recommended that you install Alexandria permanently if you want to use the IDE so that you can simply copy and paste the file alexandria_ide.ipynb / alexandria_ide.py in any new project folder you create. Otherwise, you need to proceed to a full local installation, as described in section 1.5.



The screenshot shows a Jupyter Notebook interface with the title bar "jupyter alexandria_ide Last Checkpoint: 05/16/2022 (autosaved)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Run, Code, and Cell Type buttons. A status bar at the bottom right shows "Trusted" and "Python 3".

The notebook contains two code cells:

```
Imports
In [ ]: import sys
import os
from warnings import filterwarnings
import matplotlib.pyplot as plt

# clear workspace and console (not to be modified)
cu.clear_console()
filterwarnings('ignore')
plt.close('all')

# initialize user inputs (not to be modified)
user_inputs = {}
user_inputs['tab_1'] = {}
user_inputs['tab_2_lr'] = {}
user_inputs['tab_3'] = {}

Editable part: tab 1
In [ ]: # model choice (1 = linear regression)
user_inputs['tab_1'][model] = 1

# endogenous variables, as list of strings (e.g. ['var1', 'var2'])
user_inputs['tab_1'][endogenous_variables] = ['']

# exogenous variables, as list of strings (e.g. ['var1', 'var2']; leave as empty list [] if no exogenous)
user_inputs['tab_1'][exogenous_variables] = []

# data frequency (1: cross-sectional/undated, 2: yearly, 3: quarterly, 4: monthly, 5: weekly, 6: daily)
user_inputs['tab_1'][frequency] = 1

# data sample: start and end periods, as list of timestamps strings (e.g. ['1990-03-31', '2020-12-31']) or periods (e
# either in timestamp format: ['1990-01-31', '2020-12-31'] or in period format: ['1990M1', '2020M12'])
user_inputs['tab_1'][sample] = ['', '']

# path to project folder, as string (e.g. 'D:\my_project')
user_inputs['tab_1'][project_path] = os.getcwd()

# name of data file, as string (e.g. 'data.csv')
user_inputs['tab_1'][data_file] = 'data.csv'

# display progress bar during estimation (True: yes, False: no)
user_inputs['tab_1'][progress_bar] = True
```

Figure 5.1: Alexandria IDE on Jupyter Notebook

```

1  #-
2  # Imports
3  #-
4
5
6  import sys
7  import IPython
8  import os
9  from warnings import filterwarnings
10 import matplotlib.pyplot as plt
11 import alexandria.console.console_utilities as cu
12
13 # clear workspace and console (not to be modified)
14 IPython.get_ipython().magic('reset -sf')
15 cu.clear_console()
16 filterwarnings('ignore')
17 plt.close('all')
18
19 # initiate user inputs (not to be modified)
20 user_inputs = {}
21 user_inputs['tab_1'] = {}
22 user_inputs['tab_2_lr'] = {}
23 user_inputs['tab_3'] = {}
24
25
26 #-----#
27 # Editable part: tab 1
28 #-----#
29
30
31 # model choice (1 = linear regression)
32 user_inputs['tab_1'][ 'model' ] = 1
33
34 # endogenous variables, as list of strings (e.g. ['var1', 'var2'])
35 user_inputs['tab_1'][ 'endogenous_variables' ] = [ '' ]
36
37 # exogenous variables, as list of strings (e.g. ['var1', 'var2']; leave as empty list [] if no exogenous)
38 user_inputs['tab_1'][ 'exogenous_variables' ] = [ '' ]
39
40 # data frequency (1: cross-sectional/undated, 2: yearly, 3: quarterly, 4: monthly, 5: weekly, 6: daily)
41 user_inputs['tab_1'][ 'frequency' ] = 1
42
43 # data sample: start and end periods, as list of timestamps strings (e.g. ['1990-03-31', '2020-12-31']) or periods (e.g. ['1990Q1', '2020Q4'])
44 # either in timestamp format: ['1990-01-31', '2020-12-31'] or in period format: ['1990M1', '2020M12']
45 user_inputs['tab_1'][ 'sample' ] = [ '', '' ]
46
47 # path to project folder, as string (e.g. 'D:\my\project')
48 user_inputs['tab_1'][ 'project_path' ] = os.getcwd()
49
50 # name of data file, as string (e.g. 'data.csv')
51 user_inputs['tab_1'][ 'data_file' ] = 'data.csv'

```

Figure 5.2: Alexandria IDE on Spyder

The file `alexandria_ide.ipynb` is shown in Figure 5.1, while `alexandria_ide.py` is depicted in Figure 5.2. As you can see they appear as simple Python scripts, with some parts editable. All you need to do is to edit the editable parts of the script to provide the relevant information about your project. The correspondance with the elements of the Graphical user Interface should be straightforward, but if needed you may refer to chapter 4 for additional details.

In terms of format, the comments provided before each editable lines will guide you to edit the part correctly. Note that even though the format is indicated for each item, using the IDE requires minimal familiarity with Python programming and its syntax. Note also that the code cannot check for improper inputs, so that any misspecified entry will result in an exception during the programme.

Once the file is properly edited, starting the toolbox only requires to execute the script: if needed, refer to section 4.1.

5.2 Running Alexandria from the Integrated Development Environment: Matlab edition

If you don't want to navigate the Graphical User Interface every time you re-estimate a model, you can use a more straightforward way to run the toolbox: the Integrated Development Environment (IDE). Fundamentally, the IDE is a simple Matlab script that is equivalent to the Graphical user Interface but replaces the interface navigation with direct Matlab code. Running Alexandria from the IDE then only requires to run the script from your Matlab console, which can prove much faster than using the Graphical User Interface.

5.2. RUNNING ALEXANDRIA FROM THE INTEGRATED DEVELOPMENT ENVIRONMENT:MATLAB EDITION

For the Matlab edition of Alexandria, there are two different ways to use the IDE. If you installed Alexandria locally, using the IDE requires that you place in your project folder the file alexandria_ide.m. If you installed Alexandria permanently you can proceed the same way, but you can also use the file alexandria_ide.m that is permanently installed on your Matlab. To edit the file, simply execute the following command in the Matlab console:

```
open alexandria_ide
```

You can then work with the opened file as if it was a local alexandria_ide.m file placed in your project folder.

```

Editor - /home/romain/MATLAB Add-Ons/Toolboxes/Alexandria/alexandria_ide.m

EDITOR PUBLISH VIEW
FILE NAVIGATE EDIT BREAKPOINTS RUN
alexandria_ide.m x + [ 1 % clear workspace and console (not to be modified)
2 - clear all;
3 - clc;
4 - warning off;
5 - close all;
6 - addpath(genpath(pwd));
7 -
8 % initiate user inputs (not to be modified)
9 - user_inputs = struct;
10 - user_inputs.tab_1 = struct;
11 - user_inputs.tab_2_lr = struct;
12 - user_inputs.tab_3 = struct;
13 -
14 %-----
15 % Editable part: tab_1
16 %-----
17 %
18 %
19 %
20 % model choice (1 = linear regression)
21 user_inputs.tab_1.model = 1;
22 %
23 % endogenous variables, as string array (e.g. ["var1" "var2"])
24 - user_inputs.tab_1.endogenous_variables = ["*"];
25 %
26 % # exogenous variables, as string array (e.g. ["var1" "var2"]; leave as empty array [] if no exogenous)
27 - user_inputs.tab_1.exogenous_variables = ["*"];
28 %
29 % data frequency (1: cross-sectional/undated, 2: yearly, 3: quarterly, 4: monthly, 5: weekly, 6: daily)
30 - user_inputs.tab_1.frequency = 1;
31 %
32 % data sample: start and end periods, as string array of timestamps (e.g. ["1990-03-31" "2020-12-31"]) or periods (e.g. ["1990Q1" "2020Q4"])
33 - user_inputs.tab_1.sample = ["* *"];
34 %
35 % path to project folder, as char (e.g. 'D:\my_project')
36 - user_inputs.tab_1.project_path = pwd();
37 %
38 % name of data file, as char (e.g. 'data.csv')
39 - user_inputs.tab_1.data_file = '';
40 %
41 % display progress bar during estimation (true: yes, false: no)
42 - user_inputs.tab_1.progress_bar = true;
43 %
44 % generate estimation graphics (true: yes, false: no)
45 - user_inputs.tab_1.create_graphics = true;
46 %
47 % save estimation results (true: yes, false: no)
48 - user_inputs.tab_1.save_results = true;
49 ] 1 Col 1

```

Figure 5.3: Alexandria IDE on Jupyter Notebook

The file alexandria_ide.m is shown in Figure 5.3. As you can see, it appears as a simple Matlab script, with some parts editable. All you need to do is to edit the editable parts of the script to provide the relevant information about your project. The correspondance with the elements of the Graphical user Interface should be straightforward, but if needed you may refer to chapter 4 for additional details.

In terms of format, the comments provided before each editable lines will guide you to edit the part correctly. Note that even though the format is indicated for each item, using the IDE requires minimal familiarity with Matlab programming and its syntax. Note also that the code cannot check for improper inputs, so that any misspecified entry will result in an error during the programme.

Once the file is properly edited, starting the toolbox only requires to execute the script: if needed, refer to section 4.2.

5.3 Running Alexandria on the fly: Python edition

The Graphical User Interface and Integrated Development Environments provide convenient ways to run your models, but they may lack flexibility. For instance, experienced computer or data scientists may want to integrate the model estimation within some larger programme or application. In this case, the GUI and IDE are not suitable. Fortunately, Alexandria also offers the possibility to call the models on the fly, as you would do for instance with libraries such as Scikit-learn or statsmodels. In this case, you can directly create model objects, call their methods and recover their attributes, as you would with any object.

The following piece of code provides a simple example of the use of Alexandria on the fly (here to replicate the simple Bayesian regression developed in section 9.8 of the textbook):

```
# imports: SimpleBayesianRegression class, load_taylor function, Numpy library
from alexandria.linear_regression import SimpleBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np

# create regression data from Taylor dataset
taylor_data = ds.load_taylor()
y = taylor_data[:,0]
X = taylor_data[:,1:]

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01

# create regression object
sbr = SimpleBayesianRegression(endogenous=y, exogenous=X, constant=True,
b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)

# use method 'estimate' to train the model
sbr.estimate()

# print posterior estimates recovered from model attribute 'estimates_beta'
estimates = sbr.estimates_beta
print('posterior median (constant): ' + str(round(estimates[0,1],2)))
print('posterior median (inflation): ' + str(round(estimates[1,1],2)))
print('posterior median (output gap): ' + str(round(estimates[2,1],2)))
```

Let's take a closer look at the code. Lines 2-4 deal with imports. Line 2 and 3 import Alexandria modules for the model: the SimpleBayesianRegression class, and the load_taylor function to load the Taylor rule toy dataset (see section 8.1 for additional details). Line 4 additionally imports the Numpy data science library which is used later in the code.

Lines 7-9 load the Taylor dataset and split it into the exogenous variable y and the exogenous regressors X .

Lines 12-15 generate the elements for the prior mean and variance of the linear regression, consistent with the values introduced in section 9.8 of the textbook. Note that the constant is treated separately from the other exogenous regressors.

Lines 18-19 create the linear regression object "sbr" from the SimpleBayesianRegression class. The constructor takes as argument the endogenous and exogenous regressors (endogenous=y, exogenous=X), adds a constant to the regression (constant=True), and defines the prior mean and variance for the exogenous variables and the constant (b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const). Note that the class could accept many other optional arguments that are here ignored and thus set to their default values.

Line 22 uses the estimate method to train the model and compute its posterior distribution.

Finally, lines 25-28 recover the posterior estimates from the model attribute estimates_beta, then print them on the Python console. The outcome of these final statements is:

```
posterior median (constant): 1.03
posterior median (inflation): 1.11
posterior median (output gap): 0.34
```

It can be verified that these values correspond to that shown in Table 9.1 of the textbook.

This simple example illustrates the flexibility of Alexandria on the fly. The question that remains is: where to find the information on the different classes, methods and attributes? Chapter 7 of this guide provides full documentation for all the model classes proposed in Alexandria, along with a description of their methods and attributes.

If you don't want to read the guide, a quick fix consists in using the help function for the class. For instance, you can execute the command:

```
help(SimpleBayesianRegression)
```

This will display the help of the class in the console (Figure 5.4), detailing the class constructor, methods and attributes.

Two remarks to conclude. First, it is strongly recommended that you proceed to a permanent installation if you intend to use Alexandria on the fly. This will save the hassle of creating local copies of the toolbox everytime you want to estimate a model. Second, a use on the fly provides additional flexibility but requires good knowledge of Python programming, and in particular some familiarity with the object-oriented paradigm. Novice users should prefer the GUI and IDE for an optimal experience.

```
Entrée [2]: help(SimpleBayesianRegression)

Help on class SimpleBayesianRegression in module alexandria.linear_regression.simple_bayesian_regression:

class SimpleBayesianRegression(alexandria.linear_regression.linear_regression.LinearRegression)
| SimpleBayesianRegression(endogenous, exogenous, constant=True, trend=False, quadratic_trend=False, b_exogenous=
0, V_exogenous=1, b_constant=0, V_constant=1, b_trend=0, V_trend=1, b_quadratic_trend=0, V_quadratic_trend=1, credib
ility_level=0.95, verbose=False)
|
| Simplest Bayesian linear regression, developed in section 9.2
|
| Parameters:
| -----
| endogenous : ndarray of shape (n_obs,)
|   endogenous or explained variable
|
| exogenous : ndarray of shape (n_obs,n_regressors)
|   exogenous or explanatory variables
|
| constant : bool, default = True
|   if True, an intercept is included in the regression
|
| trend : bool, default = False
|   if True, a linear trend is included in the regression
|
| quadratic_trend : bool, default = False
|   if True, a quadratic trend is included in the regression
|
| b_exogenous : float or ndarray of shape (n_regressors,), default = 0
|   prior mean for regressors
|
| V_exogenous : float or ndarray of shape (n_regressors,), default = 1
|   prior variance for regressors (positive)
|
| b_constant : float, default = 0
|   prior mean for constant term
|
| V_constant : float, default = 1
|   prior variance for constant (positive)
|
| b_trend : float, default = 0
|   prior mean for trend
```

Figure 5.4: The help function for the SimpleBayesianRegression class

5.4 Running Alexandria on the fly: Matlab edition

The Graphical User Interface and Integrated Developement Environments provide convenient ways to run your models, but they may lack flexibility. For instance, experienced computer or data scientists may want to integrate the model estimation within some larger programme or application. In this case, the GUI and IDE are not suitable. Fortunately, Alexandria also offers the possibility to call the models on the fly, as you would do for instance with libraries such as Scikit-learn or statsmodels. In this case, you can directly create model objects, call their methods and recover their attributes, as you would with any object.

The following piece of code provides a simple example of the use of Alexandria on the fly (here to replicate the simple Bayesian regression developed in section 9.8 of the textbook):

```
% create regression data from Taylor dataset
taylor_data = ds.load_taylor();
y = taylor_data(:,1);
X = taylor_data(:,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;
```

```
% create regression object
sbr = SimpleBayesianRegression(y, X, 'constant', true, 'b_exogenous', b, ...
    'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);

% use method 'estimate' to train the model
sbr.estimate();

% print posterior estimates recovered from model attribute 'estimates_beta'
estimates = sbr.estimates_beta;
disp(['posterior median (constant): ' num2str(round(estimates(1,2),2))]);
disp(['posterior median (inflation): ' num2str(round(estimates(2,2),2))]);
disp(['posterior median (output gap): ' num2str(round(estimates(3,2),2))]);
```

Let's take a closer look at the code. Lines 2-4 load the Taylor dataset and split it into the exogenous variable y and the exogenous regressors X .

Lines 7-10 generate the elements for the prior mean and variance of the linear regression, consistent with the values introduced in section 9.8 of the textbook. Note that the constant is treated separately from the other exogenous regressors.

Lines 13-14 create the linear regression object "sbr" from the SimpleBayesianRegression class. The constructor takes as argument the endogenous and exogenous regressors (y , X), adds a constant to the regression ('constant', true), and defines the prior mean and variance for the exogenous variables and the constant ('b_exogenous', b , 'V_exogenous', V , 'b_constant', b_{const} , 'V_constant', V_{const}). Note that the class could accept many other optional arguments that are here ignored and thus set to their default values.

Line 17 uses the estimate method to train the model and compute its posterior distribution.

Finally, lines 20-23 recover the posterior estimates from the model attribute `estimates_beta`, then print them on the Matlab console. The outcome of these final statements is:

```
posterior median (constant): 1.03
posterior median (inflation): 1.11
posterior median (output gap): 0.34
```

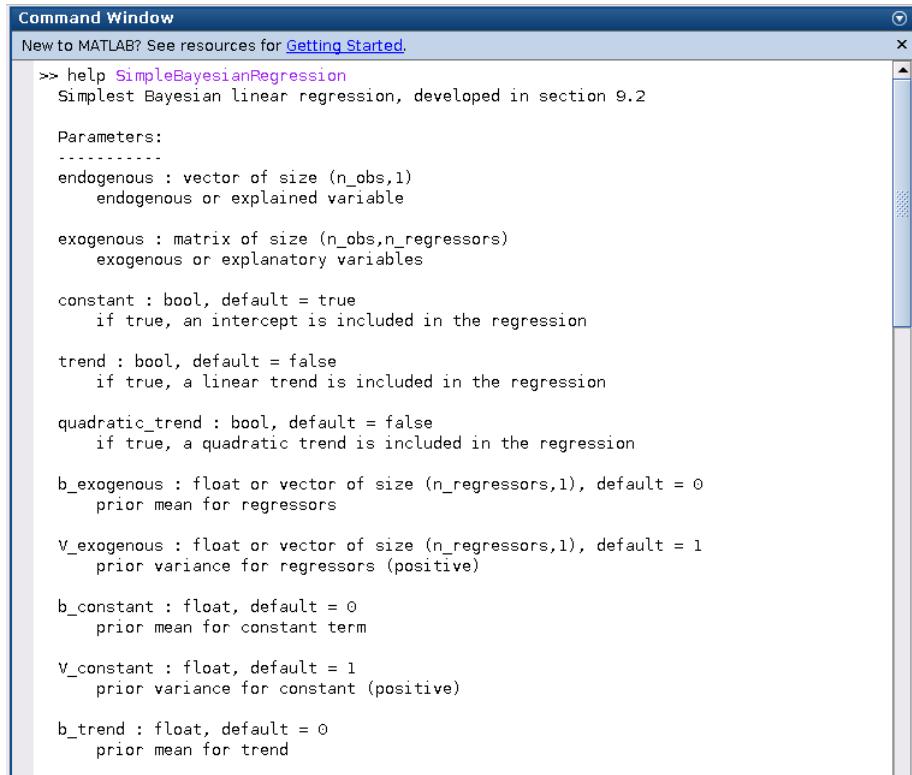
It can be verified that these values correspond to that shown in Table 9.1 of the textbook.

This simple example illustrates the flexibility of Alexandria on the fly. The question that remains is: where to find the information on the different classes, methods and attributes? Chapter 7 of this guide provides full documentation for all the model classes proposed in Alexandria, along with a description of their methods and attributes.

If you don't want to read the guide, a quick fix consists in using the help function for the class. For instance, you can execute the command:

```
help SimpleBayesianRegression
```

This will display the help of the class in the console (Figure 5.5), detailing the class constructor, methods and attributes.



The screenshot shows the MATLAB Command Window with the title 'Command Window'. At the top, it says 'New to MATLAB? See resources for [Getting Started](#)'. Below that, the command `>> help SimpleBayesianRegression` is entered, followed by its documentation. The documentation describes the Simple Bayesian linear regression class, its parameters, and their types and defaults.

```

>> help SimpleBayesianRegression
Simplest Bayesian linear regression, developed in section 9.2

Parameters:
-----
endogenous : vector of size (n_obs,1)
    endogenous or explained variable

exogenous : matrix of size (n_obs,n_regressors)
    exogenous or explanatory variables

constant : bool, default = true
    if true, an intercept is included in the regression

trend : bool, default = false
    if true, a linear trend is included in the regression

quadratic_trend : bool, default = false
    if true, a quadratic trend is included in the regression

b_exogenous : float or vector of size (n_regressors,1), default = 0
    prior mean for regressors

v_exogenous : float or vector of size (n_regressors,1), default = 1
    prior variance for regressors (positive)

b_constant : float, default = 0
    prior mean for constant term

v_constant : float, default = 1
    prior variance for constant (positive)

b_trend : float, default = 0
    prior mean for trend

```

Figure 5.5: The help function for the SimpleBayesianRegression class

Two remarks to conclude. First, it is strongly recommended that you proceed to a permanent installation if you intend to use Alexandria on the fly. This will save the hassle of creating local copies of the toolbox everytime you want to estimate a model. Second, a use on the fly provides additional flexibility but requires good knowledge of Matlab programming, and in particular some familiarity with the object-oriented paradigm. Novice users should prefer the GUI and IDE for an optimal experience.

CHAPTER 6

Alexandria outputs

Once your model is successfully estimated, Alexandria will produce a number of estimation outputs. If you selected the option 'save results in project folder' in the GUI/IDE, Alexandria will create a "results" folder within your project folder and will store in it a number of output files. If you selected the option "graphics and figures" in the GUI/IDE, Alexandria will create a "graphics" folder and save in it a copy of all the plots resulting from model estimation.

6.1 Console outputs

The first estimation output produced by Alexandria is a console output that summarizes model estimation. A typical output is displayed in Figure 6.1.

The output comprises several parts. The top part (green frame) shows the progress bars used to visualize the progression of the different estimation algorithms in real time. It appears only if the progress bar option is selected in the GUI.

The second part (red frame) displays the model estimates for each parameter, including a point estimate and, if applicable, the standard deviations and credibility bands.

The third part exhibits a number of in-sample fit criteria, including traditional frequentist criteria (sum of squared residuals, R^2 and adjusted R^2), but also the Bayesian log10 marginal likelihood. This part is displayed only if the in-sample fit/marginal likelihood options are selected in the GUI.

The final part (purple frame) displays a number of out-of-sample forecast evaluation criteria. Here also traditional frequentist criteria (RMSE, MAE, MAPE, Theil's U and bias) are proposed along with specific Bayesian criteria (CRPS and log score). This part is only displayed if the forecast evaluation option is selected in the GUI and proper counterfactual values have been supplied in the forecast data file.

If the option "save results in project folder" is selected in the GUI, a copy of this console output will be saved in the file "results.txt" in your "results" folder. Also, a second file named "settings.txt" will be generated, which recapitulates all the inputs and options of your model.

Figure 6.1: Alexandria IDE on Jupyter Notebook

6.2 Graphics outputs

The second type of outputs produced by Alexandria is a set of plots summarizing your model estimates and applications. For convenience, these plots are automatically displayed in tab 4 of the Graphical User Interface, as a simple navigator. The navigator is displayed in Figure 6.2:

The "Application" panel (green frame) allows you to navigate across the different applications. So far Alexandria proposes "actual and fitted", "residuals", and "forecasts". To switch from one application to another you can either use the slider, or the rollmenu located below it.

The next three panels ("Display type" panel in orange frame, "Variable" panel in blue frame, and "Responding to" panel in purple frame) are irrelevant so far. They are dedicated to multivariate models and have no use yet since Alexandria only proposes the univariate linear regression for now. These elements have been created for architecture reasons and will become available as Alexandria develops new applications.

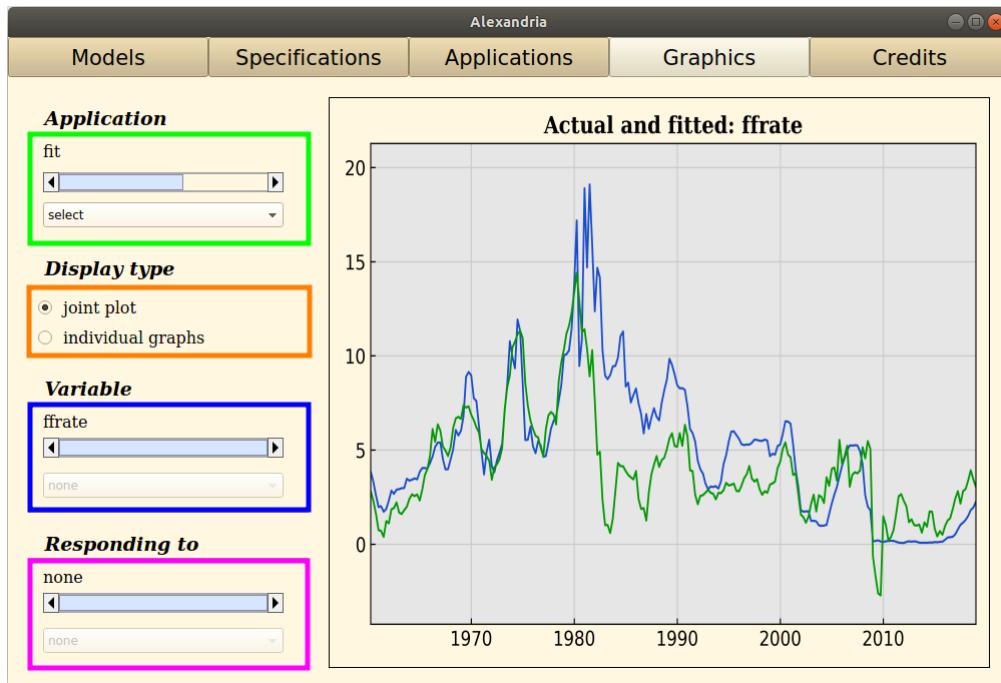


Figure 6.2: Tab 4 of the GUI as a graphical navigator

6.3 File outputs

Alexandria produces a number of additional estimation output files in the "results" folder of the project, depending on the selected applications. For instance, if you selected the "forecasts" application, Alexandria will produce a file "forecasts.csv" that recapitulates the prediction estimates. The file is fairly straightforward: it provides a summary of the posterior estimates of the forecasts, indicating for each prediction its lower bound, median and upper bound. An example of such file for the linear regression is displayed in Figure 6.3:

	A	B	C	D	E	F	G	H
1		lower_bound	median	upper_bound				
2	31/03/2019	-1.490181438	3.364903389	8.007401201				
3	30/06/2019	-1.780786526	3.379606958	7.991236419				
4	30/09/2019	-1.592103928	3.334017837	8.066680876				
5	31/12/2019	-0.79889496	3.88313814	8.720881438				
6	31/03/2020	-2.470712525	2.472950756	7.44532085				
7	30/06/2020	-7.108454019	-1.926935779	2.945229056				
8	30/09/2020	-3.491153154	1.141194934	6.121771683				
9	31/12/2020	-3.711353641	1.321109356	6.238147068				
10								
11								
12								
13								

Figure 6.3: Tab 4 of the GUI as a graphical navigator

So far the applications for which Alexandria can record the posterior estimates and save them in csv files are: actual and fitted values, residuals, and forecasts.

CHAPTER 7

Alexandria classes - documentation

This chapter proposes an exhaustive documentation for the classes corresponding to the different models proposed in Alexandria. The documentation is provided for the Python version of Alexandria, but it is immediately applicable in a similar way to the Matlab version.

Note also that the documentation can be accessed directly from the console, by using the help function. For instance, to obtain the documentation on the MaximumLikelihoodRegression class, you may simply execute the Python command:

```
help(MaximumLikelihoodRegression)
```

On Matlab, equivalently:

```
help MaximumLikelihoodRegression
```

This will print on the console the same information as that developed in the incoming sections.

7.1 Linear regression: MaximumLikelihoodRegression

A simple maximum likelihood (OLS) regression, described in section 9.1.

Class:

```
alexandria.linear_regression.MaximumLikelihoodRegression(endogenous, exogenous, constant = True,
trend = False, quadratic_trend = False, credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,:): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
n	int: number of observations, defined in (3.9.1)
k	int: dimension of beta, defined in (3.9.1)
beta	ndarray of shape (k,:): regression coefficients
sigma	float: residual variance, defined in (3.9.1)
estimates_beta	ndarray of shape (k,4): estimates for beta column 1: interval lower bound; column 2: point estimate; column 3: interval upper bound; column 4: standard deviation
sigma	float: residual variance, defined in (3.9.1)
X_hat	ndarray of shape (m,k): predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
estimates_forecasts	ndarray of shape (m,3): estimates for predictions column 1: interval lower bound; column 2: point estimate; column 3: interval upper bound
estimates_fit	ndarray of shape (n,:): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,:): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)

Methods:**■ estimate()**

trains the model, produces estimates for parameters β and σ .

parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals.

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2).

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions
 - column 1: credibility interval lower bound; column 2: median;
 - column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

Example (Python):

```
# imports
from alexandria.linear_regression import MaximumLikelihoodRegression
from alexandria.datasets import data_sets as ds

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]

# create and train regression
mlr = MaximumLikelihoodRegression(endogenous=y_train, exogenous=X_train,
constant=True)
mlr.estimate()
```

```
# get predictions on test sample, run forecast evaluation, display RMSE
estimates_forecasts = mlr.forecast(X_test, 0.95)
mlr.forecast_evaluation(y_test)
print('RMSE on test sample : '
+ str(round(mlr.forecast_evaluation_criteria['rmse'], 2)))

'RMSE on test sample: 3.21'
```

Example (Matlab):

```
% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% create and train regression
mlr = MaximumLikelihoodRegression(y_train, X_train, 'constant', true);
mlr.estimate();

% get predictions on test sample, run forecast evaluation, display RMSE
estimates_forecasts = mlr.forecast(X_test, 0.95);
mlr.forecast_evaluation(y_test);
disp(['RMSE on test sample: ' ...
num2str(round(mlr.forecast_evaluation_criteria.rmse, 2))]);

'RMSE on test sample: 3.21'
```

7.2 Linear regression: SimpleBayesianRegression

A simple Bayesian regression, described in section 9.2.

Class:

```
alexandria.linear_regression.SimpleBayesianRegression(endogenous, exogenous, constant = True, trend = False, quadratic_trend = False, b_exogenous = 0, V_exogenous = 1, b_constant = 0, V_constant = 1, b_trend = 0, V_trend = 1, b_quadratic_trend = 0, V_quadratic_trend = 1, credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,:): endogenous variable, defined in (3.9.3)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.3)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,), default = 0: prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,), default = 1: prior variance for regressors
b_constant	float, default = 0: prior mean for constant term
V_constant	float, default = 1: prior variance for constant (positive)
b_trend	float, default = 0: prior mean for trend
V_trend	float, default = 1: prior variance for trend (positive)
b_quadratic_trend	float, default = 0: prior mean for quadratic trend
V_quadratic_trend	float, default = 1: prior variance for quadratic trend (positive)
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,): prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,): prior variance for regressors
b_constant	float: prior mean for constant term
V_constant	float: prior variance for constant (positive)
b_trend	float: prior mean for trend
V_trend	float: prior variance for trend (positive)
b_quadratic_trend	float: prior mean for quadratic trend
V_quadratic_trend	float: prior variance for quadratic trend (positive)
b	ndarray of shape (k,): prior mean, defined in (3.9.10)
V	ndarray of shape (k,k): prior variance, defined in (3.9.10)
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
n	int: number of observations, defined in (3.9.1)
k	int: dimension of beta, defined in (3.9.1)
sigma	float: residual variance, defined in (3.9.1)
b_bar	ndarray of shape (k,): posterior mean, defined in (3.9.14)
V_bar	ndarray of shape (k,k): posterior variance, defined in (3.9.14)
estimates_beta	ndarray of shape (k,4): posterior estimates for beta column 1: interval lower bound; column 2: median; column 3: interval upper bound; column 4: standard deviation
X_hat	ndarray of shape (m,k): predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
estimates_forecasts	ndarray of shape (m,3): posterior estimates for predictions column 1: interval lower bound; column 2: median; column 3: interval upper bound
estimates_fit	ndarray of shape (n,): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)
m_y	float: log10 marginal likelihood

Methods:**■ estimate()**

trains the model, produces estimates for parameters β and σ

parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2)

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions
 - column 1: credibility interval lower bound; column 2: median;
 - column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

■ marginal_likelihood()

log10 marginal likelihood, defined in (3.10.20).

parameters:

- none

returns:

- m_y: float, log 10 marginal likelihood value.

■ optimize_hyperparameters(type)

optimize V by maximizing the marginal likelihood

parameters:

- type: int, optimization type (1 or 2); 1 = optimize scalar v; 2 = optimize vector V

returns:

- none

Example (Python):

```
# imports
from alexandria.linear_regression import SimpleBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01
```

```

# create and train regression
sbr = SimpleBayesianRegression(endogenous=y_train, exogenous=X_train,
constant=True, b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)
sbr.estimate()

# get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = sbr.forecast(X_test, 0.95)
sbr.forecast_evaluation(y_test)
print('log score on test sample : '
+ str(round(sbr.forecast_evaluation_criteria['log_score'], 2)))

'log score on test sample : 2.17'

```

Example (Matlab):

```

% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;

% create and train regression
sbr = SimpleBayesianRegression(y_train, X_train, 'constant', true, 'b_exogenous', b, ...
'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);
sbr.estimate();

% get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = sbr.forecast(X_test, 0.95);
sbr.forecast_evaluation(y_test);
disp(['log score on test sample: ' ...
num2str(round(sbr.forecast_evaluation_criteria.log_score, 2))]);

'log score on test sample : 2.17'

```

7.3 Linear regression: HierarchicalBayesianRegression

A hierarchical Bayesian regression, described in section 9.3.

Class:

```
alexandria.linear_regression.HierarchicalBayesianRegression(endogenous, exogenous, constant = True,
trend = False, quadratic_trend = False, b_exogenous = 0, V_exogenous = 1, b_constant = 0, V_constant
= 1, b_trend = 0, V_trend = 1, b_quadratic_trend = 0, V_quadratic_trend = 1, alpha = 1e-4, delta = 1e-4,
credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,:): endogenous variable, defined in (3.9.3)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.3)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,), default = 0: prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,), default = 1: prior variance for regressors
b_constant	float, default = 0: prior mean for constant term
V_constant	float, default = 1: prior variance for constant (positive)
b_trend	float, default = 0: prior mean for trend
V_trend	float, default = 1: prior variance for trend (positive)
b_quadratic_trend	float, default = 0: prior mean for quadratic trend
V_quadratic_trend	float, default = 1: prior variance for quadratic trend (positive)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,): prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,): prior variance for regressors
b_constant	float: prior mean for constant term
V_constant	float: prior variance for constant (positive)
b_trend	float: prior mean for trend
V_trend	float: prior variance for trend (positive)
b_quadratic_trend	float: prior mean for quadratic trend
V_quadratic_trend	float: prior variance for quadratic trend (positive)
b	ndarray of shape (k,): prior mean, defined in (3.9.10)
V	ndarray of shape (k,k): prior variance, defined in (3.9.10)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
n	int: number of observations, defined in (3.9.1)
k	int: dimension of beta, defined in (3.9.1)
b_bar	ndarray of shape (k,): posterior mean, defined in (3.9.14)
V_bar	ndarray of shape (k,k): posterior variance, defined in (3.9.14)
alpha_bar	float: posterior shape, defined in (3.9.24)
delta_bar	float: posterior scale, defined in (3.9.24)
location	ndarray of shape (k,): location for the student posterior of beta, defined in (3.9.28)
scale	ndarray of shape (k,k): scale for the student posterior of beta, defined in (3.9.28)
df	float: degrees of freedom for the student posterior of beta, defined in (3.9.28)
estimates_beta	ndarray of shape (k,4): posterior estimates for beta column 1: interval lower bound; column 2: median; column 3: interval upper bound; column 4: standard deviation
estimates_sigma	float: posterior estimate for sigma
X_hat	ndarray of shape (m,k): predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
estimates_forecasts	ndarray of shape (m,3): posterior estimates for predictions column 1: interval lower bound; column 2: median; column 3: interval upper bound
estimates_fit	ndarray of shape (n,): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)
m_y	float: log10 marginal likelihood

Methods:**■ estimate()**

trains the model, produces estimates for parameters β and σ
parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2)

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions

column 1: credibility interval lower bound; column 2: median;

column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

■ marginal_likelihood()

log10 marginal likelihood, defined in (3.10.20).

parameters:

- none

returns:

- m_y: float, log 10 marginal likelihood value.

■ optimize_hyperparameters(type)

optimize V and delta by maximizing the marginal likelihood

parameters:

- type: int, optimization type (1 or 2); 1 = optimize scalar v; 2 = optimize vector V

returns:

- none

Example (Python):

```
# imports
from alexandria.linear_regression import HierarchicalBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np
```

```

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01

# create and train regression
hbr = HierarchicalBayesianRegression(endogenous=y_train, exogenous=X_train,
constant=True, b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)
hbr.estimate()

# get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = hbr.forecast(X_test, 0.95)
hbr.forecast_evaluation(y_test)
print('log score on test sample : '
+ str(round(hbr.forecast_evaluation_criteria['log_score'], 2)))

'log score on test sample : 2.33'

```

Example (Matlab):

```

% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;

% create and train regression
hbr = HierarchicalBayesianRegression(y_train, X_train, 'constant', true, 'b_exogenous', b, ...
'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);
hbr.estimate();

% get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = hbr.forecast(X_test, 0.95);
hbr.forecast_evaluation(y_test);
disp(['log score on test sample: ' ...
num2str(round(hbr.forecast_evaluation_criteria.log_score, 2))]);

'log score on test sample : 2.34'

```

7.4 Linear regression: IndependentBayesianRegression

A independent Bayesian regression with Gibbs sampling, described in section 9.4.

Class:

```
alexandria.linear_regression.IndependentBayesianRegression(endogenous, exogenous, constant = True,
trend = False, quadratic_trend = False, b_exogenous = 0, V_exogenous = 1, b_constant = 0, V_constant
= 1, b_trend = 0, V_trend = 1, b_quadratic_trend = 0, V_quadratic_trend = 1, alpha = 1e-4, delta = 1e-4,
iterations = 2000, burn = 1000, credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,:): endogenous variable, defined in (3.9.3)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.3)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,), default = 0: prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,), default = 1: prior variance for regressors
b_constant	float, default = 0: prior mean for constant term
V_constant	float, default = 1: prior variance for constant (positive)
b_trend	float, default = 0: prior mean for trend
V_trend	float, default = 1: prior variance for trend (positive)
b_quadratic_trend	float, default = 0: prior mean for quadratic trend
V_quadratic_trend	float, default = 1: prior variance for quadratic trend (positive)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
iterations	int, default = 2000: post burn-in iterations for MCMC algorithm
burn	int, default = 1000: burn-in iterations for MCMC algorithm
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,): prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,): prior variance for regressors
b_constant	float: prior mean for constant term
V_constant	float: prior variance for constant (positive)
b_trend	float: prior mean for trend
V_trend	float: prior variance for trend (positive)
b_quadratic_trend	float: prior mean for quadratic trend
V_quadratic_trend	float: prior variance for quadratic trend (positive)
b	ndarray of shape (k,): prior mean, defined in (3.9.10)
V	ndarray of shape (k,k): prior variance, defined in (3.9.10)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
iterations	int, default = 2000: post burn-in iterations for MCMC algorithm
burn	int, default = 1000: burn-in iterations for MCMC algorithm
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
n	int: number of observations, defined in (3.9.1)
k	int: dimension of beta, defined in (3.9.1)
alpha_bar	float: posterior shape, defined in (3.9.24)
mcmc_beta	matrix of size (k, iterations): storage of mcmc values for beta
mcmc_sigma	vector of size (iterations,): storage of mcmc values for sigma
estimates_beta	ndarray of shape (k,4): posterior estimates for beta column 1: interval lower bound; column 2: median; column 3: interval upper bound; column 4: standard deviation
estimates_sigma	float: posterior estimate for sigma
X_hat	ndarray of shape (m,k): predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
mcmc_forecasts	matrix of size (m, iterations): storage of mcmc values for forecasts
estimates_forecasts	ndarray of shape (m,3): posterior estimates for predictions column 1: interval lower bound; column 2: median; column 3: interval upper bound
estimates_fit	ndarray of shape (n,): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)
m_y	float: log10 marginal likelihood

Methods:**■ estimate()**

trains the model, produces estimates for parameters β and σ

parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2)

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions

column 1: credibility interval lower bound; column 2: median;

column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

■ marginal_likelihood()

log10 marginal likelihood, defined in (3.10.20).

parameters:

- none

returns:

- m_y: float, log 10 marginal likelihood value.

Example (Python):

```
# imports
from alexandria.linear_regression import IndependentBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]
```

```

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01

# create and train regression
ibr = IndependentBayesianRegression(endogenous=y_train, exogenous=X_train,
constant=True, b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)
ibr.estimate()

# get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = ibr.forecast(X_test, 0.95)
ibr.forecast_evaluation(y_test)
print('log score on test sample : '
+ str(round(ibr.forecast_evaluation_criteria['log_score'], 2)))

'log score on test sample : 2.18'

```

Example (Matlab):

```

% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;

% create and train regression
ibr = IndependentBayesianRegression(y_train, X_train, 'constant', true, 'b_exogenous', b, ...
'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);
ibr.estimate();

% get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = ibr.forecast(X_test, 0.95);
ibr.forecast_evaluation(y_test);
disp(['log score on test sample: ' ...
num2str(round(ibr.forecast_evaluation_criteria.log_score, 2))]);

'log score on test sample : 2.18'

```

7.5 Linear regression: Heteroscedastic Bayesian Regression

A Bayesian regression with heteroscedastic disturbances, described in section 9.5.

Class:

```
alexandria.linear_regression.HeteroscedasticBayesianRegression(endogenous, exogenous, heteroscedastic = None, constant = True, trend = False, quadratic_trend = False, b_exogenous = 0, V_exogenous = 1, b_constant = 0, V_constant = 1, b_trend = 0, V_trend = 1, b_quadratic_trend = 0, V_quadratic_trend = 1, alpha = 1e-4, g = 0, Q = 100, tau = 0.001, delta = 1e-4, iterations = 2000, burn = 1000, thinning = False, thinning_frequency = 10, credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.3)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.3)
heteroscedastic	ndarray of shape (n_obs,h), default = exogenous: heteroscedasticity variables, defined in (3.9.37)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,), default = 0: prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,), default = 1: prior variance for regressors
b_constant	float, default = 0: prior mean for constant term
V_constant	float, default = 1: prior variance for constant (positive)
b_trend	float, default = 0: prior mean for trend
V_trend	float, default = 1: prior variance for trend (positive)
b_quadratic_trend	float, default = 0: prior mean for quadratic trend
V_quadratic_trend	float, default = 1: prior variance for quadratic trend (positive)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
g	float or ndarray of shape (h,), default = 0: prior mean, defined in (3.9.43)
Q	float or ndarray of shape (h,), default = 100: prior variance, defined in (3.9.43)
tau	float, default = 0.001: variance of the random walk shock, defined in (3.9.50)
iterations	int, default = 2000: post burn-in iterations for MCMC algorithm
burn	int, default = 1000: burn-in iterations for MCMC algorithm
thinning	bool, default = False: if True, thinning is applied to posterior draws from MCMC algorithm
thinning_frequency	int, default = 10: if thinning is True, retains only one out of so many draws from MCMC algorithm
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
heteroscedastic	ndarray of shape (n_obs,h), default = exogenous: heteroscedasticity variables, defined in (3.9.37)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,): prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,): prior variance for regressors
b_constant	float: prior mean for constant term
V_constant	float: prior variance for constant (positive)
b_trend	float: prior mean for trend
V_trend	float: prior variance for trend (positive)
b_quadratic_trend	float: prior mean for quadratic trend
V_quadratic_trend	float: prior variance for quadratic trend (positive)
b	ndarray of shape (k,): prior mean, defined in (3.9.10)
V	ndarray of shape (k,k): prior variance, defined in (3.9.10)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
g	ndarray of shape (h,): prior mean, defined in (3.9.43)
Q	ndarray of shape (h,h): prior variance, defined in (3.9.43)
tau	float: variance of the random walk shock, defined in (3.9.50)
iterations	int: post burn-in iterations for MCMC algorithm
burn	int: burn-in iterations for MCMC algorithm
thinning	bool: if True, thinning is applied to posterior draws from MCMC algorithm
thinning_frequency	int: if thinning is True, retains only one out of so many draws from MCMC algorithm
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
Z	ndarray of shape (n,h): heteroscedasticity variables, defined in (3.9.39)
n	int: number of observations, defined in (3.9.1)
k	int: dimension of beta, defined in (3.9.1)
h	int: dimension of gamma, defined in (3.9.37)
alpha_bar	float: posterior shape, defined in (3.9.24)
mcmc_beta	matrix of size (k, iterations): storage of mcmc values for beta
mcmc_sigma	vector of size (iterations,): storage of mcmc values for sigma
mcmc_gamma	ndarray of shape (h,iterations): storage of mcmc values for gamma
estimates_beta	ndarray of shape (k,4): posterior estimates for beta column 1: interval lower bound; column 2: median; column 3: interval upper bound; column 4: standard deviation
estimates_sigma	float: posterior estimate for sigma
estimates_gamma	ndarray of shape (h,3): posterior estimates for gamma column 1: interval lower bound; column 2: median; column 3: interval upper bound
X_hat	ndarray of shape (m,k): predictors for the model
Z_hat	ndarray of shape (m,h): heteroscedasticity predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
mcmc_forecasts	matrix of size (m, iterations): storage of mcmc values for forecasts
estimates_forecasts	ndarray of shape (m,3): posterior estimates for predictions column 1: interval lower bound; column 2: median; column 3: interval upper bound
estimates_fit	ndarray of shape (n,): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)
m_y	float: log10 marginal likelihood

Methods:

■ estimate()

trains the model, produces estimates for parameters β , σ and γ
parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2)

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions

column 1: credibility interval lower bound; column 2: median;

column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

■ marginal_likelihood()

log10 marginal likelihood, defined in (3.10.20).

parameters:

- none

returns:

- m_y: float, log 10 marginal likelihood value.

Example (Python):

```
# imports
from alexandria.linear_regression import HeteroscedasticBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]
```

```

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01

# create and train regression
hbr = HeteroscedasticBayesianRegression(endogenous=y_train, exogenous=X_train,
constant=True, b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)
hbr.estimate()

# get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = hbr.forecast(X_test, 0.95)
hbr.forecast_evaluation(y_test)
print('log score on test sample : '
+ str(round(hbr.forecast_evaluation_criteria['log_score'], 2)))

'log score on test sample : 2.14'

```

Example (Matlab):

```

% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;

% create and train regression
hbr = HeteroscedasticBayesianRegression(y_train, X_train, 'constant', true, 'b_exogenous', b,
'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);
hbr.estimate();

% get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = hbr.forecast(X_test, 0.95);
hbr.forecast_evaluation(y_test);
disp(['log score on test sample: ' ...
num2str(round(hbr.forecast_evaluation_criteria.log_score, 2))]);

'log score on test sample : 2.14'

```

7.6 Linear regression: AutocorrelatedBayesianRegression

A Bayesian regression with autocorrelated disturbances, described in section 9.6.

Class:

```
alexandria.linear_regression.AutocorrelatedBayesianRegression(endogenous, exogenous, q = 1, constant = True, trend = False, quadratic_trend = False, b_exogenous = 0, V_exogenous = 1, b_constant = 0, V_constant = 1, b_trend = 0, V_trend = 1, b_quadratic_trend = 0, V_quadratic_trend = 1, alpha = 1e-4, delta = 1e-4, p = 0, H = 100, iterations = 2000, burn = 1000, credibility_level = 0.95, verbose = False)
```

Parameters:

parameter	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.3)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.3)
q	int, default = 1: order of autocorrelation (number of residual lags)
constant	bool, default = True: if True, an intercept is included in the regression
trend	bool, default = False: if True, a linear trend is included in the regression
quadratic_trend	bool, default = False: if True, a quadratic trend is included in the regression
b_exogenous	float or ndarray of shape (n_regressors,), default = 0: prior mean for regressors
V_exogenous	float or ndarray of shape (n_regressors,), default = 1: prior variance for regressors
b_constant	float, default = 0: prior mean for constant term
V_constant	float, default = 1: prior variance for constant (positive)
b_trend	float, default = 0: prior mean for trend
V_trend	float, default = 1: prior variance for trend (positive)
b_quadratic_trend	float, default = 0: prior mean for quadratic trend
V_quadratic_trend	float, default = 1: prior variance for quadratic trend (positive)
alpha	float, default = 1e-4: prior shape, defined in (3.9.21)
delta	float, default = 1e-4: prior scale, defined in (3.9.21)
p	float or ndarray of shape (q,), default = 0: prior mean, defined in (3.9.62)
H	float or ndarray of shape (q,), default = 100: prior variance, defined in (3.9.62)
iterations	int, default = 2000: post burn-in iterations for MCMC algorithm
burn	int, default = 1000: burn-in iterations for MCMC algorithm
credibility_level	float, default = 0.95: credibility level (between 0 and 1)
verbose	bool, default = False: if True, displays a progress bar

Attributes:

attribute	description
endogenous	ndarray of shape (n_obs,): endogenous variable, defined in (3.9.1)
exogenous	ndarray of shape (n_obs,n_regressors): exogenous variables, defined in (3.9.1)
q	int: order of autocorrelation (number of residual lags)
constant	bool: if True, an intercept is included in the regression
trend	bool: if True, a linear trend is included in the regression
quadratic_trend	bool: if True, a quadratic trend is included in the regression
b_exogenous	ndarray of shape (n_regressors,): prior mean for regressors
V_exogenous	ndarray of shape (n_regressors,): prior variance for regressors
b_constant	float: prior mean for constant term
V_constant	float: prior variance for constant (positive)
b_trend	float: prior mean for trend
V_trend	float: prior variance for trend (positive)
b_quadratic_trend	float: prior mean for quadratic trend
V_quadratic_trend	float: prior variance for quadratic trend (positive)
b	ndarray of shape (k,): prior mean, defined in (3.9.10)
V	ndarray of shape (k,k): prior variance, defined in (3.9.10)
alpha	float: prior shape, defined in (3.9.21)
delta	float: prior scale, defined in (3.9.21)
p	ndarray of shape (q,): prior mean, defined in (3.9.62)
H	ndarray of shape (q,q): prior variance, defined in (3.9.62)
iterations	int: post burn-in iterations for MCMC algorithm
burn	int: burn-in iterations for MCMC algorithm
credibility_level	float: credibility level (between 0 and 1)
verbose	bool: if True, displays a progress bar
y	ndarray of shape (n,): endogenous variable, defined in (3.9.3)
X	ndarray of shape (n,k): exogenous variables, defined in (3.9.3)
T	int: number of observations, defined in (3.9.52)
k	int: dimension of beta, defined in (3.9.1)
alpha_bar	float: posterior shape, defined in (3.9.24)
mcmc_beta	matrix of size (k, iterations): storage of mcmc values for beta
mcmc_sigma	vector of size (iterations,): storage of mcmc values for sigma
mcmc_phi	ndarray of shape (q,iterations): storage of mcmc values for phi
estimates_beta	ndarray of shape (k,4): posterior estimates for beta column 1: interval lower bound; column 2: median; column 3: interval upper bound; column 4: standard deviation
estimates_sigma	float: posterior estimate for sigma
estimates_phi	ndarray of shape (q,3): posterior estimates for phi column 1: interval lower bound; column 2: median; column 3: interval upper bound
X_hat	ndarray of shape (m,k): predictors for the model
m	int: number of predicted observations, defined in (3.10.1)
mcmc_forecasts	matrix of size (m, iterations): storage of mcmc values for forecasts
estimates_forecasts	ndarray of shape (m,3): posterior estimates for predictions column 1: interval lower bound; column 2: median; column 3: interval upper bound
estimates_fit	ndarray of shape (n,): posterior estimates (median) for in sample-fit
estimates_residuals	ndarray of shape (n,): posterior estimates (median) for residuals
insample_evaluation	dict: in-sample fit evaluation (SSR, R2, adj-R2)
forecast_evaluation_criteria	dict: out-of-sample forecast evaluation (RMSE, MAE, ...)
m_y	float: log10 marginal likelihood

Methods:**■ estimate()**

trains the model, produces estimates for parameters β , σ and γ
parameters:

- none

returns:

- none

■ fit_and_residuals()

estimates in-sample fit and regression residuals

parameters:

- none

returns:

- none

■ forecast(X_hat, credibility_level)

predictions for the linear regression model, using (3.10.1) and (3.10.2)

parameters:

- X_hat: ndarray of shape (m,k), predictors for the model

- credibility_level: float, credibility level for predictions (between 0 and 1)

returns:

- estimates_forecasts: ndarray of shape (m, 3), posterior estimates for predictions

column 1: credibility interval lower bound; column 2: median;

column 3: credibility interval upper bound

■ forecast_evaluation(y)

forecast evaluation criteria for the linear regression model.

parameters:

- y: ndarray of shape (m,), array of realised values for forecast evaluation

returns:

- none

■ marginal_likelihood()

log10 marginal likelihood, defined in (3.10.20).

parameters:

- none

returns:

- m_y: float, log 10 marginal likelihood value.

Example (Python):

```
# imports
from alexandria.linear_regression import AutocorrelatedBayesianRegression
from alexandria.datasets import data_sets as ds
import numpy as np

# load Taylor dataset, split as train/test
taylor_data = ds.load_taylor()
y_train, X_train = taylor_data[:198,0], taylor_data[:198,1:]
y_test, X_test = taylor_data[198:,0], taylor_data[198:,1:]
```

```

# set prior mean and prior variance for the model
b = np.array([1.5, 0.5])
b_const = 1
V = np.array([0.01, 0.0025])
V_const = 0.01

# create and train regression
abr = AutocorrelatedBayesianRegression(endogenous=y_train, exogenous=X_train,
constant=True, b_exogenous=b, V_exogenous=V, b_constant=b_const, V_constant=V_const)
abr.estimate()

# get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = abr.forecast(X_test, 0.95)
abr.forecast_evaluation(y_test)
print('log score on test sample : '
+ str(round(abr.forecast_evaluation_criteria['log_score'], 2)))

'log score on test sample : 2.12'

```

Example (Matlab):

```

% load Taylor dataset, split as train/test
taylor_data = ds.load_taylor();
y_train = taylor_data(1:198,1); X_train = taylor_data(1:198,2:end);
y_test = taylor_data(198:end,1); X_test = taylor_data(198:end,2:end);

% set prior mean and prior variance for the model
b = [1.5, 0.5]';
b_const = 1;
V = [0.01, 0.0025]';
V_const = 0.01;

% create and train regression
abr = AutocorrelatedBayesianRegression(y_train, X_train, 'constant', true, 'b_exogenous', b,
'V_exogenous', V, 'b_constant', b_const, 'V_constant', V_const);
abr.estimate();

% get predictions on test sample, run forecast evaluation, display log score
estimates_forecasts = abr.forecast(X_test, 0.95);
abr.forecast_evaluation(y_test);
disp(['log score on test sample: ' ...
num2str(round(abr.forecast_evaluation_criteria.log_score, 2))]);

'log score on test sample : 2.12'

```

CHAPTER 8

Alexandria datasets

Alexandria comes with a few pre-built datasets that can be loaded directly into the working space without external files. These are mostly toy datasets proposed for pedagogical purposes, but they constitute interesting economic data sources on their own.

8.1 The Taylor rule dataset

Description:

A dataset for the estimation of a Taylor rule for the United States.

Characteristics:

observations	264
frequency	quarterly
start date	1955q1
end date	2020q4
variables	3
variable description	ffrate : federal funds rate inf : year-to-year inflation gap : output gap, as percentage deviation from potential output

This is the dataset used in sections 9.8 and 10.4 of the textbook. It can be called with the following functions:

Example (Python):

```
# imports: required to use the load functions
from alexandria.datasets import data_sets as ds

# load Taylor dataset as raw numpy array, with numeric data only
taylor_data = ds.load_taylor()

# or load Taylor dataset as pandas dataframe, with dates and variable names
taylor_data = ds.load_taylor_table()
```

Example (Matlab):

```
% load Taylor dataset as regular matrix, with numeric data only
taylor_data = ds.load_taylor();

% load Taylor dataset as Matlab table, with dates and variable names
taylor_data = ds.load_taylor_table();
```