APPENDIX A

Installation

This Appendix covers the installation and setup of a Python environment for scientific computing on commonly used platforms. As discussed in Chapter 1, the scientific computing environment for Python is not a single product, but rather a diverse ecosystem of packages and libraries, and there are numerous possible ways to install and configure a Python environment on any given platform. Python itself is rather easy to install,[1] and on many operating systems it is even preinstalled. All pure Python libraries that are hosted on the *Python Package Index*[2] are also easily installed, for example, using `pip` and a command such as `pip install PACKAGE`, where `PACKAGE` is the name of the package to install. The *pip* software then searches for the package on the Python Package Index, and downloads and installs it, if it is found. For example, to install IPython we can use:

```
$ pip install ipython
```

and to upgrade an already installed package we simply add the `--upgrade` flag to the `pip` command:

```
$ pip install --upgrade ipython
```

However, many libraries for computing with Python are not pure Python libraries, and they frequently have dependencies on system libraries written in other languages, such as C and Fortran. These dependencies cannot be handled by `pip` and the Python Package Index, and to build such libraries from source requires C and Fortran compilers to be installed. In other words, installing a full scientific computing software stack for Python manually can be difficult, or at least time consuming and tedious. To solve this problem, there have emerged a number of prepackaged Python environments with automated installers. The most popular environments are Continuum Analytics's *Anaconda*[3] and Enthought's *Canopy*,[4] which are both sponsored by corporations with close connections to the open-source scientific Python community, and *Python(x,y)*,[5] which is a community-maintained environment that targets Microsoft's operating systems. These environments all have in common that they bundle the Python interpreter, the required system libraries and tools, and a large number of scientific computing-oriented Python libraries in an easy-to-install distribution. Any of these environments can readily be used to set up the software required to run the code discussed in this book, but in the following we use the Anaconda environment from Continuum Analytics. In particular, we discuss Miniconda – a lightweight version of Anaconda – and the package manager `conda`.

[1] Installers for all major platforms are available for download at `http://www.python.org/downloads`
[2] `http://pypi.python.org`
[3] `http://continuum.io/downloads`
[4] `http://www.enthought.com/products/canopy`
[5] `http://python-xy.github.io`

R. Johansson, *Numerical Python*, DOI 10.1007/978-1-4842-0553-2_20

Miniconda and Conda

The Anaconda environment, which comes bundled with a large number of libraries, is a convenient way to get a scientific computing environment for Python up and running quickly. However, for clarity, here we start with a Miniconda environment and explicitly install the packages that we need. This way we control exactly which packages are included in the environment we set up. Miniconda is a minimal version of Anaconda, which only includes the most basic components: a Python interpreter, a few fundamental libraries, and the `conda` package manager. The download page for the Miniconda project `http://conda.pydata.org/miniconda.html` contains installers for Linux, Mac OS X, and Windows.[6] Download and run the installer, and follow the onscreen instructions. When the installation has finished, you should have a directory named `miniconda` in your home directory, and if you choose to add it to your `PATH` variable during the installation, you should now be able to invoke the conda package manager by running `conda` at the command prompt.

Conda[7] is a cross-platform package manager that can handle dependencies on Python packages as well as system tools and libraries. This is essential for installing scientific computing software, which by nature uses a diverse set of tools and libraries. Conda packages are prebuilt binaries for the target platform, and are therefore fast and convenient to install. To verify that conda is available on your system you can try:

```
$ conda --version
conda 3.12.0
```

In this case, the output tells us that conda is installed and that the version of conda is 3.12.0. To update to the latest version of conda, we can use the conda package manager itself:

```
$ conda update conda
```

and to update all packages installed in a particular conda environment, we can use:

```
$ conda update --all
```

Once conda is installed we can use it to install Python interpreters and libraries. When doing so we can optionally specify precise versions of the packages we want to install. The Python software ecosystem consists of a large number of independent projects, each with their own release cycles and development goals, and there are constantly new versions of different libraries being released. This is exciting – because there is steady progress and new features are frequently made available – but unfortunately not all new releases of all libraries are backwards compatible. This presents a dilemma for users that require a stable and reproducible environment over the long term, and for users that simultaneously work on projects with different version dependencies.

The best solution in the Python ecosystem for this problem is to use a package manager such as conda to set up virtual Python environments for different projects, in which different versions of the required packages are installed. With this approach, it is easy to maintain multiple environments with different configurations, such as separate Python 2 and Python 3 environments, or environments with stable versions and development versions of relevant packages. I strongly recommend using virtual Python environments over using the default system-wide Python environments for the reasons given above.

[6]Miniconda is available in both 32- and 64-bit versions. Generally the 64-bit version is recommended for modern computers, but on Windows a 64-bit compiler might not always be readily available, so staying with the 32-bit version might better in some cases on this platform.
[7]`http://conda.pydata.org/docs/index.html`.

With conda, new environments are created with the `conda create` command, to which we need to provide a name for the new environment using `-n NAME`, or alternatively a path to where the environment is to be stored using `-p PATH`. When providing a name, the environment is by default stored in the `miniconda/envs/NAME` directory. When creating a new environment, we can also give a list of packages to install. At least one package must be specified. For example, to create two new environments based on Python 2.7 and Python 3.4, we can use the commands:

```
$ conda create -n py2.7 python=2.7
$ conda create -n py3.4 python=3.4
```

where we have given the Python 2 and Python 3 environments the names `py2.7` and `py3.4`, respectively. To use one of these environments, we need to *activate* it using the command `source activate py2.7` or `source activate py3.4`, respectively, and to *deactivate* an environment we use `source deactivate`.[8] With this method it is easy to switch between different environments, as illustrated in the following sequence of commands:

```
$ source activate py2.7
discarding /Users/rob/miniconda/bin from PATH
prepending /Users/rob/miniconda/envs/py2.7/bin to PATH

(py2.7)$ python --version
Python 2.7.9 :: Continuum Analytics, Inc.

(py2.7)$ source activate py3.4
discarding /Users/rob/miniconda/envs/py2.7/bin from PATH
prepending /Users/rob/miniconda/envs/py3.4/bin to PATH

(py3.4)$ python --version
Python 3.4.2 :: Continuum Analytics, Inc.

(py3.4)$ source deactivate
discarding /Users/rob/miniconda/envs/py3.4/bin from PATH
$
```

To manage environments, the `conda env`, `conda info` and `conda list` commands are helpful tools. The `conda info` command can be used to list available environments (same as `conda env list`):

```
$ conda info --envs
# conda environments:
#
py2.7                    /Users/rob/miniconda/envs/py2.7
py3.4                    /Users/rob/miniconda/envs/py3.4
root                  *  /Users/rob/miniconda
```

[8] On Windows, leave out `source` from these commands.

and the `conda list` command can be used to list installed packages and their versions, in a given environment:

```
$ conda list -n py3.4
# packages in environment at /Users/rob/miniconda/envs/py3.4:
#
openssl                   1.0.1k                        0
python                    3.4.2                         0
readline                  6.2                           2
sqlite                    3.8.4.1                       0
tk                        8.5.15                        0
xz                        5.0.5                         0
zlib                      1.2.8                         0
$
```

and similar information in YAML format[9] is available from the `conda env export` command:

```
$ conda env export -n py3.4
name: py3.4
dependencies:
- openssl=1.0.1k=1
- pip=6.1.1=py34_0
- python=3.4.3=0
- readline=6.2=2
- setuptools=16.0=py34_0
- sqlite=3.8.4.1=1
- tk=8.5.18=0
- xz=5.0.5=0
- zlib=1.2.8=0
```

To install additional packages in an environment, we can either specify a list of packages when the environment is created, or we can activate the environment and use `conda install`, or use the `conda install` command with the `-n` flag to specify a target environment for the installation. For example, to create a Python 2.7 environment with NumPy version 1.8, we could use:

```
$ conda create -n py2.7-np1.8 python=2.7 numpy=1.8
```

To verify that the new environment `py2.7-np1.8` indeed contains NumPy of the specified version, we can use the `conda list` command again:

```
$ conda list -n py2.7-np1.8
# packages in environment at /Users/rob/miniconda/envs/py2.7-np1.8:
#
numpy                     1.8.2                    py27_0
openssl                   1.0.1k                        0
python                    2.7.9                         1
readline                  6.2                           2
sqlite                    3.8.4.1                       0
tk                        8.5.15                        0
zlib                      1.2.8                         0
```

[9] http://yaml.org

Here we see that NumPy is indeed installed, and the precise version of the library is `1.8.2`. If we do not explicitly specify the version of a library, the latest stable release is used.

To use the second method – that is to install additional packages in an already existing environment – we first activate the environment

```
$ source activate py3.4
```

and then use `conda install PACKAGE` to install the package with name `PACKAGE`. Here we can also give a list of package names. For example, to install the NumPy, SciPy and Matplotlib libraries, we can use:

```
(py3.4)$ conda install numpy scipy matplotlib
```

or, equivalently:

```
$ conda install -n py3.4 numpy scipy matplotlib
```

When installing packages using conda, all required dependencies are also installed automatically, and the command above actually also installed the packages `dateutil`, `freetype`, `libpng`, `pyparsing`, `pytz`, and `six` packages, which are dependencies for the `matplotlib` package:

```
(py3.4)$ conda list
# packages in environment at /Users/rob/miniconda/envs/py3.4:
#
dateutil                  2.1                      py34_2
freetype                  2.4.10                        1
libpng                    1.5.13                        1
matplotlib                1.4.2                np19py34_0
numpy                     1.9.1                    py34_0
openssl                   1.0.1k                        0
pyparsing                 2.0.1                    py34_0
python                    2.7.9                         1
pytz                      2014.9                   py34_0
readline                  6.2                           2
scipy                     0.15.0               np19py34_0
six                       1.9.0                    py34_0
sqlite                    3.8.4.1                       0
tk                        8.5.15                        0
zlib                      1.2.8                         0
```

Note that not all of the packages installed in this environment are Python libraries. For example, libpng and freetype are system libraries, but conda is able to handle them and install them automatically as dependencies. This is one to the strengths of conda compared to, for example, the Python-centric package manager `pip`.

To update selected packages in an environment we can use the `conda update` command. For example, to update NumPy and SciPy in the currently active environment, we can use:

```
(py3.4)$ conda update numpy scipy
```

To remove a package we can use `conda remove PACKAGE`, and to completely remove an environment we can use `conda remove -n NAME --all`. For example, to remove the environment `py2.7-np1.8`, we can use:

```
$ conda remove -n py2.7-np1.8 --all
```

Conda locally caches packages that have once been installed. This makes it fast to reinstall a package in a new environment, and also quick and easy to tear down and set up new environments for testing and trying out different things, without any risk of breaking environments used for other projects. To re-create a conda environment, all we need to do is to keep track of the installed packages. Using the `-e` flag with the `conda list` command gives a list of packages and their versions, in a format that is also compatible with the `pip` software. This list can be used to replicate a conda environment, for example, on another system or at later point in time:

```
$ conda list -e > requirements.txt
```

With the file `requirements.txt` we can now update an existing conda environment in the following manner:

```
$ conda install --file requirements.txt
```

or create a new environment that is a replication of the environment that was used to create the `requirement.txt` file:

```
$ conda create -n NAME --file requirements.txt
```

Alternatively, we can use the YAML format dump of an environment produced by `conda env export`:

```
$ conda env export -n NAME > env.yml
```

and in this case we can reproduce the environment using:

```
$ conda env create --file env.yml
```

Note that here we do not need to specify the environment name, since the `env.yml` file also contains this information. Using this method also has the advantage that packages installed using `pip` are installed when the environment is replicated or restored.

A Complete Environment

Now that we have explored the conda package manager, and seen how it can be used to setup environments and install packages, next we cover the procedures for setting up a complete environment with all the dependencies that are required for the material in this book. In the following we use the `py3.4` environment, which was previously created using the command:

```
$ conda create -n py3.4 python=3.4
```

This environment can be activated using

```
$ source activate py3.4
```

Once the target environment is activated, we can install the libraries that we use in this book with the following commands:

```
conda install pip ipython jinja2 tornado pyzmq jsonschema spyder pylint pyflakes pep8
conda install numpy scipy sympy matplotlib networkx pandas seaborn
conda install patsy statsmodels scikit-learn
conda install h5py pytables msgpack msgpack-python cython numba cvxopt
```

```
pip install scikit-monaco
pip install pygraphviz
pip install git+https://github.com/pymc-devs/pymc3
pip install version_information
```

However, one exception is the FEniCS suite of libraries that we used in Chapter 11 to solve partial different equations. The FEniCS libraries have many intricate dependencies, making it difficult to install using this standard approach.[10] For this reason, FEniCS is most easily installed using the prebuilt environments available from the project's web site: http://fenicsproject.org/download. Another good solution for obtaining a complete FEniCS environment can be to use a Docker[11] container with FEniCS preinstalled. See, for example, https://hub.docker.com/r/fenicsproject for more information about this method.

Table A-1 presents a breakdown of the installation commands for the dependencies, on a chapter-by-chapter basis.

Table A-1. *Installation instructions for dependencies for each chapter*

Chapter	Used libraries	Installation

1	IPython, Spyder	`conda install ipython jinja2 tornado pyzmq jsonschema`
		`conda install spyder pylint pyflakes pep8`
		Here `jinja2`, `tornado`, `pyzmq` and `jsonschema` are packages that are required to run the IPython notebook, and `pylint`, `pyflakes`, and `pep8` are code analysis tools that can be used by Spyder.
		With the most recent version of IPython, a package called **`notebook`** is also provided, which contains the IPython Notebook component and its relevant dependencies.
		For converting IPython notebooks to PDF, you also need a working LaTeX installation.
		To bookkeep which versions of libraries that were used to execute the IPython notebooks that accompany this book, we have used IPython extension command `%version_information`, which is available in the `version_information` package that can be installed with `pip`:
		`conda install pip`
		`pip install version_information`

2	NumPy	`conda install numpy`
3	NumPy, SymPy	`conda install numpy sympy`
4	NumPy, Matplotlib	`conda install numpy matplotlib`
5	NumPy, SymPy, SciPy, Matplotlib	`conda install numpy sympy scipy matplotlib`

(*continued*)

[10]However, there are recent efforts to create conda packages for the FEniCS libraries and their dependencies: http://fenicsproject.org/download/installation_using_conda.html. This is currently only available for Linux.
[11]For more information about software container solution Docker, see https://www.docker.com.

Table A-1. (*continued*)

Chapter	Used libraries	Installation
6	NumPy, SymPy, SciPy, Matplotlib, cvxopt	`conda install numpy sympy scipy matplotlib cvxopt`
7	NumPy, SciPy, Matplotlib	`conda install numpy scipy matplotlib`

8	NumPy, SymPy, SciPy, Matplotlib, Scikit-Monaco	`conda install numpy sympy scipy matplotlib pip`
		There is no conda package for scikit-monaco, so we need to install this library using `pip`:
		`pip install scikit-monaco`

9	NumPy, SymPy, SciPy, Matplotlib	`conda install numpy sympy scipy matplotlib`

10	NumPy, SciPy, Matplotlib, NetworkX	`conda install numpy scipy matplotlib networkx pip`
		To visualize NetworkX graphs we also need the Graphviz library (see `http://www.graphviz.org`) and its Python bindings in the `pygraphviz` library.
		`pip install pygraphviz`

11	NumPy, SciPy, Matplotlib, and FEniCS	`conda install numpy scipy matplotlib` For installation of FEniCS, see the prebuilt Python environments that are distributed by the FEniCS project: `http://fenicsproject.org/download`
12	NumPy, Pandas, Matplotlib, Seaborn	`conda install numpy pandas matplotlib seaborn`
13	NumPy, SciPy, Matplotlib, Seaborn	`conda install numpy scipy matplotlib seaborn`
14	NumPy, Pandas, Matplotlib, Seaborn, Patsy, Statsmodels	`conda install numpy pandas matplotlib seaborn patsy statsmodels`
15	NumPy, Matplotlib, Seaborn, Scikit-learn	`conda install numpy matplotlib seaborn scikit-learn`

16	NumPy, Matplotlib, PyMC3	`conda install numpy matplotlib pip`
		`pip install git+https://github.com/pymc-devs/pymc3`
		The last line installs the yet-not-released PyMC3 library, which at the time of writing is still in pre-release status. Nonetheless, the library is already very useful. See `http://pymc-devs.github.io/pymc3` for up-to-date instructions on how to install the library once it has been officially released.

17	NumPy, SciPy, Matplotlib	`conda install numpy scipy matplotlib`

18	NumPy, Pandas, h5py, PyTables, msgpack	`conda install numpy pandas h5py pytables msgpack msgpack-python`
		At the time of writing the `msgpack` and `msgpack-python` conda packages are not available for all platforms. When conda packages are not available, the msgpack library needs to be installed manually, and its python bindings can be installed using `pip`.
		`pip install msgpack-python`

19	NumPy, Matplotlib, Cython, Numba	`conda install numpy matplotlib cython numba`

A list of the packages and their exact versions that were used to run the code included in this book is also available in a `requirements.txt` file, which is available for download together with the code listing. With this file we can directly create an environment with all the required dependencies with a single command:

```
$ conda create -n py2.7 --file requirements.txt
```

Alternatively, we can re-create the `py2.7` and `py3.4` environments using the exports `py2.7-env.yml` and `py3.4-env.yml`. These files are also available together with the source code listings.

```
$ conda env create --file py2.7-env.yml
$ conda env create --file py3.4-env.yml
```

Summary

In this appendix we have reviewed the installation of the various Python libraries used in this book. The Python environment for scientific computing is not a monolithic environment, but rather consists of an ecosystem of diverse libraries that are maintained and developed by different groups of people, following different release cycles and development paces. As a consequence, it can be difficult to collect all the necessary pieces of a productive setup from scratch. In response to this problem, several solutions addressing this situation have appeared, typically in the form of prepackaged Python distributions. In the Python scientific computing community, Anaconda and Canopy are two popular examples of such environments. Here we focused on the conda package manager from the Anaconda Python distribution, which in addition to being a package manager, also allows us to create and to manage virtual installation environments.

Further Reading

If you are interested in creating Python source packages for you own projects, see, for example, `http://packaging.python.org/en/latest/index.html`. In particular, study the `setuptools` library, and its documentation at `http://pythonhosted.org/setuptools`. Using `setuptools`, we can create installable and distributable Python source packages. Once a source package has been created using `setuptools`, it is usually straightforward to create binary conda packages for distribution. For information on creating and distributing conda packages, see `http://conda.pydata.org/docs/build_tutorials/pkgs.html`. See also the `conda-recipes` repository at `github.com`, which contains many examples of conda packages: `http://github.com/conda/conda-recipes`. Finally, `http://www.binstar.org` is a conda package hosting service with many public channels (repositories) where custom-built conda packages can be published and installed directly using the conda package manager. Many packages that are not available in the standard Anaconda channel can be found on user-contributed channels on `binstar.org`.