PyMacLab Documentation

Release 0.90.1-dev

Eric M. Scheffel

September 12, 2012

CONTENTS

1 PyMacLab - Python Macroeconomics Laboratory **1**
- 1.1 Introduction . . . 1
  - 1.1.1 Description . . . 1
  - 1.1.2 First things first . . . 1
  - 1.1.3 Features at a Glance . . . 1
- 1.2 Documentation . . . 2
  - 1.2.1 Introduction . . . 2
  - 1.2.2 Series of Brief Tutorials . . . 2
  - 1.2.3 API Documentation . . . 3
  - 1.2.4 Reference . . . 3
- 1.3 Download & Installation . . . 3
  - 1.3.1 Dependencies . . . 3
  - 1.3.2 Option 1 . . . 4
  - 1.3.3 Option 2 . . . 4
  - 1.3.4 Option 3 . . . 4
- 1.4 Credit & Thanks . . . 4
- 1.5 Online Resources . . . 5

2 What is PyMacLab? **7**

3 PyMacLab Philosophy **9**

4 Macroeconomic Analysis in Python **11**
- 4.1 Advantages of PyMacLab in Python . . . 11
- 4.2 Disadvantages of PyMacLab in Python . . . 13

5 PyMacLab Tutorial Series **15**
- 5.1 Tutorial 1 - Getting started . . . 16
  - 5.1.1 Introduction . . . 16
  - 5.1.2 The PyMacLab DSGE model file . . . 18
  - 5.1.3 A Description of the model file's individual sections . . . 20
  - 5.1.4 Steady States [Closed Form] Section . . . 22
  - 5.1.5 Steady State Non-Linear System [Manual] Section . . . 22
  - 5.1.6 Log-Linearized Model Equations Section . . . 22
  - 5.1.7 Variance-Covariance Matrix Section . . . 23
  - 5.1.8 All sections . . . 23
- 5.2 Tutorial 2 - The Python DSGE instance . . . 24
  - 5.2.1 Introduction . . . 24
  - 5.2.2 Understanding the PyMacLab DSGE model class and its instances . . . 24

5.2.3 Instantiation options for DSGE model instances . . . 25
5.2.4 Working with DSGE model instances . . . 26
5.2.5 DSGE modelling made intuitive . . . 29
5.3 Tutorial 3 - The Python DSGE instance updater methods . . . 31
5.3.1 Introduction . . . 31
5.3.2 One-off alteration of one specific model property . . . 31
5.3.3 Altering many model properties and queued processing . . . 32
5.4 Tutorial 4 - Steady State Solution Methods . . . 34
5.4.1 Introduction . . . 34
5.4.2 Option 1: Using the model's declared FOCs and passing arguments at model instantiation . . . 34
5.4.3 Option 2: Supplying the non-linear steady state system in the model file . . . 36
5.4.4 Option 3: Use the numerical root finder to solve for some steady states and get remaining ones residually . . . 38
5.4.5 Option 4: Use the numerical root finder to solve for steady states with pre-computed starting values . . . 39
5.4.6 Option 5: Finding the steady state by only supplying information in the Closed Form section 41
5.5 Tutorial 5 - Dynamic Solution Methods . . . 42
5.5.1 Introduction . . . 42
5.5.2 The Jacobian and Hessian: A Detour . . . 42
5.5.3 Dynamic Solution Methods - Nth-order Perturbation . . . 43
5.5.4 Choosing the degree of approximation . . . 43
5.6 Tutorial 6 - Simulating DSGE models . . . 45
5.6.1 Introduction . . . 45
5.6.2 Simulating the model . . . 45
5.6.3 Cross-correlation tables . . . 46
5.6.4 Simulating while keeping random shocks fixed . . . 48
5.6.5 Generating impulse-response functions . . . 53

6 Release History **55**
6.1 **0.90.1-dev** (2012-9-12) . . . 55
6.2 **0.89.1** (2012-9-11) . . . 55
6.3 **0.88.1** (2012-9-3) . . . 55
6.4 **0.85** (2012-08-30) . . . 56
6.5 **0.8** (2012-08-21) . . . 56

7 Copyright & License **57**

A Bibliography **59**

Bibliography **61**

Index **63**

CHAPTER

ONE

PYMACLAB - PYTHON MACROECONOMICS LABORATORY

1.1 Introduction

1.1.1 Description

PyMacLab is the Python Macroeconomics Laboratory which currently primarily serves the purpose of providing a convenience framework written in form of a Python library with the ability to solve non-linear DSGE models. At the time of writing these words, the library supports solving DSGE models using 1st and 2nd order perturbation methods which are computed around the steady state. If you want to learn about PyMacLab as quickly as possible, skip reading this and instead start reading through the tutorial series available in the Documentation section to this site.

The library provides wrapper functions for Paul Klein's 1st-order accurate method based on the Schur Decomposition [6] as well a more recently published method by the same author (co-authored with Paul Gomme) which computes 2nd-order accurate solutions without using Tensor Algebra [2] (using the Magnus & Neudecker 1999 definition of the Hessian matrix). PyMacLab possesses the added advantage of being equipped with an advanced model file parser module, similar to the one available in Dynare, which automates cumbersome and error-prone (log-)linearization by hand. PyMacLab is also written entirely in Python, is free and incredibly flexible to use and extend.

1.1.2 First things first

- Documentation at http://www.pymaclab.com or http://packages.python.org/pymaclab/
- Latest development documentation at http://www.development.pymaclab.com
- Latest source tar ball at http://pypi.python.org/pypi/pymaclab/
- Latest bleeding-edge source via git at http://github.com/escheffel/pymaclab
- Source code issues tracker at http://github.com/escheffel/pymaclab/issues/

1.1.3 Features at a Glance

- No "paper-and-pencil" linearization required, done automatically py parsing a DSGE model file.
- Solutions based on analytical computation of Jacobian and Hessian of models using Sympycore.

- DSGE models are Python DSGE class instances, treat them as if they were data structures, pass them around, copy them, stack them into arrays, and work with many of them simultaneously!
- Loop over a DSGE model instance thousands of times to alter the parameter space, each time re-computing the solution.
- Choose from closed form or non-linear steady state solvers or a combination of both.
- Choose from a number of tried and tested perturbation methods, such as Klein's 1st order accurate and Klein & Gomme's 2nd order accurate methods.
- Solving models is as fast as using optimized compiled C or Fortran code, expensive computation of analytical Jacobian and Hessian employs parallelized multi-core CPU approach.
- DSGE example models are provided, including very complex ones such as the one based on Christiano, Eichenbaum and Evans (2001) [8].
- Benefit from a large and growing set of convience methods to simulate models and plot filtered simulated series as well as impulse-response functions.
- Use PyMacLab as a free Python library within a rich and rapidly evolving Python software ecosystem for scientists.
- Enjoy the power, flexibility and extensibility of the Python programming language and the open-source transparency of PyMacLab.
- PyMacLab is free as in freedom and distributed under a Apache 2.0 license.

Note: PyMacLab is currently known to work well but continues to mature. This documentation site is well under way but still work-in-progress. If you have used PyMacLab already and spotted some bugs or felt that some other important features are missing, you can head over to the library's Github repository to submit an Issue item. We are currently in the process of adding more example DSGE model files (and eliminating mistakes in already existing ones). If you have used PyMacLab yourself and want to contribute your own DSGE model files we are happy to include them!

1.2 Documentation

1.2.1 Introduction

What is PyMacLab? This is a succinct introduction to PyMacLab including an explanation of its current features.

Philosophy behind PyMacLab Here I discuss the basic Philosophy behind PyMacLand and what it sets out to do now and in the near future.

Why Macroeconomics in Python? In this section I touch upon the the pros and cons of doing Macroeconomics or scientific computing using Python in general.

1.2.2 Series of Brief Tutorials

1. **`Basic DSGE tutorial`** Brief tutorial on how to use PyMacLab to work with DSGE models.
2. **`PyMacLab DSGE instance tutorial`** Succinct tutorial facilitating the understanding of the DSGE OOP data structure in PyMacLab.
3. **`PyMacLab DSGE instance updater tutorial`** Tutorial on how to use DSGE model instance's intelligent runtime update features.

4. **PyMacLab DSGE steady state solver tutorial** This section illustrates various options available to solve DSGE models' steady state.
5. **PyMacLab DSGE dynamic solver tutorial** This section finally shows how dynamic solution to the PyMacLab DSGE models are obtained.
6. **PyMacLab DSGE simulation and plotting tutorial** Short tutorial on using convenience functions for simulations, IRFs and plotting.
7. **Description of all template DSGE models** Detailed description of all of the template DSGE models which come supplied with PyMacLab.

1.2.3 API Documentation

api_doc The auto-generated documentation of pymaclab's main modules and classes

1.2.4 Reference

linsci_scratch Building a Linux scientific environment from scratch.

Bibliography Reference list of academic articles and books related to the solution of DSGE models or Python programming.

Release History History of current and past releases

1.3 Download & Installation

PyMacLab is known to work with any of Python version greater than or equal to 2.4 and smaller than 3.0. In the future we will consider implementing a compatibility branch for versions of Python greater than or equal to 3.0, once all core dependencies are known to have been migrated as well.

1.3.1 Dependencies

Proper functioning of PyMacLab depends on a number of additional Python libraries already being installed on your system, such as:

- Numpy
- Scipy,
- Sympycore,
- Parallel Python
- Matplotlib
- scikits.timeseries

Sympycore and Parallel Python come distributed with PyMacLab and will be installed along with the main library; the other required Python libraries need to be installed separately before and installation of PyMacLab is attempted. All of the mentioned scientific packages are great libraries by themselves and should be checked out by any serious scientist interested in doing work in Python.

If you want to enjoy a Matlab-style interactive environment in which to execute and inspect DSGE and other data structures, you'd be hard-pressed to pass over the brilliant and now extra features-ladden IPython. When downloading and installing pymaclab using `pip` all of these dependencies should be

installed automatically for you, if they are not already present on your system. Following right below is a list of options users have to install PyMacLab on their Python-ready computers.

If you already have a working Python programming environment with some of the above libraries installed, you may want to consider installing PyMacLab in its own isolated execution environment using virtualenv which would ensure that your existing system Python installation would remain untouched by PyMacLab's setup routine and its dependency resolution.

1.3.2 Option 1

You can download the source code of PyMacLab right here. Alternatively, PyMacLab is also hosted at PyPI and can be installed in the usual way by executing the command inside a Linux shell using `pip`:

```
sudo pip install pymaclab
```

1.3.3 Option 2

Otherwise get the latest source code compressed as a tarball here:

pymaclab.tar.gz

And install it in the usual way by running in a Linux shell the command:

```
sudo python setup.py install
```

1.3.4 Option 3

Alternatively, for the brave-hearted and bleeding-edge aficionados, they can also navigate over to our open Github repository where PyMacLab is currently being maintained, and clone the most up-to-date version and/or nightly build, by having git installed on your system and calling:

```
git clone git://github.com/escheffel/pymaclab.git
```

This will create a new folder called pymaclab containing the latest version of the source code as well as the installation script `setup.py` which you can then use in the usual way to install the module on your system.

1.4 Credit & Thanks

Thanks must go to all members of the Python scientific community without whose efforts projects like PyMacLab would be much harder to implement. We are all standing on the shoulders of giants! Special thanks go to Eric Jones, Travis Oliphant and Pearu Peterson, the founding coders of the Numpy/Scipy Suite which PyMacLab heavily makes use of.

I would also like to give a special mention to Skipper Seabold, lead coder of another unique and outstanding Python library, Statsmodels, who has kindly helped me clean up some of the rough edges of my code. I would also like to thank colleagues at Nottingham University Business School China, especially Gus Hooke and Carl Fey for their kind support.

Last but most certainly not least, my expression of thanks go to my former PhD supervisor Max Gillman who has introduced me to the world of general equilibrium macroeconomics and to monetary macroeconomics more deeply. Similary, many of the lectures once delivered by Martin Ellison formerly at the Economics Department at Warwick now at Oxford made a lasting impression on me.

1.5 Online Resources

Author Homepage:	http://www.ericscheffel.com
Github Homepage:	http://github.com/escheffel/pymaclab
Scipy Homepage:	http://www.scipy.org
Download & PyPI:	http://pypi.python.org/pypi/pymaclab
Python Tutorial:	http://docs.python.org/tutorial/

CHAPTER

TWO

WHAT IS PYMACLAB?

At the moment, PyMacLab is a simple-to-use library enabling users to calibrate and solve DSGE models based on a number of well-known and efficient solution algorithms. It works in the tradition of real business cycle analysis and this permits users to simulate and analyze artificial economies. It is very convenient to use and emphasizes intuition and ease-of-use. It is also free and does not depend on any other proprietary software.

What PyMacLab is not yet is an estimation framework for DSGE models in which data can be used to estimated deep structure parameters. But it is only a small step away from that ability. For instance, implementing a limited-information estimation method minimizing the distance between theoretical impulse responses and those provided by a corresponding SVAR would barely take a week or so to program. Bayesian estimation is not on the list yet and may take a little longer to implement.

CHAPTER

THREE

PYMACLAB PHILOSOPHY

The main motivation or reason for why I decided to write PyMacLab was a relatively simple one. Nowadays in Macroeconomic research, PhD students and researchers often want to solve DSGE models many thousand times over, compare models with each other, and experiment with new methods which make use of existing ones. I therefore wanted to afford myself the convenience of working in an environment in which DSGE models could be treated and worked with in terms of simple but flexible and powerful OOP data structures. In addition, open-source OOP coding in a robust and easy to maintain language with all of the advantages discussed would translate into easy extensibility and quick adaptability to whatever needs may arise.

The whole point of PyMacLab is to view the model template files solely as instantiating information, with flexible model "moulding" turning into the standard mode of interaction thereafter. To be able to loop over a DSGE model and extend its functionality in the future easily due to its modular structure were important motivating aspects driving this project forward. This distinguishes PyMacLab from existing *programs* such as Dynare which feel somewhat as inflexible take-as-is routines with a standardized output. PyMacLab, in contrast, is *not* a program, but instead a Python library providing an abstract DSGE model class, from which users can instantiate as many instances as they like which in turn exhibit an incredible amount of post-instantiation scope for transformability. In Dynare the model solving task after the program has run is often considered finished, in PyMacLab the fun starts after models have been loaded from template model files when researchers change models' properties dynamically at runtime.

CHAPTER

FOUR

MACROECONOMIC ANALYSIS IN PYTHON

PyMacLab, so far, is the only Python library designed with the specific purpose in mind to permit solving DSGE models conveniently. For some this may raise the question of why one would want to make available such a library for the Python programming environment in the first place, especially in light of the fact that many other alternatives already exist. In this section I will attempt to briefly outline reasons for why having and using PyMacLab is of benefit to many potential users and why it fits well into the existing software ecosystem. This section may very well read like your usual list of advantages vs. disadvantages. So let's perhaps first start with the advantages of using PyMacLab in Python.

4.1 Advantages of PyMacLab in Python

Python's growing scientific user community

Perhaps the largest benefit of having access to a library for solving DSGE models programmed in the Python language, is that for various easily identifiable reasons, Python is rapidly turning itself into the language best supplied with ready-to-use libraries aimed at the requirements of a sophisticated scientific community. An important stepping stone in this larger development was the availability of the Numpy/Scipy library suite which has rapidly turned Python into an open-source replacement for proprietary software environments such as Matlab.

But the growing availability of mature scientific libraries has not faltered since and has continued to grow at a dramatic pace. To provide an exhaustive list of all production-ready scientific libraries for Python would be a difficult task to achieve, so I will limit myself to a few well-known and preferred packages:

Library	Description
Numpy	Doing linear algebra and providing an array and matrix data type in Python
Scipy	Library of many scientific routines,such as basic statistics, optimization, filtering, etc.
Statsmodels	More advanced and dedicated library for advanced statistics in Python
Pandas	Library providing a data frame and time series data type and a large number of data methods
MDP Toolkit	A data processing library with wrappers for unsupervised learning routines, etc.
Matplotlib	The Python de facto standard library for all-purpose graphing and plotting
Scik-its.TimeSeries	The first library to provide a convenient library containing an advanced time series data type

This list barely touches the surface of what is currently out there available for free for Python prgrammers wanting to do scientific computing. Even if a specific library does not exist directly, it is usually easy to produce wrappers for traditional and mature libraries originally written in C, C++ or Fortran. A great and

comprehensive reference text book introducing serious scientific computing using Python is Langtangen's "Python Scripting for Computational Science" [7]. This last point brings me straight to the next advantage users can expect to benefit from when choosing to use Python in their scientific work.

Python glues well into traditional scientific languages

Do you come from a Fortran, C or C++ background. Do you still have some old tried and tested routines in source code lying around? Then you will be glad to hear that Python has a number of outstanding tools and built-in properties available which allow you to easily link your existing source code into Python programs allowing them to be called inside Python scripts as if they were normal Python routines. For C++ aficionados, there is Swig, a translation tool turning C++ code into Python modules. Fortran users can make use of the easy-to-use Py2f while C users should have no problems whatsoever using the Ctypes library.

The latter users are perhaps best supported, as the original implementation of Python is actually implemented and written in C itself, which explains why it still does well in terms of execution speed in spite of being a dynamically typed and interpreted languages (more on this later). The lesson to take away from this - and this is something Python as a language is well-known for - is that Python is a great "gluing" language which allows you to work well with a large number of software libraries originally coming from quite disparate software environments/ecosystems. Using tools such as Scikits.mlabwrap or RPy2 you can even interface Python with Mathworks Matlab or Gnu R.

Python is dynamic and interpreted

Python by itself is a programming language like any other such as Java, C++ or C and supports pretty much any functionality these languages are also capable of. What sets it apart is that it is not compiled and linked, but insteady interpreted and thus belongs to the family of scripting languages. This means that compile-time and run-time are woven together in one single environment which makes the programming experience much more seemless, interactive and transparent.

This turns Python into a so-called RAD tool - a rapid application development tool, which dramatically cuts down development time and allows developers to design code which is much easier to read and maintain. Also, Python is a mixed language supporting both OOP and procedural code. Coupled with the Python-specific interactive shell IPython Python programming is just as interactive and dynamic as working in a Matlab interactive environment, only much more powerful and flexible and its abilities stretch far beyond matrix algebra and scientific computing.. Work for a while in an IPython shell and you will know the difference. Although it has not happened yet, it stands to reason to expect that one day a fully-fledged Python compiler may appear, giving developers the choice to compile their programs all the way down to machine code.

PyMacLab in Python encourages learning and extending

Many routines aimed at solving DSGE models often feel like canned algorithms which by their very design encourage use of them as simple and unreflective input-blackbox-output procedures in which the users are mostly concerned with learning the syntactic rules of the program to quickly "get out of it what they need". I feel that for the sake of productivity this is not an entirely wrong or indeed deplorable circumstance, quite to the contrary. But it often does imply that users substitute away from learning and understanding under-the-hood details of implementation which in themselves would be worthwhile try to come to grasp with as a means of learning. PyMacLab, as a result of the language it is written in and the way it chooses to implement DSGE modelling in form of an "intelligent" DSGE model instance, encourages students and researchers to look underneath the hood and to use the structure of the DSGE data type as it exists at any given point in time in order to develop easy-to-add extensions. The open-source, improved readability and maintainability nature of Python and PyMacLab itself further enforce this advantage. Canned routines encourage unreflective use, but does human capital theory not teach us that learning-by-doing is an important aspect of stimulating economic growth?

4.2 Disadvantages of PyMacLab in Python

Python is dynamic and interpreted

The previous stated *advantage* of Python is simultaneously also its disadvantage. In many areas of scientific research in which heavy-duty *number-crunching* and *brute-force* methods prevail, execution speed is usually perceived as a top priority. Python's dynamism comes at the cost of much slower execution speed than comparable source code written in Fortran or C++ compiled all the way down to machine code. However, this last point needs to be qualified in light of what has already been pointed out above. Since Python glues in well with existing traditional programming languages, it is comparatively easy to design Python programs in which CPU-intensive code is simply "outsourced" to a dynamically linked library originally written and compiled in Fortran.

This last remark is particularly relevant when reference is made to the well-known 20/80 rule of computing, stating that for most computer programs 20% of its code uses up 80% of its total execution time. Writing the other 80% of your code in easily maintainable Python source code and the remaining 20% in Fortran or another compiled language is a golden recipe which is advocated and applied by many professional users. Actually, the execution speed vs. development speed is the only real drawback worth the trouble to mention. And given the above recipe and the plausible possibility of one day seeing a real Python compiler, the benefits of Python in scientific computing by far outweigh its drawbacks.

CHAPTER

FIVE

PYMACLAB TUTORIAL SERIES

5.1 Tutorial 1 - Getting started

5.1.1 Introduction

PyMacLab's strength or orginal design goal has been that of providing users with a rich and flexible DSGE data structure (also called class in object-oriented programming speak) which allows them to do lots of interesting things with DSGE models. Don't forget that another stated goal of PyMacLab is to permit researchers to work with DSGE models who know how to and enjoy to use the programming language Python. This tutorial assumes that you already know how to use Python, if you are completely new to it I suggest you consult one of the many tutorials available on the internet to get started with the language. Don't forget that Python is free as in free as freedom, no proprietary software such as Matlab is tainting the freedom of the environment you will be working in when using PyMacLab. The easiest way to get started and explore the features of PyMacLab is to launch a IPython session and to import PyMacLab into it

```
# Import the pymaclab module into its namespace
In [1]: import pymaclab as pm

# Get the version and author's name
In [2]: pm.__version__
'0.8'

# Get the library's author's name
In [3]: pm.__author__
'Eric M. Scheffel'
```

Here we simply have imported the PyMacLab module and inspected some of its attributes, such as the current version numbering as well as them module's author's name. Let's look deeper into the recesses of the module though to better understand who it is organized

```
# Import the pymaclab module into its namespace
In [1]: import pymaclab as pm

# Use the dir() command to view all available attributes and method calls,
# this command returns a list
In [2]: dir(pm)
['OPS',
'__author__',
'__builtins__',
'__date__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__path__',
'__revision__',
'__version__',
'db_graph',
'dsge',
'explain',
'filters',
'ldbs',
'linalg',
'lmods',
'lvars',
'macrolab',
```

```
'make_modfile',
'modedit',
'modfiles',
'modinfo',
'modsolve',
'newDB',
'newMOD',
'newVAR',
'pyed',
'stats',
'texedit']
```

As you can see the module contains a quite a few attributes, many of which are still expermimental and perhaps best not called at this stage. The most mature and arguable most interesting method call is that called `pm.newMOD`, which allows users to instantiate a DSGE model instance, which would be done like so:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: import os

# Define the relative path to your modfiles
In [3]: modpath = "../pymaclab/modfiles/"

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(os.path.join(modpath,"rbc1_res.txt"))

# As an example, check the models computed steady stated
In [5]: print rbc1.sstate
{'betta': 0.99009900990099009,
'c_bar': 2.7560505909330626,
'k_bar': 38.160700489842398,
'y_bar': 3.7100681031791227}
```

Alternatively, you can also test some of the DSGE model files which come supplied with PyMacLab's standard installation. For this to work all you have to do is to import a provided handler module, `pymaclab.modfiles.models`, which contains all of the DSGE models' names and their corresponding full file paths:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
# Import the DSGE models' filepath handle
In [2]: from pymaclab.modfiles import models

#Check all of the available models
In [3]: dir(models)
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'cee',
 'max1',
 'max2',
 'mbc1',
 'merz',
 'model2',
```

```
 'model3',
 'nk_nocapital',
 'rbc1_res',
 'rbc1_ext',
 'rbc2',
 'sims']


# The DSGE models objects in pymaclab.modfiles.models
# are just references to full file paths, i.e.
In [4]: pm.modfiles.models.rbc1_res
'/usr/lib/python2.7/site-packages/pymaclab/modfiles/rbc1_res.txt'

#Instantiate a new DSGE model instance like so
In [5]: rbc1 = pm.newMOD(models.rbc1_res)

#As an example, check the models computed steady stated
In [6]: print rbc1.sstate
{'betta': 0.99009900990099009,
'c_bar': 2.7560505909330626,
'k_bar': 38.160700489842398,
'y_bar': 3.7100681031791227}
```

Now we have already seen some of the power and simplicity we can leverage by using PyMacLab. Before learning some of its additional power, we do however still need to take a quick detour to study the model file `rbc1.txt` which we had to pass as an argument to the `pm.newMOD` method call, as its structure is determined by a set of conventions which are important to adhere to in order to enable PyMacLab to parse and employ the information contained therein correctly and efficiently.

5.1.2 The PyMacLab DSGE model file

In order to be able to load or instantiate your first DSGE model and work with it, you have to make sure to first fill in a so-called PyMacLab DSGE model file. The idea behing this is the same as the Dynare model file which typically ends in .mod. PyMacLab already comes provided with a number of such files pre-compiled for you to experiment with. For instance the most basic real business cycle model is described in the model file `rbc1.txt` which looks as follows

```
%Model Description++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
This is just a standard RBC model, as you can see.


%Model Information+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Name = Standard RBC Model;


%Parameters++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
rho       = 0.36;
delta     = 0.025;
R_bar     = 1.01;
eta       = 2.0;
psi       = 0.95;
z_bar     = 1.0;
sigma_eps = 0.052;


%Variable Vectors++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
[1]  k(t):capital{endo}[log,bk]
[2]  c(t):consumption{con}[log,bk]
[4]  y(t):output{con}[log,bk]
[5]  z(t):eps(t):productivity{exo}[log,bk]
[6]  @inv(t):investment[log,bk]
[7]  @R(t):rrate

%Boundary Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variable Substitution Non-Linear System++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @inv(t)   = k(t)-(1-delta)*k(t-1);
[2]   @inv_bar  = SS{@inv(t)};
[2]   @F(t)     = z(t)*k(t-1)**rho;
[2]   @Fk(t)    = DIFF{@F(t),k(t-1)};
[2]   @Fk_bar   = SS{@Fk(t)};
[2]   @F_bar    = SS{@F(t)};
[3]   @R(t)     = 1+DIFF{@F(t),k(t-1)}-delta;
[4]   @R_bar    = SS{@R(t)};
[3]   @R(t+1)   = FF_1{@R(t)};
[4]   @U(t)     = c(t)**(1-eta)/(1-eta);
[5]   @MU(t)    = DIFF{@U(t),c(t)};
[5]   @MU_bar   = SS{@U(t)};
[6]   @MU(t+1)  = FF_1{@MU(t)};



%Non-Linear First-Order Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Insert here the non-linear FOCs in format g(x)=0

[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;


%Steady States [Closed Form]++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   y_bar = @F_bar;


%Steady State Non-Linear System [Manual]++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @F_bar-@inv_bar-c_bar = 0;
[2]   betta*@R_bar-1 = 0;
[3]   betta*R_bar-1 = 0;

[1]   c_bar = 1.0;
[2]   k_bar = 1.0;
[3]   betta = 0.9;

%Log-Linearized Model Equations++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variance-Covariance Matrix++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sigma = [sigma_eps**2];
```

```
%End Of Model File++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

So what does this file mean, and in particular, what is the meaning and purpose of the individual sections? These and related questions are addressed in the sections to follow below. They mostly discuss the syntax conventions model builders have to adhere to in order to use PyMacLab correctly.

5.1.3 A Description of the model file's individual sections

Model Description Section

In the model description section of the model file you can use plain text in order to described more verbosely the type of the model summarized in the file, perhaps added with references to important academic journal articles in which the model first appeared.

Information Section

This section allows you to add more succinct model properties, including a shorter denominator given by *Name=* qualifier. These shorter attributes will then be attached to the model instance where they help to uniquely identify the model. In contrast to the information contained in the previous section these qualifiers should be short.

Parameters Section

As the name suggests, this section provides space for writing down the model's deep and presumably invariable parameters which are important as they appear in functionals such as the household's utility or the firm's production function. Don't forget to close each declaration with a semi-colon, as this is one of the text parser's conventions.

Variable Vectors Section

This section is very important as it contains a summary of all of the (time-subscripted) variables of the model. The general format of this section for each variable is:

```
[1] x(t):var_name{endo|con|exo}[log,hp|bk,cf]
```

The first element is a descriptor of how the time-subscripted variable will appear in the system of nonlinear equations. The second descriptor is a more revealing but still short name, such as *capital* or *consumption*. It is preferable to write longer variable names with an underscore, such as for example *physical_capital* or *human_capital*. Thirdly, the descriptor in curly brackets allows you to specifically mark of each variable as either, control variable, endogenous state or exogenous state variable, using optimal control theory language. These are inserted in abbreviated style using either *con*, *endo* or *exo*. Finally, the last option given enclosed in squared brackets allows for two additional options to be specified. Supplying the keyword *log* means that the approximation of the model showed be formed about the log of the variable, while the last option allows to supply a filtering option which is applied to the computation of results based on simulations of the solved model. Currently available choices are either *hp* for the HP-Filter, *bk* for the Baxter-King-Filter or *cf* for the Christiano-Fitzgerald filter.

Boundary Conditions Section

This section is currently not in use but has been included for future compatibility with solution methods which are not based on the perturbation paradigm.

Variable Substitution Non-Linear System

This is perhaps one of the most useful and convenient sections of the model file. In the section right after this one users are asked to insert the DSGE model's firs-order conditions of optimality which can often be quite tedious and long algebraically. One way of giving users a more convenient and intuitive way of writing down the model's FOCs is to work with a subsitution system which can be declared in this section.

So for example if one wanted to write down the expression for output or the Euler equation for physical capital, one could resort to the following useful replacement definitions:

```
[1]   @inv(t)    = k(t)-(1-delta)*k(t-1);
[2]   @F(t)      = z(t)*k(t-1)**rho;
[3]   @F_bar     = SS{@F(t)};
[4]   @R(t)      = 1+DIFF{@F(t),k(t-1)}-delta;
[5]   @R(t+1)    = FF_1{@R(t)};
[6]   @U(t)      = c(t)**(1-eta)/(1-eta);
[7]   @MU(t)     = DIFF{@U(t),c(t)};
[8]   @MU(t+1)   = FF_1{@MU(t)};
```

These can then be used in the following section instead of having to work with the full expressions instead. Additionally, convience operators are accessible, given by:

```
DIFF{EXPRESSION,x(t)}  # replaced by first derivate if expression w.r.t. x(t)

SS{EXPRESSION}         # expression is converted to its steady state equivalent

FF_X{EXPRESSION}       # replaced with expression forwarded in time by X periods.
                       # Timing of the information set for expectations is unchanged!

BB_X{EXPRESSION}       # replaced with expression lagged in time by X periods.
                       # Timing if the information set for expectations is unchanged!
```

When declaring replacement items in this section make sure to adhere to the syntax of always naming them beginning with a @. Also, within this section substitutions within substitutions are permitted. Replacement items for steady-state calculations in the subsequent sections can also be supplied here, but have to be of the form such as:

```
[1]   @F_bar   = z_bar*k_bar**rho;
```

In PyMacLab steady state expressions of variables strictly have to adhere to the *x_bar* naming convention, i.e. be expressed by the stem variable name abbreviation followed by and underscore and the word *bar*.

Non-Linear First-Order Conditions Section

In this section users can supply the model's first order conditions of optimality which are passed to PyMacLab for differentiation and evaluation. So to use the example from the RBC1 example file given above, filling in this section would look as follows:

```
[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;
```

where we have made ample use of the convenient substitution definitions declared in the previous section. Expressions, such as the law of motion for the productivity shock, can be supplied in logs for the sake of readability, but otherwise could also alternatively be written as:

```
[4]   E(t)|z(t+1)/(z(t)**psi) = 0;
```

Deprecated since version 0.85: In previous versions of PyMacLab it was possible to write down the law of motion of exogenous states without expectations, i.e. *z(t)/(z(t-1)**psi) = 0;*. This behaviour is now deprecated and no longer supported.

5.1.4 Steady States [Closed Form] Section

For relatively simple models, closed form solutions for the steady state may exist and can be entered here as follows:

```
betta   = 1.0/R_bar;
k_bar   = ((rho*z_bar)/(R_bar - 1 + delta))**(1.0/(1 - rho));
y_bar   = (z_bar*k_bar)**rho;
c_bar   = y_bar - delta*k_bar;
```

Note that not only steady-state variables like *x_bar* can be supplied here, but indeed any variable who's steady-state value has to be determined endogenously withing the model. Sometimes, depending on the model builder's assumptions taken, this could also involve the' determination of a parameter such as *betta*. Sometimes the model's full steady-state can best be determined as a combination of closed form expressions AND the additional numerical solution of a system on nonlinear equations, as is the case in the model file provided above. Notice that here one set of steady state variables are calculated in closed from, given the knowledge of a set of other steady state variables, while these in turn are first solved for in the section using the nonlinear root-finding algorithm.

5.1.5 Steady State Non-Linear System [Manual] Section

In this section a partial list of or the entire model's variables' steady states can be determined numerically here using good starting values and a Newton-like root-finder algorithm. So this section would something like this:

```
[1]   z_bar*k_bar**(rho)-delta*k_bar-c_bar = 0;
[2]   rho*z_bar*k_bar**(rho-1)+(1-delta)-R_bar = 0;
[3]   (betta*R_bar)-1 = 0;
[4]   z_bar*k_bar**(rho)-y_bar = 0;

[1]  c_bar = 1.0;
[2]  k_bar = 1.0;
[3]  y_bar = 1.0;
[4]  betta = 1.0;
```

Very often, this section is simply a restatement of the first order conditions of optimality but with time subscripts removed and instead replaced with the steady state *x_bar* notation. This section and the previous can often be the most difficult ones to specify well, as many more complex DSGE models' steady states are not easy to determine and often require some good judegement, experience and good starting values for the root-finding algorithm.

5.1.6 Log-Linearized Model Equations Section

In this section you could theoretically also supply the first-order log-linearized equations manually, such as was necessary in Harald Uhlig's toolbox. But this feature is perhaps best relegated to compatibility tests and proof-of-concept experiments to show that PyMacLab's computed solutions based on automatic differentiation are identical with the ones computed from this section. An example would be:

```
# foc consumption
[1]   (1/C_bar)**Theta*X_bar**(Psi*(1-Theta))*x(t)...
     -(1/C_bar)**Theta*X_bar**(Psi*(1-Theta))*c(t)=...
       LAM_bar*lam(t)+A_bar*MU_bar*mu(t);
# foc leisure
[2]   (1-Theta)*c(t)+(Psi*(1-Theta)-1)*x(t)=lam(t)+...
       z(t)+(1-alpha)*k(t-1)-(1-alpha)*l(t);
```

In this case all variables already have to be interpreted as percentage deviations from steady state. Both in this and in the nonlinear FOCs section, model equations DO NOT necessarily have to be expressed as *g(x)=0*, but can also be written as *f(x)=g(x)*. In this case the PyMacLab parser simply internally generates *f(x)-g(x) = 0* and works with this expression instead.

5.1.7 Variance-Covariance Matrix Section

The standard way of supplying information on the variance-convariance structure of the iid shocks hitting the laws of motions of the exogenous state variables. So this section would look something like this:

```
Sigma = [sigma_eps**2];
```

or for more elaborate models like this:

```
Sigma = [sigma_eps**2   0;
          0    sigma_xi**2];
```

5.1.8 All sections

If in any of the lines of one of the sections the keyword *None* is inserted, even in a section which has otherwise been declared in the correct way as described above, then the entire section will be ignored and treated as empty, such as for instance:

```
%Log-Linearized Model Equations++++++++++++++++++++++++++++
None
```

If alebraic expression become to long, one can also employ a line-breaking syntax using the elipsis, such as:

```
[1]    (1-Theta)*c(t)+(Psi*(1-Theta)-1)*x(t)=lam(t)+...
        z(t)+(1-alpha)*k(t-1)-(1-alpha)*l(t);
```

Finally, as is customary from other programming languages, comments can also be inserted into DSGE model files. However, in contrast to other languages conventions, such as Python itself, at the moment the library will only parse model files correctly if the comments are on a line of their own, and not intermingled with model description items. As usual comments are identified by beginning a new line with the hash symbol #.

Finally, in all sections where it may be applicable, the operators *LOG(x)* and *EXP(x)* can be employed, where the former takes the natural logarithm of expression x while the latter raises e to the power x. An example of this would be:

```
[1]    @U(t)    = LOG(c(t));
```

5.2 Tutorial 2 - The Python DSGE instance

5.2.1 Introduction

As already stated in the introduction of the introductory basic tutorial, PyMacLab's strength or orginal design goal has been that of providing users with a rich and flexible DSGE data structure (also called *Class* in object-oriented programming speak) which allows them to do lots of interesting things with DSGE models and to treat them as if they were some kind of primitive *data type* in their own right. While the previous tutorial described some basics as well as the all-important DSGE model file structure and syntax conventions, in this section I am going to stress some of the object-oriented programming features of PyMacLab, in particular the structure of a PyMacLab DSGE model *instance* or data structure.

Readers with a background in modern programming languages supporting the object-oriented programming (OOP) paradigm will easily relate to concepts in this sections, while for others it may appear more cryptic at first sight. But to convey these concepts to researchers is important, as it stresses many particular advantages of PyMacLab over other programs, and in particular its *flexibility*, *transparency*, *consistency*, *persistence* and enormous scope for *extensibility*. All example code fragments provided here assume that you are replicating them within an IPyton interactive session, but they could also be called from a Python program "batch" file.

5.2.2 Understanding the PyMacLab DSGE model class and its instances

PyMacLab has been written in the Python programming language which supports object-oriented programming. This means, that more than 80% of PyMacLab's code is devoted to the definition of *data fields* and *methods* of the *DSGE_model Class*, which forms the basis for all DSGE models users can load, spawn or *instantiate* and interact with once they have imported they PyMacLab library into their programs. As already explained elsewhere, the basis of all DSGE model *instances* is the DSGE model's model text file in which it is defined in terms of its specific characteristics, such as its parameters and first-order conditions of optimality. We recall that this process of loading or *instantiating* a DSGE model worked as follows:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1)
```

After executing these lines of code in an interactive environment such as that provided by IPython, which emulates well the feel and behaviour of the Matlab interactive environment, the DSGE instance or data object going by the name of *rbc1* now exists in the namespace of the running program and can be put to further use. But before looking at these various ways possible to make effective use of this DSGE model instance, let's first trace the various steps the programs goes through when a DSGE model get instantiated. So what happens internally when the above last line in the code fragment is called:

1. The empty shell DSGE model instance gets instantiated
2. The DSGE model instance reads the model file provided to it and any other arguments and saves them by attaching them to itself.
3. Instantiation Step 1: The files get read in and a method defined on the instance simply splits the file into its individual sections and saves these raw sections by attaching them to itself.
4. Instantiation Step 2: A parser method is called which disaggregates the raw information provided in each section of the model file and begins to extract meaningful information from it, each time saving this information by attaching it to itself as data fields. Also, the DSGE model instance is prepared

and set up in order to attempt to solve for the steady state of it manually at the command line, instead of doing it automatically. If you want the model instance to do ONLY this next step and stop there for you to explore further interactively, you must call the command with and extra argument like this:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so, but adding initlev=0 as extra argument
In [3]: rbc1 = pm.newMOD(models.rbc1,initlev=0)
```

5. Instantiation Step 3: The information is used in order to attempt to compute the numerical steady-state of the model. If you want the model instance to do ONLY this next step and stop there for you to explore further interactively, you must call the command with and extra argument like this:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so, but adding initlev=1 as extra argument
In [3]: rbc1 = pm.newMOD(models.rbc1,initlev=1)
```

6. Instantiation Step 4: If the steady state was computed successfully then the model's analytical and numerical Jacobian and Hessian are computed. Finally a preferred dynamic solution method is called which solves the model for its policy function and other laws of motion.

To give users a choice of "solution depths" at DSGE object instantiation time is important and useful, especially in the initial experimentation phase during which the DSGE model file gets populated. That way researchers can first carefully solve one part of the problem (i.e. looking for the steady state) and indeed choose to do so manually on the IPython interactive command shell, allowing them to immediately inspect any errors.

5.2.3 Instantiation options for DSGE model instances

There are a couple of instance invocation or instantiation arguments one should be aware of. At the time of writing these lines there are in total 5 other arguments (besides the DSGE model template file path) which can be passed to the pymaclab.newMOD function out of which 1 is currently not (yet) supported and not advisable to employ. The other 4 options determine the initiation level of the DSGE model (i.e. how far it should be solved if at all), whether diagnosis messages should be printed to screen during instantiation, how many CPU cores to employ when building the Jacobian and Hessian of the model, and finally whether the expensive-to-compute Hessian should be computed at all. Remember that the last option is useful as many researchers often - at least initially - want to explore the solution to their model to a first order of approximation before taking things further. So here are the options again in summary with their default values:

Option with default value	Description
`pm.newMOD(mpath,initlev=2)` Initlev=1 does initlev=0 and attempts to solve for the model's steady state automatically	Initlev=0 only parses and prepares for manual steady state calculation
Initlev=2 does initlev=0, initlev=1 and generates Jacobian and Hessian and solves model dynamically	
`pm.newMOD(mpath,mesg=False)`	Prints very useful runtime instantiation messages to the screen for users to follow progress
`pm.newMOD(mpath,ncpus=1)`	CPU cores to be used in expensive computation of model's derivatives, 'auto' for auto-detection
`pm.newMOD(mpath,mk_hessian=True)`	Should Hessian be computed at all, as is expensive?
`pm.newMOD(mpath,use_focs=False)`	Should only the model's FOCs be used to computed the steady state? Accepts Python list or tuple
`pm.newMOD(mpath,ssidic=None)`	Use in conjunction with previous argument to specify initial starting values as Python dictionary

Needless to say, all of the options can be and usually are called in combination, they are only shown separately here for sake of expositional clarity. Medium-sized to large-sized models can take considerable time to compute the Jacobian alone, let alone the Hessian. On the other hand passing more (real as opposed to virtual) CPU cores to the instantiation process can significantly cut down computation time. In this case, the FOCs nonlinear equations are distributed to individual cores for analytical differentiation as opposed to doing this serially on one CPU core.

5.2.4 Working with DSGE model instances

The most useful feature is to call the model with the option *initlev=0*, because this will allow you more control over the steady-state computation of the model by permitting a closer interactive *inspection* of the DSGE model instance as created thus far. Let's demonstrate this here:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so, but adding initlev=0 as extra argument
In [3]: rbc1 = pm.newMOD(models.rbc1,initlev=0)

# This datafield contains the original nonlinear system expressed as g(x)=0
In [4]: rbc1.sssolvers.fsolve.ssm
['z_bar*k_bar**(rho)-delta*k_bar-c_bar',
 'rho*z_bar*k_bar**(rho-1)+(1-delta)-R_bar',
 '(betta*R_bar)-1',
 'z_bar*k_bar**(rho)-y_bar']

# This datafield contains the initial values supplied to the rootfinder algorithm
In [5]: rbc1.sssolvers.fsolve.ssi
{'betta': 1.0, 'c_bar': 1.0, 'k_bar': 1.0, 'y_bar': 1.0}

# Instead of letting the model during instantiation solve the model all the way through,
# we can solve for the steady state by hand, manually
```

```
In [6]: rbc1.sssolvers.fsolve.solve()

# And then inspect the solution and some message returned by the rootfinder
In [6]: rbc1.sssolvers.fsolve.fsout
{'betta': 0.9900990099009901,
 'c_bar': 2.7560505909330626,
 'k_bar': 38.1607004898424,
 'y_bar': 3.7100681031791227}

In [7]: rbc1.sssolvers.fsolve.mesg
'The solution has converged.'
```

Another useful lesson to take away from this example is that a DSGE model instance is like a many-branch tree structure, just like the Windows File Explorer so many people are familiar with, where individual "nodes" represent either data fields or methods (function calls) which equip the model instance with some functionality. This kind of approach of structuring and programming a solution to the problem of designing a program which handles the solution-finding of DSGE models offers enormous scope for experimentation and extensibility. After a DSGE model has been instantiated without passing the *initlev* argument, you can inspect this structure like so:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [3]: rbc1 = pm.newMOD(models.rbc1)

# Inspect the data fields and methods of the DSGE model instance
In [4]: dir(rbc1)
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_initlev',
 'audic',
 'author',
 'ccv',
 'dbase',
 'deltex',
 'getdata',
 'info',
 'init2',
 'manss_sys',
 'mkeigv',
```

```
 'mkjahe',
 'mkjahen',
 'mkjahenmat',
 'mkjahepp',
 'mkjaheppn',
 'mod_name',
 'modfile',
 'nall',
 'ncon',
 'nendo',
 'nexo',
 'nlsubs',
 'nlsubs_list',
 'nlsubs_raw1',
 'nlsubs_raw2',
 'nother',
 'nstat',
 'numssdic',
 'paramdic',
 'pdf',
 'setauthor',
 'ssidic',
 'sssolvers',
 'sstate',
 'ssys_list',
 'subs_vars',
 'switches',
 'texed',
 'txted',
 'txtpars',
 'updf',
 'updm',
 'vardic',
 'vreg']
```

As you can see, the attributes exposed at the root of the instance are plenty and can be acccessed in the usual way:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [3]: rbc1 = pm.newMOD(models.rbc1)

# Access one of the model's fields
In [4]: rbc1.ssys_list
['z_bar*k_bar**(rho)-delta*k_bar-c_bar',
'rho*z_bar*k_bar**(rho-1)+(1-delta)-R_bar',
'(betta*R_bar)-1',
'z_bar*k_bar**(rho)-y_bar']
```

So one can observe that the data field `rbc1.ssys_list` simply summarizes the system of nonlinear equations which has been described in the relevant section of the DSGE model file. Now you know how to explore the DSGE model instance and understand its general structure, and we conclude this short tutorial by inviting you to do so. Don't forget that some nodes at the root possess further sub-nodes, as was the case when cascading down the `rbc1.sssolvers` branch. To help your search, the only other node with many more sub-nodes is the `rbc1.modsolvers` branch, which we will explore more in the next

section to this tutorial series.

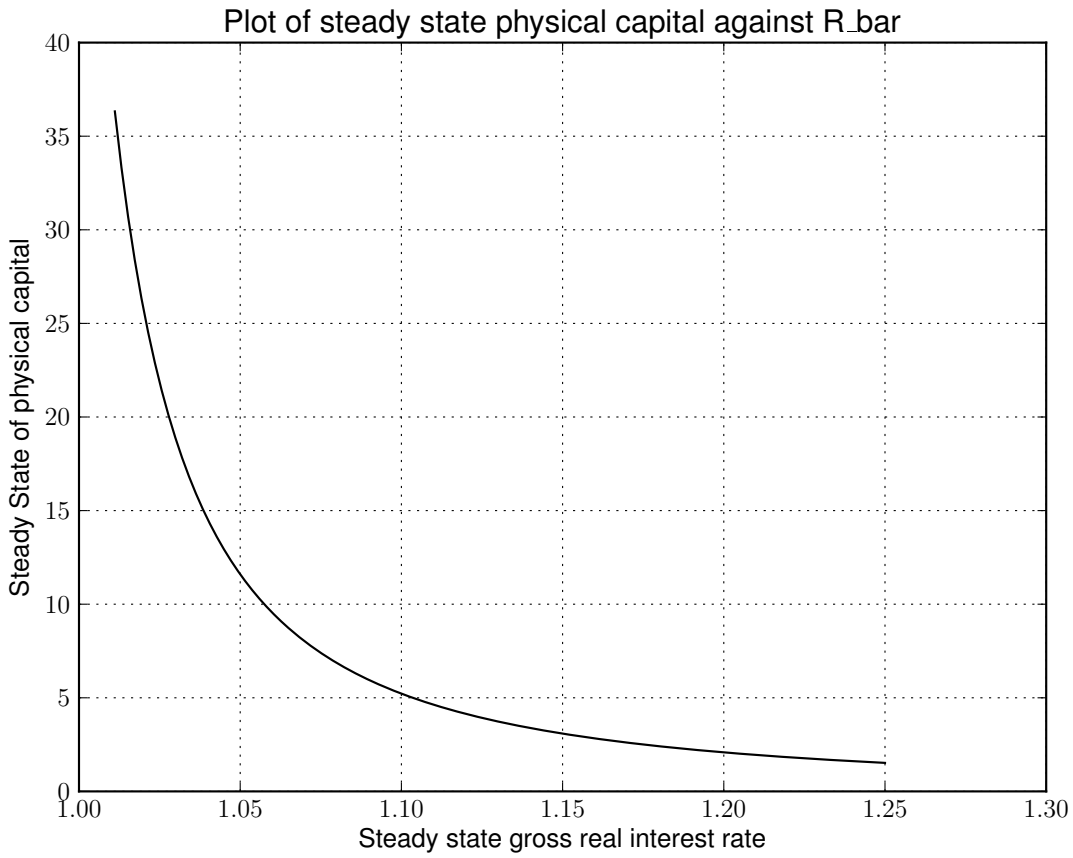
5.2.5 DSGE modelling made intuitive

Before concluding this tutorial, we will demonstrate how PyMacLab's DSGE data structure (or instance) approach allows researchers to implement ideas very intuitively, such as for instance "looping" over a DSGE model instance in order to explore how incremental changes to the parameter space alter the steady state of the model. Leaving our usual interactive IPyton shell, consider the following Python program file:

```
# Import the pymaclab module into its namespace
# Also import Numpy for array handling and Matplotlib for plotting
import pymaclab as pm
from pymaclab.modfiles import models
import numpy as np
from matplotlib import pyplot as plt

# Instantiate a new DSGE model instance like so
rbc1 = pm.newMOD(models.rbc1)

# Create an array representing a finely-spaced range of possible impatience values
# Then convert to corresponding steady state gross real interest rate values
betarr = np.arange(0.8,0.99,0.001)
betarr = 1.0/betarr

# Loop over the RBC DSGE model, each time re-computing for new R_bar
ss_capital = []
for betar in betarr:
    rbc1.updaters.paramdic['R_bar'] = betar # assign new R_bar to model
    rbc1.sssolvers.fsolve.solve() # re-compute steady stae
    ss_capital.append(rbc1.sssolvers.fsolve.fsout['k_bar']) # fetch and store k_bar

# Create a nice figure
fig1 = plt.figure()
plt.grid()
plt.title('Plot of steady state physical capital against R\_bar')
plt.xlabel(r'Steady state gross real interest rate')
plt.ylabel(r'Steady State of physical capital')
plt.plot(betarr,ss_capital,'k-')
plt.show()
```

Anybody who has done some DSGE modelling in the past will easily be able to intuitively grasp the purpose of the above code snippet. All we are doing here is to loop over the same RBC model, each time feeding it with a slightly different steady state groos real interest rate value and re-computing the steady state of the model. This gives rise to the following nice plot exhibting the steady state relationship between the interest rate and the level of physical capital prevailing in steady state:

That was nice and simple, wasn't it? So with the power and flexibility of PyMacLab DSGE model instances we can relatively painlessly explore simple questions such as how differing deep parameter specifications for the impatience factor β can affect the steady state level of physical capital. And indeed, as intuition would suggest, less patient consumers are less thrifty and more spend-thrifty thus causing a lower steady state level of physical capital in the economy. This last example also serves to make another important point. PyMacLab is *not* a *program* such as Dynare, but instead an add-in *library* for Python prividing an advanced DSGE model data structure in form of a DSGE model class which can be used in conjunction with any other library available in Python.


Plot of steady state physical capital against R_bar
Steady State of physical capital
Steady state gross real interest rate
0
5
10
15
20
25
30
35
40
1.00
1.05
1.10
1.15
1.20
1.25
1.30

5.3 Tutorial 3 - The Python DSGE instance updater methods

5.3.1 Introduction

In the previous tutorial we described the general structure and some of the behaviour of PyMacLab's DSGE model's object-oriented design and what kind of advantages model builders can derive from this. In particular, at the end of the last tutorial, we saw how the decision to design DSGE model's instances essentially in from of an object-oriented advanced data structure with well-defined data fields and methods allowed us to "loop over" a DSGE model instance many times over, each time feeding the instance with slightly a different value for the impatience factor β and obtaining a new value for the corresponding steady state value of physical capital.

To be able to do something like this *simply* and *intuitively* is one of the many advantages that PyMacLab has over traditional DSGE model solving *programs* such as Dynare. While Dynare is a complete *program* allowing users to solve for specific DSGE models with little added post-solution programmatic flexibility, PyMacLab is from ground up designed to be a library providing an object-oriented Class-defined advanced data structure with data fields and behaviour in form of methods. This makes it incredibly easy to build programs which treat DSGE models as if they were simple data structures, such as integers, floats, lists or any other data structure you are familiar with from using a number of different programming languages. This makes PyMacLab incredibly flexible for within- and post-solution *program interventions*, where in this section we will define more clearly what we meant by *program interventions*. The main point to take away from all of this is that Dynare is a canned program to get specific solutions, while PyMacLab is a Python plugin library, whose only purpose is to make available an advanced DSGE model class suitable for carrying out a variety of tasks.

While in programs like Dynare the most important aspect from the point of view of model builders to obtaining the solution of DSGE models lies in the specification of so-called model files, the proper way to understand the more flexible operation of PyMacLab is to view the model template only as a specific *point of departure* from which to start your analysis from. The only point of making use of model template files is to initialize or instantiate a DSGE model instance, the real power in using PyMacLab lies in the methods made available to researchers which become available *after* model files have been read in and the DSGE model instances become available inside the Python interactive environment. It is this post-instantiation scope for activity and alteration which makes using PyMacLab so much fun for researchers. As already illustrated in the previous tutorial, this design decision opens up the possibility of easily and intuitively obtaining a large permutation of possible solution outcomes quickly. There are two convenience methods or avenues open to the researcher to intelligently update DSGE model instances dynamically at runtime which we will describe in some detail next.

5.3.2 One-off alteration of one specific model property

Once a DSGE model is instantiated using a model template files as a point of departure and a specific source of information from which model properties get parsed and attached as data fields to the DSGE model instance, all we have done is to initialize or read into memory a specific *state* of our DSGE model. You may also recall that the nature of this *state* differs crucially depending upon what kind of *initlev* parameter value was passed at DSGE model instantiation. So we recall that at *initlev=0* only information gets parsed in but nothing at all gets solved, not even the steady state. Other values for *initlev* gave rise to instantiated model instances which were solved all down to ever deeper levels including possibly including the steady state and dynamic elasticities (policy functions). In this section I will briefly describe methods allowing a functionality akin to comparative statics analysis in PyMacLab in which models can get easily be re-initialized with different model information injected into existing DSGE model instances dynamically at runtime.

One way of using PyMacLab to carry out a kind of comparative statics analysis has already been touched upon for one specific case in the previous tutorial, using the example of the

`rbc1.updaters.paramdic` attribute. In general you will discover that navigating into the `rbc1.updaters.` branch of the model exposes a number of data fields which in exact name are also available at the root `rbc1.` of the model. So how then does `rbc1.updaters.paramdic` differ from its counterpart available at the root `rbc1.paramdic`?

The answer to this question is that while `rbc1.paramdic` is just the parameter dictionary which get populated at model instantiation time, `rbc1.updaters.paramdic` is a "wrapped" version of the same dictionary with the added behaviour that if a new values gets inserted into it using the relevant dictionary methods, such as `rbc1.updaters.paramdic['R_bar']=1.02` or `rbc1.updates.paramdic.update(dic_alt)` where *dic_alt* is some other dictionary containing updated values of one or more keyed item in the original paramdic, then an internal function gets triggered at assignment time which re-initializes the model using the updated values for paramdic.

If the newly assigned value is exactly identical to the value which was already stored in the original paramdic, then the model will not get updated as its state has remained unaltered. The following smart one-off updaters are available all of which possess the above described behaviour. Notice also that the DSGE model will only get updated down to the level first specified in *initlev* at model instantiation time. Also, when any of these wrapped updater objects just gets called in the normal way they behave exactly as their non-wrapped counterparts by returning the values stored in them without any additional behaviour updating the model.

Wrapped updater object	Description
`model.updaters.paramdic`	Inserting updated values re-initializes the model with new set of parameter values
`model.updaters.vardic`	Inserting new values updates/changes values and attributes defined for variables used in the model
`model.updaters.nlsubsdic`	Inserting new values updates the dictionary of variable substitution items beginning with @
`model.updaters.focedic`	Inserting new string equations updates the set of non-linear first-order conditions of optimality
`model.updaters.mansslv`	In this list users can insert new equations as strings into the closed form steady state system
`model.updaters.ssys_list`	In this list users can insert new equations as strings into the numerical steady state system
`model.updaters.sigma`	A wrapped matrix which updates the model when changed values are inserted in the varcovar-matrix

Notice how in this branch `rbc1.updaters.` changing wrapped objects by assigning new values will trigger automatic updating of the DSGE model immediately upon assignment. This behaviour may not always be desirable whenever a series of changes need to be made before updating of the model can be considered. Whenever such situations occur an alternative route needs to be taken which we will explore next.

5.3.3 Altering many model properties and queued processing

At times researchers may want to load or instantiate a particular DSGE model instance using a corresponding template file but then perhaps plan to radically modify the model dynamically at runtime, by combining such actions as introducing new time-subscripted variables, altering the deep parameter space and adding new or augmenting existing equations in the system of non-linear FOCs. Whenever such radical alterations are considered, they will often have to happen in combindation *before* the model gets updated using the new information passed to it. In this case users will use the same wrapped objects already described above but instead use them in the `rbc1.updaters_queued.` branch.

Here, first a number of changes can be made to objects such as `rbc1.updaters_queued.paramdic`

or `rbc1.updaters_queued.foceqs`, etc. which by themselves will *not* trigger an automatic model updating functionality. Instead all changes will be put into a queue which will then have to be processed manually by calling the method `rbc1.updaters_queued.process_queue()` after all desired changes have been made. This addscenormous flexibility to model builders' options, as they can essentially build a completely new model at runtime dynamically starting from a simple model instantiated at the outset of their Python scripts/batch files. Therefore, this functionality allows users to dynamically update all information at runtime which was first parsed from the model template file, each time re-computing the DSGE model's new state given the changes made after the call to the queue processing method has been made.

Wrapped updater object	Description
`model.updaters_queued.parameters`	Inserting updated values re-initializes the model with new set of parameter values
`model.updaters_queued.variables`	Inserting new values updates/changes values and attributes defined for variables used in the model
`model.updaters_queued.nlss_vars`	Inserting new values updates the dictionary of variable substitution items beginning with @
`model.updaters_queued.foceqs`	Inserting new string equations updates the set of non-linear first-order conditions of optimality
`model.updaters_queued.marss_eqs`	In this list users can insert new equations as strings into the closed form steady state system
`model.updaters_queued.ssys_eqs`	In this list users can insert new equations as strings into the numerical steady state system
`model.updaters_queued.sigma`	A wrapped matrix which updates the model when changed values are inserted in the varcovar-matrix
`model.updaters_queued.queue`	The actual queue. Here objects which have been altered will be stored as strings
`model.updaters_queued.process_queue()`	The queue processing method which finally updates the queued objects in the right order

5.4 Tutorial 4 - Steady State Solution Methods

5.4.1 Introduction

In the previous tutorial we learnt how a PyMacLab DSGE model instance possesses the capability to intelligently upate its properties following the re-declaration at runtime of attached data fields such as the parameter space or the set of non-linear first-order conditions of optimality. In this section we will learn an important component to PyMacLab's DSGE models which provides users with a large number of options available for solving models' steady state solution. The great number of possible avenues to take here is quite deliberate; it would be reasonable to argue that for medium- to large-sized models the most difficult part to finding the general dynamic solution based on the approximation method of perturbations is to first obtain the steady state solution around which the approximations are computed. In total we are going to explore 5 different variants suitable for seeking to compute the steady state. So let's get started.

5.4.2 Option 1: Using the model's declared FOCs and passing arguments at model instantiation

Choosing option one allows users to leave the numerical as well as closed form steady state sections in the model template files entirely empty or unused indicated by the "None" keyword inserted into any line in these sections. In this case, the library has to rely on the time- subscripted non-linear first-order conditions of optimality, convert them to steady state equivalents and somehow discover the required set of initial guesses for the variables' steady states to be searched for using the non-linear root-finding algorithm. This is accomplised in the following way. Consider first the following simple example of a DSGE model file:

```
%Model Description++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
This is just a standard RBC model, as you can see.


%Model Information+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Name = Standard RBC Model;


%Parameters++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
rho       = 0.36;
delta     = 0.025;
R_bar     = 1.01;
betta     = 1/R_bar;
eta       = 2.0;
psi       = 0.95;
z_bar     = 1.0;
sigma_eps = 0.052;


%Variable Vectors++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]  k(t):capital{endo}[log,bk]
[2]  c(t):consumption{con}[log,bk]
[4]  y(t):output{con}[log,bk]
[5]  z(t):eps(t):productivity{exo}[log,bk]
[6]  @inv(t):investment[log,bk]
[7]  @R(t):rrate

%Boundary Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None
```

```
%Variable Substitution Non-Linear System++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @inv(t)    = k(t)-(1-delta)*k(t-1);
[2]   @inv_bar   = SS{@inv(t)};
[3]   @F(t)      = z(t)*k(t-1)**rho;
[4]   @F_bar     = SS{@F(t)};
[5]   @R(t)      = 1+DIFF{@F(t),k(t-1)}-delta;
[6]   @R_bar     = SS{@R(t)};
[7]   @R(t+1)    = FF_1{@R(t)};
[8]   @U(t)      = c(t)**(1-eta)/(1-eta);
[9]   @MU(t)     = DIFF{@U(t),c(t)};
[10]  @MU(t+1)   = FF_1{@MU(t)};



%Non-Linear First-Order Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Insert here the non-linear FOCs in format g(x)=0

[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;


%Steady States [Closed Form]++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Steady State Non-Linear System [Manual]++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Log-Linearized Model Equations++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variance-Covariance Matrix++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sigma = [sigma_eps**2];


%End Of Model File+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Notice how we have left the usual sections employed to supply information useful for finding the steady state unused indicated by inserting the keyword "None". As you can see by inspecting the system of non-linear first order conditions, a steady state could be obtained by passing a steady state version of the FOCs to the non-linear root-finding algorithm, with the additional qualifier that in this particular case we would ideally like to omit passing the last line which is just a declaration of the own-lagged law of motion of the exogenous state productivity shock **[#f1]_**. This would lead to a 3 equation system in *c_bar*, *k_bar* and *y_bar*. Further more, we would have to let the model somehow know the set of intial guesses for these three variables, which we often tend to set to some generic values, such as 1.0 for all three of them. How is all of this accomplished? By passing the relevant variables directly to the DSGE model at instantiation time like so:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Define the ssidic of initial guesses or starting values
In [3]: ssidic = {}
```

```
In [4]: ssidic['c_bar'] = 1.0
In [5]: ssidic['k_bar'] = 1.0
In [6]: ssidic['y_bar'] = 1.0

# Instantiate a new DSGE model instance like so
In [7]: rbc1 = pm.newMOD(models.rbc1_ext,use_focs=[0,1,2],ssidic=ssidic)
```

The default value passed to the DSGE model instance's argument "use_focs" is *False*, the alternative value is a zero-indexed Python list (or tuple) indicating the equations of the declared system of FOCs to use in finding the steady state numerically. In the case of the model file given here, we don't want to use the last line of 4 equations and thus set the list equal to *[0,1,2]*. We also define a dictionary of initial starting values or guesses for the three steady state values we wish to search for and pass this as a value to the argument *ssidic*. This method has the added advantage that steady state initial starting values can be determined intelligently at runtime external to the model file.

5.4.3 Option 2: Supplying the non-linear steady state system in the model file

Yet another way available for finding the model's steady state is similar to the one in option one in that it uses a system of non-linear equations specified in this case directly inside the model template file. The reason why one would want to prefer this option over option one has to do with the fact that the steady state version of the non-linear first-order conditions of optimality can often collapse to much easier to work with and succincter equations which the model builder would want to write down explicitly inside the model file. So this example would be exemplified by the following model template file:

```
%Model Description++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
This is just a standard RBC model, as you can see.


%Model Information+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Name = Standard RBC Model;


%Parameters+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
rho       = 0.36;
delta     = 0.025;
R_bar     = 1.01;
eta       = 2.0;
psi       = 0.95;
z_bar     = 1.0;
sigma_eps = 0.052;


%Variable Vectors++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]  k(t):capital{endo}[log,bk]
[2]  c(t):consumption{con}[log,bk]
[4]  y(t):output{con}[log,bk]
[5]  z(t):eps(t):productivity{exo}[log,bk]
[6]  @inv(t):investment[log,bk]
[7]  @R(t):rrate

%Boundary Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variable Substitution Non-Linear System++++++++++++++++++++++++++++++++++++++++++++
[1]   @inv(t)   = k(t)-(1-delta)*k(t-1);
```

```
[2]   @inv_bar  = SS{@inv(t)};
[2]   @F(t)     = z(t)*k(t-1)**rho;
[2]   @Fk(t)    = DIFF{@F(t),k(t-1)};
[2]   @Fk_bar   = SS{@Fk(t)};
[2]   @F_bar    = SS{@F(t)};
[3]   @R(t)     = 1+DIFF{@F(t),k(t-1)}-delta;
[4]   @R_bar    = SS{@R(t)};
[3]   @R(t+1)   = FF_1{@R(t)};
[4]   @U(t)     = c(t)**(1-eta)/(1-eta);
[5]   @MU(t)    = DIFF{@U(t),c(t)};
[5]   @MU_bar   = SS{@U(t)};
[6]   @MU(t+1)  = FF_1{@MU(t)};


%Non-Linear First-Order Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Insert here the non-linear FOCs in format g(x)=0

[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;


%Steady States [Closed Form]++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Steady State Non-Linear System [Manual]++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @F_bar-@inv_bar-c_bar = 0;
[2]   betta*@R_bar-1 = 0;
[3]   betta*R_bar-1 = 0;
[4]   y_bar-@F_bar = 0;

[1]   c_bar = 1.0;
[2]   k_bar = 1.0;
[3]   y_bar = 1.0;
[4]   betta = 0.9;

%Log-Linearized Model Equations++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variance-Covariance Matrix++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sigma = [sigma_eps**2];


%End Of Model File+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

As one can see easily in this case, we are instructing the model to solve the 4 equation system in the four variables *c_bar*, *k_bar*, *y_bar* and *betta*. This is also a very common option to choose in order to obtain the model's steady state efficiently and conveniently.

5.4.4 Option 3: Use the numerical root finder to solve for some steady states and get remaining ones residually

Option 3 perhaps one of the most useful ways one can employ in order to obtain a DSGE model's steady state solution as it focuses the numerical non-linear root-finding algorithm on a very small set of equations and unknown steady state variables, leaving the computation of the remaining steady state variables to be done separately and residually after the small set of steady state variables have been solved for. So using again a slightly tweaked version of the model file given in option 2 we could write this as:

```
%Model Description++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
This is just a standard RBC model, as you can see.


%Model Information+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Name = Standard RBC Model;


%Parameters+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
rho       = 0.36;
delta     = 0.025;
R_bar     = 1.01;
eta       = 2.0;
psi       = 0.95;
z_bar     = 1.0;
sigma_eps = 0.052;


%Variable Vectors++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]  k(t):capital{endo}[log,bk]
[2]  c(t):consumption{con}[log,bk]
[4]  y(t):output{con}[log,bk]
[5]  z(t):eps(t):productivity{exo}[log,bk]
[6]  @inv(t):investment[log,bk]
[7]  @R(t):rrate

%Boundary Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variable Substitution Non-Linear System++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @inv(t)   = k(t)-(1-delta)*k(t-1);
[2]   @inv_bar  = SS{@inv(t)};
[2]   @F(t)     = z(t)*k(t-1)**rho;
[2]   @Fk(t)    = DIFF{@F(t),k(t-1)};
[2]   @Fk_bar   = SS{@Fk(t)};
[2]   @F_bar    = SS{@F(t)};
[3]   @R(t)     = 1+DIFF{@F(t),k(t-1)}-delta;
[4]   @R_bar    = SS{@R(t)};
[3]   @R(t+1)   = FF_1{@R(t)};
[4]   @U(t)     = c(t)**(1-eta)/(1-eta);
[5]   @MU(t)    = DIFF{@U(t),c(t)};
[5]   @MU_bar   = SS{@U(t)};
[6]   @MU(t+1)  = FF_1{@MU(t)};



%Non-Linear First-Order Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Insert here the non-linear FOCs in format g(x)=0
```

```
[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;


%Steady States [Closed Form]+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   y_bar = @F_bar;


%Steady State Non-Linear System [Manual]++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @F_bar-@inv_bar-c_bar = 0;
[2]   betta*@R_bar-1 = 0;
[3]   betta*R_bar-1 = 0;

[1]   c_bar = 1.0;
[2]   k_bar = 1.0;
[3]   betta = 0.9;

%Log-Linearized Model Equations+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variance-Covariance Matrix+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sigma = [sigma_eps**2];


%End Of Model File+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

In this case we have simply taken the equation for *y_bar* outside of the section passed on to the non-linear root-finder and instead included it into the section for closed form steady state expressions. Whenever a model is instantiate like this, it first attempts to solve the smaller steady state system in the *Manual* section, before turning to the *Closed Form* section in which remaining steady states are computed residually based on the subset of steady states already solved numerically in the first step.

This is an extremely useful way of splitting down the problem, as many complex DSGE models often possess a large number of such residually determinable steady state values, while the *core* system on non-linear equations in a subset of steady states can be kept small in dimension and thus easier to solve. This really keeps the iteration burden on the non-linear solver to a minimum and often also allows the researcher to be less judicious in his choice of starting values leaving them at the generic default values. As a general rule, passing ever more complex and larger-dimensioned non-linear systems to the root-finding algorithm will decrease the chances of finding a solution easily, especially when simple generic starting values are employed. The issue of starting values take us straight to the next available option available to PyMacLab users.

5.4.5 Option 4: Use the numerical root finder to solve for steady states with pre-computed starting values

It is often useful and sometimes even outright necessary to supply the root-finding algorithm with pre-computed "intelligently" chosen initial starting values which are better than the generic choice of just passing a bunch of 1.0s to the system. To this end, whenever the model encounters exactly the same variable declarations in the closed form section as those in the list of generic starting values given in the *Manual* section passed to the root-finder, these starting values automatically get replaced by the computed suggestions found in the *Closed Form* section. So an example of this would be:

```
%Model Description++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
This is just a standard RBC model, as you can see.


%Model Information+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Name = Standard RBC Model;


%Parameters++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
rho       = 0.36;
delta     = 0.025;
R_bar     = 1.01;
eta       = 2.0;
psi       = 0.95;
z_bar     = 1.0;
sigma_eps = 0.052;


%Variable Vectors++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]  k(t):capital{endo}[log,bk]
[2]  c(t):consumption{con}[log,bk]
[4]  y(t):output{con}[log,bk]
[5]  z(t):eps(t):productivity{exo}[log,bk]
[6]  @inv(t):investment[log,bk]
[7]  @R(t):rrate

%Boundary Conditions++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variable Substitution Non-Linear System++++++++++++++++++++++++++++++++++++++++++++
[1]   @inv(t)   = k(t)-(1-delta)*k(t-1);
[2]   @inv_bar  = SS{@inv(t)};
[2]   @F(t)     = z(t)*k(t-1)**rho;
[2]   @Fk(t)    = DIFF{@F(t),k(t-1)};
[2]   @Fk_bar   = SS{@Fk(t)};
[2]   @F_bar    = SS{@F(t)};
[3]   @R(t)     = 1+DIFF{@F(t),k(t-1)}-delta;
[4]   @R_bar    = SS{@R(t)};
[3]   @R(t+1)   = FF_1{@R(t)};
[4]   @U(t)     = c(t)**(1-eta)/(1-eta);
[5]   @MU(t)    = DIFF{@U(t),c(t)};
[5]   @MU_bar   = SS{@U(t)};
[6]   @MU(t+1)  = FF_1{@MU(t)};



%Non-Linear First-Order Conditions++++++++++++++++++++++++++++++++++++++++++++++++++
# Insert here the non-linear FOCs in format g(x)=0

[1]   @F(t)-@inv(t)-c(t) = 0;
[2]   betta*(@MU(t+1)/@MU(t))*@R(t+1)-1 = 0;
[3]   @F(t)-y(t) = 0;
[4]   LOG(E(t)|z(t+1))-psi*LOG(z(t)) = 0;


%Steady States [Closed Form]++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   k_bar = 10.0;
```

```
[2]   y_bar = @F_bar;
[3]   c_bar = y_bar - delta*k_bar;
[4]   betta = 1/(1+@Fk_bar-delta);


%Steady State Non-Linear System [Manual]++++++++++++++++++++++++++++++++++++++++++++++++++
[1]   @F_bar-@inv_bar-c_bar = 0;
[2]   betta*@R_bar-1 = 0;
[3]   betta*R_bar-1 = 0;
[4]   y_bar-@F_bar = 0;

[1]   c_bar = 1.0;
[2]   k_bar = 1.0;
[3]   y_bar = 1.0;
[3]   betta = 0.9;

%Log-Linearized Model Equations++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
None


%Variance-Covariance Matrix++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sigma = [sigma_eps**2];


%End Of Model File++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

As is apparent, in this case the suggested values for the steady states given in the closed form section exactly mirror or overlap with the steady variables to be searched for using the non-linear root finder specified in the *Manual* section in the model file. Whenever this overlap is perfect, the values in the *Closed Form* section will always be interpreted as suggested starting values passed on to the non-linear root finder. Notice that in this case it is also possible to omit the additional specification of the generic starting values in the *Manual* section alltogether. However it is advisable to leave them there to give the program a better way of checking the overlap of the two sets of variables. Whenever they are omitted, this specific case of computing the steady state is triggered whenever the number of suggested starting values in the *Closed Form* section is exactly equal to the number of non-linear equations in the *Manual* section.

5.4.6 Option 5: Finding the steady state by only supplying information in the Closed Form section

This is the most straightforward but at the same time possibly also least-used method for finding a steady state and will not be explained in greater depth here. In this variant, the *Manual* section is marked as unused employing the "None" keyword and only information in the *Closed Form* section is provided. Since only the most simple DSGE models afford this option of finding the steady state, we will not discuss this option any further.

5.5 Tutorial 5 - Dynamic Solution Methods

5.5.1 Introduction

In the previous tutorial we discovered the general structure of the PyMacLab DSGE model instance and saw how this programming approach lent itself well to the idea of inspecting and exploring the instantiated models' current state, summarized by its data fields and supplied instance methods equipping them with functionality. This section finally discusses how the library equips DSGE model instances with methods which make use of the models' computed Jacobian and Hessian, which are evaluated at the models' numerical steady state. As a short reminder, we may recall here that it is often this step of obtaining a steady state can prove difficult in the case of a number of well-known models. Notwithstanding, for the remainder of this section we will assume that a steady state has successfully been attained and that the model's Jacobian and Hessian have been computed. Let's first start with our usual setup of lines of code entered into an IPython shell:

```
# Import the pymaclab module into its namespace
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1)
```

Since we have not supplied the instantiation call with the *initlev* argument, the model has been solved out completely, which includes the computation of a preferred dynamic solution, which in the current library's version is Paul Klein's 1st-order accurate method based on the Schur Decomposition. This means that one particular solution - if existing and found - has already been computed by default. Readers familar with this solution method will know that this method requires a partitioning of the model's evaluated Jacobian into two separate matrices, which for simplicity we denote *A* and *B*. They are computed by default and are for instance inspectable at `rbc1.jAA` and `rbc1.jBB`.

5.5.2 The Jacobian and Hessian: A Detour

One of the great advantages of PyMacLab is that the Jacobian and Hessian are NOT computed or approximated numerically using the method of finite differences, but are calculated in exact fashion analytically using the special-purpose Python library Sympy, which is a CAS - or computer algebra system, similar in functionality to Mathematica and Maple. You can inspect the analytical counterparts to the exact numerical Jacobian and Hessian which have not yet been evaluated numerically at `rbc1.jdicc` and `rbc1.hdicc`, where the latter reference actually refers to the 3-dimensional analytical Hessian. If the model is composed of *n* equations describing equilibrium and optimality conditions, then the Jacobian is made up of $n \times (n \times 2)$ elements and has dimension $\{n, n \times 2\}$, because the derivatives are formed not only w.r.t. to current-period, but also future-period variables. Equally, the Hessian is a 3-dimensional matrix of dimension $\{n, n \times 2, n \times 2\}$ **[#f1]_**.

Notice that for the numerical counterpart to the 3-dimensional Hessian, PyMacLab instead uses an alternatively dimensioned version of dimension $\{n \times n \times 2, n \times 2\}$, which is the Magnus and Neudecker definition of a Hessian and is useful when one wishes to avoid using matrices of dimension larger than 2 and the corresponding Tensor notation. Again, you can exploit PyMacLab's DSGE model instance's design in order to inspect the derivatives contained in the Jacobian and Hessian inside an IPython interactive shell environment, such as follows:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models
```

```
# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1)


# Check some elements of the analytical Jacobian and Hessian
# Jacobian, equation 3, derivative w.r.t. k(t-1). Notice that Python arrays are 0-indexed.
In [5]: rbc1.jdicc[2]['k(t-1)']
'rho*exp(z(t))*exp(k(t-1)*rho)'

# You could also retrieve the evaluated equivalent of the above expression.
# Here you need to know the position of k(t-1) in the ordered list of derivatives, you can check
# that k(t-1) is on position 5 by inspecting rbc1.var_li
In [6]: rbc1.numj[2,5]
1.3356245171444843

# Hessian, equation 3, 2nd derivative w.r.t k(t-1).
In [6]: rbc1.hdicc[2]['k(t-1)']['k(t-1)']
'rho**2*exp(z(t))*exp(k(t-1)*rho)'

# The numerical evaluated equivalent can be retrieved as well.
# We are not retrieving the above value via rbc1.numh[2,5,5] as we are not working with
# the usual 3D notation of Hessians, but with the Magnus & Neudecker 2D definition of it.
# The result is correct, as the 2nd derivative is just rho(=0.36) times the first derivate.
In [7]: rbc1.numh[21,5]
0.48082482617201433
```

Now we have explored the ins and outs of PyMacLab's way of handling the computation of a DSGE model's Jacobian and Hessian. Equipped with these building blocks, it is now time to move on to a discussion of the actual solution methods which PyMacLab provides by default.

5.5.3 Dynamic Solution Methods - Nth-order Perturbation

Solving nonlinear rational expectations DSGE models via the method of perturbation represents an approximate solution around the computed steady state of the model. Since this approach is not too dissimilar from a Taylor Series expansion of a function around some point students learn about in some 101 maths course, it should come as no surprise that here too 1st and higher-order derivatives are needed in order to arrive at solutions which increasingly reflect the true exact solution of the system.

PyMacLab offers methods suitable for computing such approximated solutions based on linearization techniques which can either be 1st- or 2nd-order accurate. In order to obtain these solutions, we make use of Paul Klein's solution algorithms, which are available on the internet and have been incorporated into PyMacLab. Needless to say, Klein's 1st-order accurate method using the Schur Decomposition has been around for a while and only requires knowledge of the models Jacobian, while his latest paper (co-authored with Paul Gomme) spelling out the solution of the 2nd-order accurate approximation, also requires knowledge of the Hessian.

5.5.4 Choosing the degree of approximation

At the time of writing these words, PyMacLab includes full support for both of these two methods, where the first method has been made available by binding Klein's original Fortran code into PyMacLab and making it accessible via the node `rbc1.modsolvers.forkleind` which provides the solution method callable via `rbc1.modsolvers.forkleind.solve()`. Once this method has been called and a solution has been found, it is essentially encapsulated in the matrices available at `rbc1.modsolvers.forkleind.P` and `rbc1.modsolvers.forkleind.F`, which represent matrices of dynamic elasticities summarizing the optimal laws of motion for the set of endogenous state

and control variables, respectively. Since this method is actually internally calling a compiled Fortran dynamically linked library, its name is prefixed with *for*.

Klein & Gomme's 2nd-order accurate method uses the solution from the 1st-order accurate method as a starting point but in addition also makes use of the model's Hessian *and* the information provided by the model's shocks variance-covariance matrix, in order to produce solutions which are *risk-adjusted* in some loosely defined sense. This solution method therefore no longer displays the well-known property of *certainty equivalence* for which first-order approximations are so well known for. At the moment, this solution method is completely implemented in the Python language itself and is callable at `rbc1.modsolvers.pyklein2d.solve()`. As already mentioned, the method makes use of `rbc1.modsolvers.forkleind.P` and `rbc1.modsolvers.forkleind.F`, the variance-covariance matrix `rbc1.modsolvers.pyklein2d.ssigma`, and the Magnus & Neudecker definition of the Hessian `rbc1.modsolvers.pyklein2d.hes`. It's solution is encapsulated in the following objects:

Object	Description
`rbc1.modsolvers.forklein2d.hx`	Matrix of elasticities describing optimal law of motion for endog. state variables, 1st-order part
`rbc1.modsolvers.forklein2d.gx`	Matrix of elasticities describing optimal law of motion for endog. state variables, 1st-order part
`rbc1.modsolvers.forklein2d.hxx`	Matrix of elasticities describing optimal law of motion for endog. state variables, 2nd-order part
`rbc1.modsolvers.forklein2d.gxx`	Matrix of elasticities describing optimal law of motion for endog. state variables, 2nd-order part
`rbc1.modsolvers.forklein2d.hss`	Array of risk-adjustment values for steady states of endogenous state variables
`rbc1.modsolvers.forklein2d.gss`	Array of risk-adjustment values for steady states of control variables

Once the above mentioned matrices are calculated, the solutions to either the 1st-order or 2nd-order accurate approximations are available and can be used by researchers to compute (filtered) simulations as well as impulse-response functions in order to either plot them or generate summary statistics from them. Luckily, neither of this has to be done by hand, as simulation- and IRF-generating methods are already supplied and convenience plotting functions are also readily available. But this will be the topic of our next tutorial in the tutorial series for PyMacLab.

5.6 Tutorial 6 - Simulating DSGE models

5.6.1 Introduction

In the previous tutorial we discovered the general structure of the PyMacLab DSGE model instance and saw how this programming approach lent itself well to the idea of inspecting and exploring the instantiated models' current state, summarized by its data fields and supplied instance methods equipping them with functionality. This section finally discusses how the library equips DSGE model instances with methods which make use of the models' computed Jacobian and Hessian, which are evaluated at the models' numerical steady state. As a short reminder, we may recall here that it is often this step of obtaining a steady state can prove difficult in the case of a number of well-known models. Notwithstanding, for the remainder of this section we will assume that a steady state has successfully been attained and that the model's Jacobian and Hessian have been computed. Let's first start with our usual setup of lines of code entered into an IPython shell:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1_res)
```

5.6.2 Simulating the model

Recall that instantiating a DSGE model without any additional parameters means that an automatic attempt of finding the steady state is being made and if successfully found the model is already solved dynamically using a preferred (1st-order approximate) method. This means that you now have all the information you need to simulate the model as well as to generate impulse-response functions and plot them. Let's focus on the simulations first, they are being generate using the following command:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [3]: rbc1 = pm.newMOD(models.rbc1_res)

# Now simulate the model
In [4]: rbc1.modsolvers.forkleind.sim(200,('productivity'))
```

This would simulate the model for 200 time periods using only iid shocks hitting the law of motion of the total factor productivity term in the model. Notice that here RBC1 is a very simple model only containing one structural shock, but more complicated models may possess more than one exogenous state variable. In that case, if you called instead:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Instantiate a new DSGE model instance like so
In [3]: rbc1 = pm.newMOD(models.rbc1_res)

# Now simulate the model
In [4]: rbc1.modsolvers.forkleind.sim(200)
```

The model would be simulated using all shocks (exogenous state variables) specified in the model. However, since `rbc1` only contains one shock, the two variants shown here of simulating the model would yield the same results as "productivity" is is the only exogenous state available here anyway. We can then also graph the simulation to get a better understanding of the model by running the command:

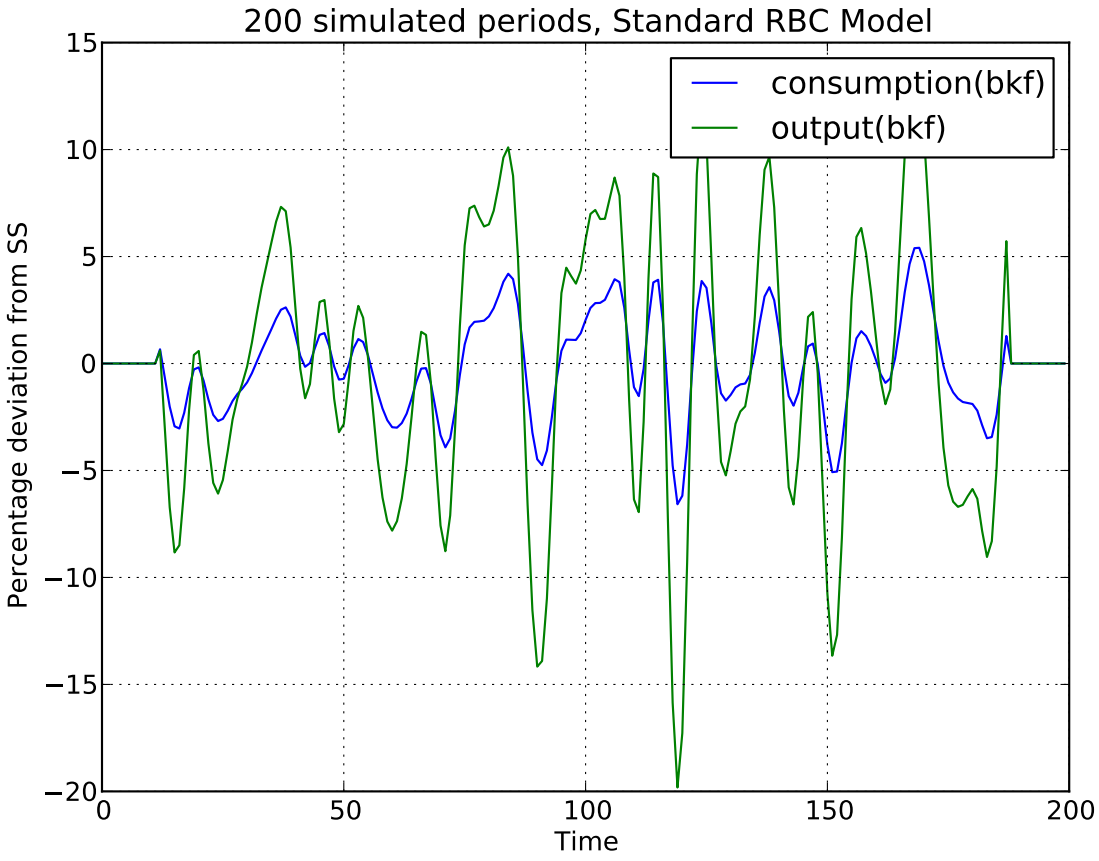
```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Also import matplotlib.pyplot for showing the graph
In [3]: from matplotlib import pyplot as plt

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1_res)

# Now solve and simulate the model
In [5]: rbc1.modsolvers.forkleind.solve()
In [6]: rbc1.modsolvers.forkleind.sim(200)

# Plot the simulation and show it on screen
In [7]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [8]: plt.show()
```

This produces the following nice graph. Notice that you must specify the variables to be graphed and all simulated data is filtered according to the argument passed to each variable in the model file. So the key "hp" produces hp-filtered data, the key "bk" results in Baxter-King-filtered data while the key "cf" leads to cycles extraced using the Christiano-Fitzgerald filter.

5.6.3 Cross-correlation tables

Notice that filtered simulations are always stored in data fields which means that statistics such as correlations at leads and lags can easily be computed as well. Specifically, the simlulated data corresponding to the above graph can be retrieved from the object `rbc1.modsolver.forkleind.insim` [#f1]_. There already exist a number of simple convenience functions allowing users to generate cross-correlation tables for simulated data. The functions can be used as follows:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Also import matplotlib.pyplot for showing the graph
In [3]: from matplotlib import pyplot as plt
In [4]: from copy import deepcopy

# Instantiate a new DSGE model instance like so
In [5]: rbc1 = pm.newMOD(models.rbc1_res)

# Now solve and simulate the model
In [6]: rbc1.modsolvers.forkleind.solve()
In [7]: rbc1.modsolvers.forkleind.sim(200)

# Generate the cross-correlation table and show it
# Produce table with 4 lags and 4 leads using output as baseline
In [8]: rbc1.modsolvers.forkleind.mkact('output',(4,4))
In [9]: rbc1.modsolvers.forkleind.show_act()

Autocorrelation table, current output
```


200 simulated periods, Standard RBC Model
consumption(bkf)
output(bkf)
Percentage deviation from SS
Time
15
10
5
0
−5
−10
−15
−20
0
50
100
150
200

```
============================================================
productivity  |-0.016  0.109  0.335  0.663  0.997  0.619  0.264  0.034 -0.084
capital       |-0.433 -0.429 -0.381 -0.258 -0.024  0.318  0.522  0.599  0.596
consumption   |-0.134 -0.009  0.228  0.587  0.98   0.699  0.404  0.198  0.08
output        |-0.049  0.077  0.308  0.647  1.     0.646  0.305  0.08  -0.039
```

If users wish to obtain the data of the above table directly in order to import them into a different environment more suitable for producing publication-quality tables, the cross-correlation data can be accesssed at `rbc1.modsolvers.forkleind.actm` which is a matrix object of cross-correlations at the leads and lags specified in the previous calling function generating that table data.

5.6.4 Simulating while keeping random shocks fixed

Yet another useful feature to know about is that after each call to `rbc1.modsolvers.forkleind.sim()` the vector of randomly drawn iid shocks gets saved into object `rbc1.modsolver.forkleind.shockvec`. This is useful because when calling the simulation function, we can also pass an existing pre-computed vector of shocks as an argument instead of allowing the call to generate a new draw of random shocks. That way we can keep the random shocks fixed from model run to model run. So this would be accomplished as follows:

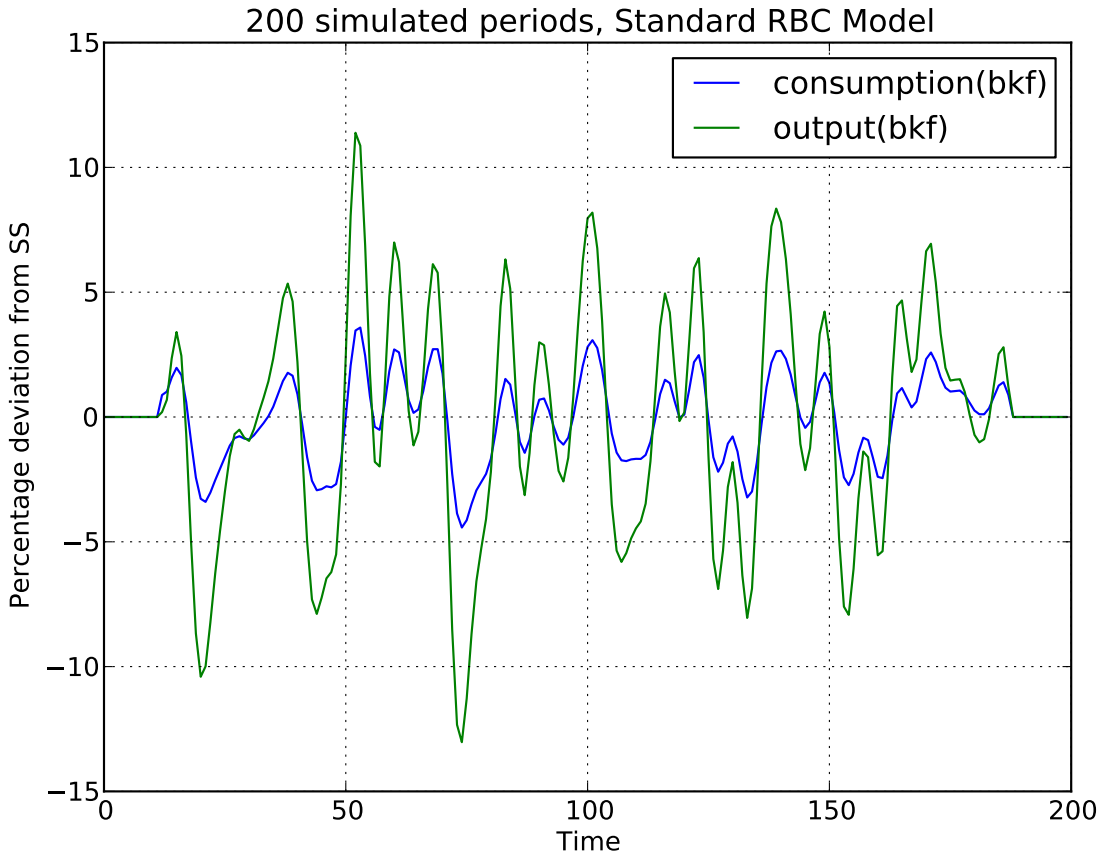
```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Also import matplotlib.pyplot for showing the graph
In [3]: from matplotlib import pyplot as plt
In [4]: from copy import deepcopy

# Instantiate a new DSGE model instance like so
In [5]: rbc1 = pm.newMOD(models.rbc1_res)

# Now solve and simulate the model
In [6]: rbc1.modsolvers.forkleind.solve()
In [7]: rbc1.modsolvers.forkleind.sim(200)

# Plot the simulation and show it on screen
In [8]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [9]: plt.show()

# Now save the shocks, by saving a clone or copy, instead of a reference
In [10]: shockv = deepcopy(rbc1.modsolvers.forkleind.shockvec)

# Now we could run the simulation again, this time passing the randomly drawn shocks
In [11]: rbc1.modsolvers.forkleind.sim(200,shockvec=shockv)

# Plot the simulation and show it on screen
In [12]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [13]: plt.show()
```

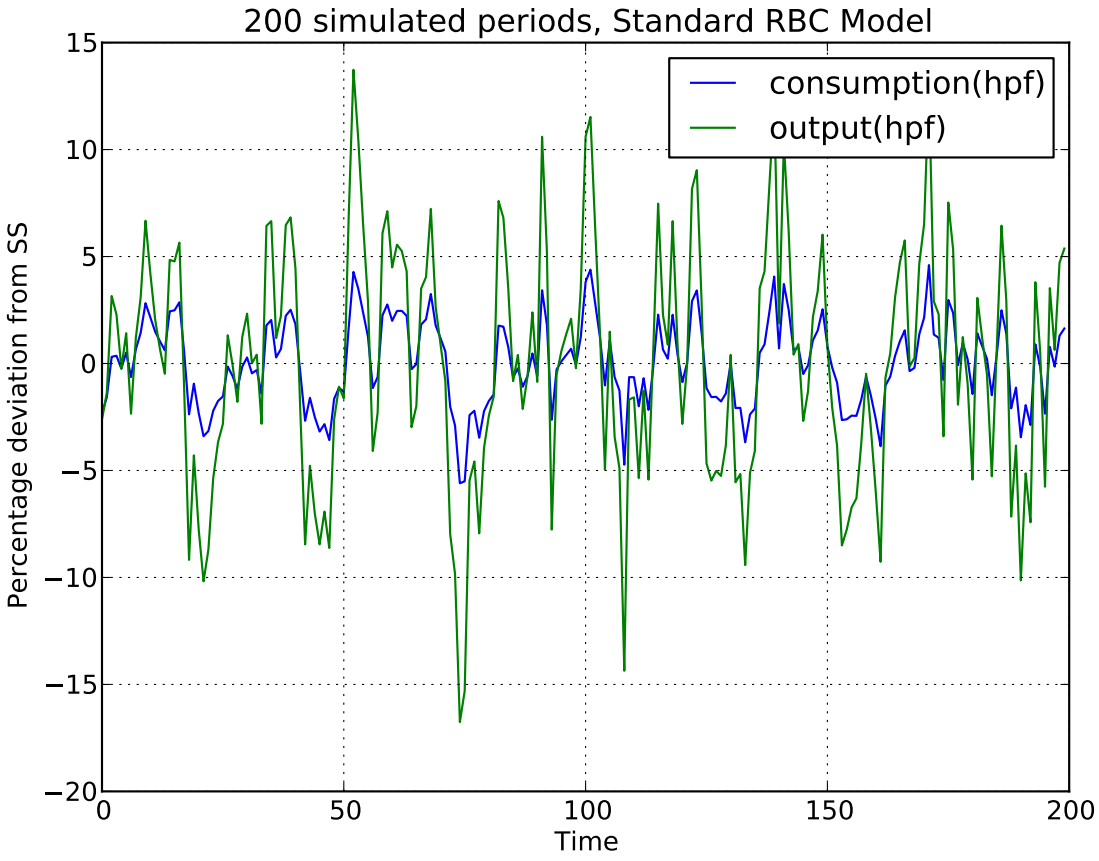
Notice that in this script the graphs plotted to screen using the `plt.show()` command will produce identical graphs as the random draw of shocks only occurs in the first call to `sim()` while in the second it gets passed as an argument with a value retrieved and retained from the first simulation run. The reason why this feature is so useful has to do with the fact that sometimes we wish to produce summary statistics from simulation runs of one version of a model, then tweak the model's properties dynamically at runtime and re-compute the very same summary statistics, under the assumption of holding the iid errors fixed, so that we can observe the pure net effect from changing the model's properties elimiting any unwanted

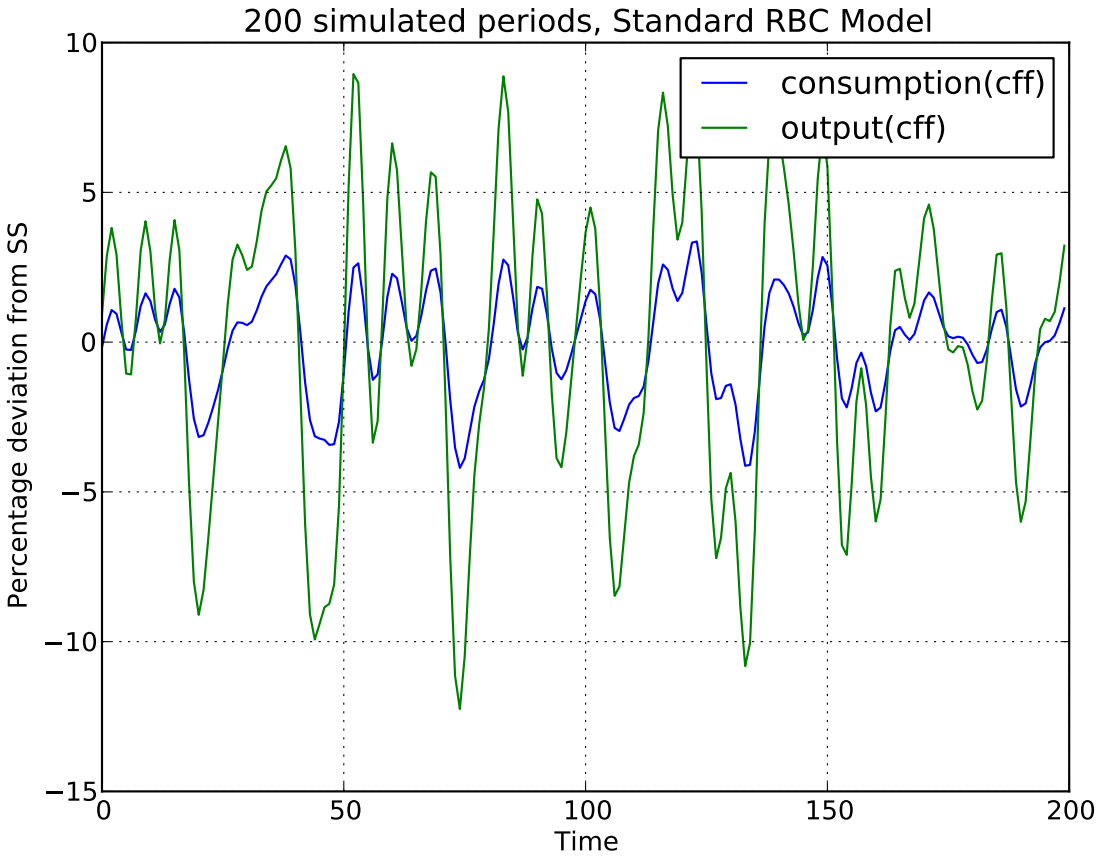
variation from "sampling variation". As an example of this we demonstrate a script in which simulations are run and plotted under different filtering assumption.

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Also import matplotlib.pyplot for showing the graph
In [3]: from matplotlib import pyplot as plt
In [4]: from copy import deepcopy

# Instantiate a new DSGE model instance like so
In [5]: rbc1 = pm.newMOD(models.rbc1_res,mk_hessian=False)

# Now solve and simulate the model
In [6]: rbc1.modsolvers.forkleind.solve()
In [7]: rbc1.modsolvers.forkleind.sim(200)

# Plot the simulation and show it on screen
In [8]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [9]: plt.show()

# Now save the shocks, by saving a clone or copy, instead of a reference
In [10]: shockv = deepcopy(rbc1.modsolvers.forkleind.shockvec)

# Change the filtering assumption of output and consumption using the queued updater branch
In [11]: rbc1.updaters_queued.vardic['con']['mod'][0][1] = 'hp'
In [12]: rbc1.updaters_queued.vardic['con']['mod'][1][1] = 'hp'
In [13]: rbc1.updaters_queued.process_queue()

# Now we could run the simulation again, this time passing the randomly drawn shocks
In [14]: rbc1.modsolvers.forkleind.solve()
In [15]: rbc1.modsolvers.forkleind.sim(200,shockvec=shockv)

# Plot the simulation and show it on screen
In [16]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [17]: plt.show()

# Change the filtering assumption of output and consumption using the queued updater branch
In [18]: rbc1.updaters_queued.vardic['con']['mod'][0][1] = 'cf'
In [19]: rbc1.updaters_queued.vardic['con']['mod'][1][1] = 'cf'
In [20]: rbc1.updaters_queued.process_queue()

# Now we could run the simulation again, this time passing the randomly drawn shocks
In [21]: rbc1.modsolvers.forkleind.solve()
In [22]: rbc1.modsolvers.forkleind.sim(200,shockvec=shockv)

# Plot the simulation and show it on screen
In [23]: rbc1.modsolvers.forkleind.show_sim(('output','consumption'))
In [24]: plt.show()
```

As is apparent from the three plots produced above, the simulated data is first filtered using the Baxter-King filter, then the more commonly used Hodrick-Prescott filter and finally the Christian-Fitzgerald asymmetric filter. Notice that the BK filter by default (or rather by specification) cuts off 6 time periods at the beginning and at the end of the simulated sample. The purpose for using any of the three filters is of course to make the simulated data stationary and to extract the cycle only.


200 simulated periods, Standard RBC Model
consumption(bkf)
output(bkf)
Percentage deviation from SS
Time
15
10
5
0
−5
−10
−15
0
50
100
150
200


200 simulated periods, Standard RBC Model
consumption(hpf)
output(hpf)
Percentage deviation from SS
Time
15
10
5
0
−5
−10
−15
−20
0
50
100
150
200

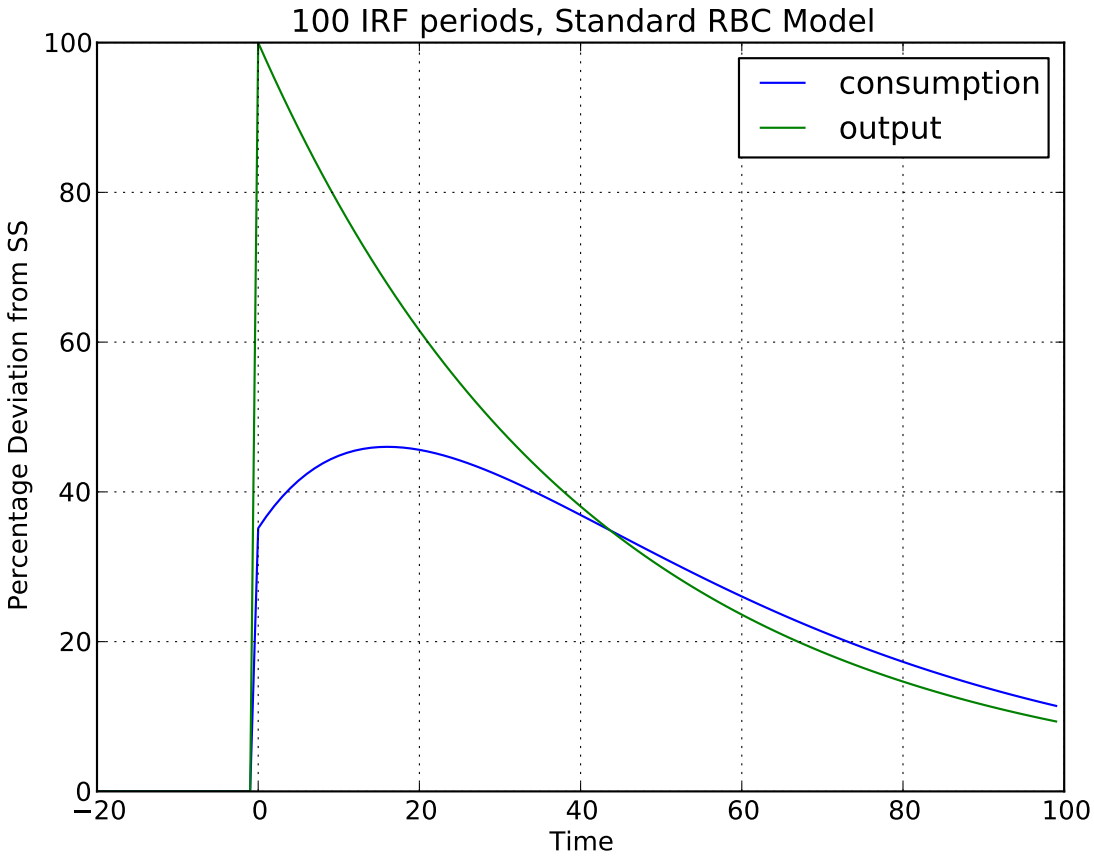

200 simulated periods, Standard RBC Model
consumption(cff)
output(cff)
Percentage deviation from SS
Time
10
5
0
−5
−10
−15
0
50
100
150
200

5.6.5 Generating impulse-response functions

Dynamic solutions obtained to first-order approximated DSGE models using the method of perturbations have a great deal in common with standard Vector Autoregression (VAR) models commonly used in applied Macroeconometrics. This in turn implies that solved DSGE models can be described using so-called impulse-response functions (also abbreviated as IRFs) or impulse-response graphs which show how the solved model responds to a one-off shock to a particular exogenous state variable. In PyMacLab this can easily be achieved as follows:

```
# Import the pymaclab module into its namespace, also import os module
In [1]: import pymaclab as pm
In [2]: from pymaclab.modfiles import models

# Also import matplotlib.pyplot for showing the graph
In [3]: from matplotlib import pyplot as plt

# Instantiate a new DSGE model instance like so
In [4]: rbc1 = pm.newMOD(models.rbc1_res)

# Now solve and simulate the model
In [5]: rbc1.modsolvers.forkleind.solve()
In [6]: rbc1.modsolvers.forkleind.irf(100,('productivity',))

# Plot the simulation and show it on screen
In [7]: rbc1.modsolvers.forkleind.show_irf(('output','consumption'))
In [8]: plt.show()
```

This produces the following nice graph. Notice that here the shock to total productivity has been normalized to 100%.


100 IRF periods, Standard RBC Model
consumption
output
Percentage Deviation from SS
Time
100
80
60
40
20
0
−20
0
20
40
60
80
100

CHAPTER

SIX

RELEASE HISTORY

6.1 0.90.1-dev (2012-9-12)

- Implemented the Christiano-Fitzgerald asymmetric band-pass filter to extract stationary cycle from simulated data.
- Updated documentation substantially.

6.2 0.89.1 (2012-9-11)

- Re-introduced sympycore as default CAS. It is MUCH faster than sympy. Now comes packaged with PyMacLab.
- Also decided to always package the latest (working and tested) parallel python(pp) library with PyMacLab.
- PyMacLab's parser now also replaces substituted items in the closed form steady state section.
- A new parameter can be passed to pymaclab.newMOD(mesg=True|False) to generate diagnosis print-outs during instantiation.
- Another new parameter is pymaclab.newMOD(ncpus='auto'|INT) where INT is some integer. Number of CPU cores to be used. Default=1.
- Another new parameter is pymaclab.newMOD(mk_hessian=True|False). Should expensive Hessian be computed? Default=True.
- PyMacLab now supports 5 different ways of finding a DSGE model's steady state.
- One of the 5 SS methods uses externally passed arguments pymaclab.newMOD(use_focs=[],ssidic={}) to use FOCS to solve for SS.
- One-off and queued dynamic run-time DSGE model updaters objects have been added
- Improved various DSGE model templates. Monika Merz's unemployment RBC DSGE model is now working.
- Further improvements and additions to the online documentation

6.3 0.88.1 (2012-9-3)

- PyMacLab now only depends on Sympy and no longer the outdated Sympycore.
- Significantly improved documentation, now on its way of being first draft.

- Translated some model files to make use of new functions.
- Deprecated alternative ways of specifying law of motion of exogenous states.
- Added new SS{EXPRESSION} steady-state conversion feature to Variable Substitution section.
- DIFF{EXPRESSION} now also works for steady state expressions w.r.t. steady state variables.

6.4 0.85 (2012-08-30)

- Added Sphinx Documentation with API Section.
- Analytical differentiation using DIFF{EXPRESSION,x(t)} now possible in Substitution Section of modfiles.
- Forward and Backward operators FF_X{EXPRESSION} and BB_X{EXPRESSION} for X=1...N now possible in Substitution section of modfiles.

6.5 0.8 (2012-08-21)

First public release.

CHAPTER

SEVEN

COPYRIGHT & LICENSE

PyMacLab is released under an Apache 2.0 license, and is (c) Eric M. Scheffel:

```
Copyright 2007-2012 Eric M. Scheffel

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

APPENDIX

A

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Olivier Jean Blanchard and Charles M Kahn. The solution of linear difference models under rational expectations. *Econometrica*, 48(5):1305–11, July 1980.

[2] Paul Gomme and Paul Klein. Second-order approximation of dynamic models without the use of tensors. *Journal of Economic Dynamics and Control*, 35(4):604–615, April 2011.

[3] Michel Juillard. Dynare: a program for the simulation of rational expectation models. Computing in Economics and Finance 2001 213, Society for Computational Economics, Apr 2001.

[4] Henry Kim, Jinill Kim, Ernst Schaumburg, and Christopher A. Sims. Calculating and using second order accurate solutions of discrete time dynamic equilibrium models. Discussion Papers Series, Department of Economics, Tufts University 0505, Department of Economics, Tufts University, 2005.

[5] Robert G King and Mark W Watson. System reduction and solution algorithms for singular linear difference systems under rational expectations. *Computational Economics*, 20(1-2):57–86, October 2002.

[6] Paul Klein. Using the generalized schur form to solve a multivariate linear rational expectations model. *Journal of Economic Dynamics and Control*, 24(10):1405–1423, September 2000.

[7] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[8] Charles L. Evans Martin Eichenbaum, Lawrence J. Christiano. Nominal rigidities and the dynamic effects of a shock to monetary policy. *Journal of Political Economy*, 113(1):1–45, February 2005.

[9] Stephanie Schmitt-Grohe and Martin Uribe. Solving dynamic general equilibrium models using a second-order approximation to the policy function. *Journal of Economic Dynamics and Control*, 28(4):755–775, January 2004.

[10] H. Uhlig. A toolkit for analyzing nonlinear dynamic stochastic models easily. Discussion Paper 97, Tilburg University, Center for Economic Research, 1995.

INDEX

C

cloud
    installation, 57

P

PyMacLab
    changelog, 53
    license, 56

T

tutorial
    DSGE instance
        dynamic updating, 29
        solution, 41