A Cook-Book of IRIS

For beginners

Jaromír Beneš Martin Fukač

October 4, 2008

Contents

List of Figures

Preface

This manuscript is prepared for the **DSGE modelling course** at the Reserve Bank of New Zealand, Wellington, October 6-10, 2008. The aim is to provide a cook-book for the first-time users of the IRIS Toolbox in Matlab, and its use for dynamic stochastic general equilibrium (DSGE) modelling. We presume basic Matlab literacy of the users. Being written for beginners, the manuscript deals only with standard modelling problems. For demonstration purposes we work with a simple *non-linear* but *stationary* DSGE model.

We appreciate your feedback on the manuscript, in particular, if you find it incomprehensive, find errors or omissions, or you seek more clarification. Any comments are welcome. Please approach the corresponding author Martin Fukač at `martin.fukac@rbnz.govt.nz`.

Chapter 1

Simulating stationary models

Creating a model structure in IRIS involves three basic steps:

1. Rewrite the model dynamic equilibrium in Iris.

2. Find the model steady state.

3. Solve the model.

The goal of this chapter is to take you through each individual step so that at the end of the chapter you are able to rewrite your own models in the syntax of IRIS and simulate them. We will show you a simple way how to perform an intervention analysis, and how to run Monte Carlo experiments.

In practice, you can deal with linear models, non-linear models, and their stationary or non-stationary variant. Iris has the capacity to deal with all these models. For demonstration purposes, it is sufficient to begin with a simple non-linear and stationary model. The non-linearity property is suitable for demonstrating how to solve for the steady state (the solution is trivial for linear models). And stationarity makes the solution simple.

1.1 Write a model in IRIS

1.1.1 The theoretical model

For demonstration purposes we will use the model from **?** adjusted for price indexation, and modified households' preferences. We also use an ad hoc policy rule. The dynamic (competitive) equilibrium of the model economy is characterised by the following set of equations:

Households:

$$\begin{aligned} MUC_t &= (1-\chi)/[C_t \exp(-\varepsilon_t^c) - \chi C_{t-1}], && (1.1)\\ MDL_t &= L_t^\eta, && (1.2)\\ MUC_t &= \beta(1+r_t)E_t(MUC_{t+1}), && (1.3)\\ \Delta w_t - \Delta w_{t-1} &= \beta E_t(\Delta w_{t+1} - \Delta w_t) \\ &\quad + \kappa_w \left[\log\left(\frac{MDL_t}{MUC_t}\right) - \log Q_t\right] + \varepsilon_t^w. && (1.4) \end{aligned}$$

Producers:

$$\begin{aligned} Y_t &= XK^\gamma L^{1-\gamma} && (1.5)\\ MPL_t &= (1-\gamma)\frac{Y}{L} && (1.6)\\ \Delta p_t - \Delta p_{t-1} &= \beta E_t(\Delta p_{t+1} - \Delta p_t) + \kappa_p(\log Q_t - \log MPL_t) + \varepsilon_t^p && (1.7)\\ \log X_t &= \rho_x \log X_{t-1} + \varepsilon_t^x && (1.8) \end{aligned}$$

Monetary policy:

$$r_t = \rho_r r_{t-1} + (1-\rho_r)(\bar{r} + \mu_p \Delta p_t + \mu_g g_t) + \varepsilon_t^r \qquad (1.9)$$

Market clearing condition:

$$Y_t = C_t \qquad (1.10)$$

Definitions:

$$g_t = \log \frac{MDL_t}{MUC_t} - \log MPL_t \tag{1.11}$$

$$Q_t = Q_{t-1} \frac{1 + \Delta w_t}{1 + \Delta p_t} \tag{1.12}$$

MUC_t is the marginal utility of consumption, MDL_t is the marginal disutility of labour, MPL_t is the marginal product of labour, C_t is consumption, L_t is labour, Δw_t is wage inflation, Δp_t is price inflation, r_t is the interest rate, Q_t is the effective output, Y_t is the output, X_t is the level of technology. ε_t^c, ε_t^w, ε_t^p, ε_t^r is the preference shock, wage setting shock, price setting shock, and monetary policy shock, respectively.

In Chapter 3, the model is estimated for 4 observable variables: output growth `dy_`, Q2Q price inflation `dp_`, Q2Q wage inflation `dw_` and policy rate `r_`. We use the US data ranging from 1990:Iq to 2008:IIq.

1.1.2 Iris model syntax

Writing a *model file* is the first step in obtaining a *model structure*. Throughout the manuscript, we will call a model that is uploaded in the Matlab's workspace the *model structure*. Writing a model in to IRIS format takes creating an ASCII file. Such a file must contain the following model characteristics:

1. **`!variables:transition`**
2. **`!parameters`**
3. **`!variables:residual`**
4. **`!equations:transition`**
5. **`!variables:measurement`**
6. **`!equations:measurement`**

Among optional characteristics belong:

1. **`!variables:log`**
2. **`!substitutions`**

We will employ `!substitutions` for `rbar` to demonstrate its use. It will shorten the expression for the steady-state real interest rate $1/\beta - 1$, which repeats in the model code. The use of `!substitutions` makes the model code shorter and more easily readable. Instead of typing `1/beta-1` many times, you use `$rbar$`. `$...$` characters tell to Iris that the particular expression defined in `!substitutions` must be substituted in when solving the model.

A model file for the Erceg *et al.* (EHL, 2000) from section 1.1.1 is described below.

Open a new file in the m-editor, and save it as `EHL2000.model` (`model` is the file extension). Then rewrite the model (1.1)-(1.12) following the structure outlined above. When rewriting the actual model, the time subscripts (forward, and backward looking ones) come in curly braces {}.[1] You do not (need to) declare contemporaneous time subscript.

[1]In Iris, there is no limit on the order of backward or forward lookiness of your model.

```
!variables:transition
'Marginal utility of consumption' MUC
'Marginal disutility of labour' MDL
'Consumption' C, 'Real wage rate' Q, 'Nominal wage rate, Q/Q' dw
'Marginal product of labour' MPL
'Output' Y, 'Labour' L, 'Final goods price, Q/Q' dp
'Productivity' X, 'Nominal rate' r, 'Inefficiency gap' g

!variables:log
!allbut dw, dp, r, g

!parameters
% Steady-state parameters
beta, eta, K = 1, gamma

% Transitory parameters
chi, kappaw, kappap, rhox

% Policy parameters
rhor, mup, mug

!variables:residual
'Consumption preference shock' ec , 'Wage setting shock' ew
'Price setting shock' ep, 'Produtivity shock' ex
'Policy shock' er

!substitutions
rbar = (1/beta - 1);

!equations:transition
% Households
MUC = (1-chi)/(C*exp(-ec) - chi*C{-1});
MDL = Lêta;
dw - dw{-1} = beta*(dw{1} - dw) + kappaw*[log(MDL/MUC) - log(Q)] + ew;

% Producers
Y = X*Kĝamma*L(̂1-gamma);
MPL = (1-gamma)*Y/L;
dp - dp{-1} = beta*(dp{1} - dp) + kappap*[log(Q) - log(MPL)] + ep;
log(X) = rhox*log(X{-1}) + ex;

% Monetary policy
r = rhor*r{-1} + (1-rhor)*[$rbar$ + mup*dp + mug*g] + er;
```

```
    % Market clearing
    Y = C;

    % Definitions
    g = log(MDL/MUC) - log(MPL);
    Q = Q{-1}*(1 + dw)/(1 + dp);

    !variables:measurement
    'Output, Q/Q' dy_, 'Final goods prices, Q/Q' dp_
    'Nominal wage rate, Q/Q' dw_, 'Nominal rate' r_

    !variables:log
    !allbut dy_, dp_, dw_, r_

    !equations:measurement
    dy_ = Y/Y{-1} - 1;
    dp_ = dp;
    dw_ = dw;
    r_ = r - $rbar$;
```

When the *model file* `EHL2000.model` is completed and saved, the next step is uploading the model in to Matlab. We will use the Iris command `model()` that does the trick. The model is uploaded in the Matlab's memory as an active *model structure*, which you can already work with (e.g., checking basic properties). Before we proceed, it is useful to open a new m-file in which you code the sequence of commands that upload, solve and simulate the model. You can save is as `read_model.m`.

```
m = model('EHL2000.model');
```

You can check that the model was uploaded correctly by running the `read_model.m` file and typing `m` in to the Matlab command window. It prints out the following message:

```
non-linear model object: 1 parameterisation(s) solution not
available for any parameterisation
```

The `solution not available` means that the model is in its structural form, and needs to be *solved* – linearised and solved for rational expectations.

When uploading the model in to Matlab's memory, you should count with problems. It is very common that you make an error or typo when rewriting the model in to the model file. Usually, you make typos in variable names, forget declare some parameters, etc., which will be revealed when uploading the model. In most cases, Iris is able to name and locate the error. The only thing you have to do is to go back to your model file and correct it. For large models, or complicated models the problems may be such that Iris cannot name or locate them exactly. Then it depends on the modeler's experience to fix the problem.[2]

Creating the `EHL2000.model` file and uploading the model in to the Matlab's working memory completes the first step of creating a workable *model structure*. The second step is to solve the model.

1.2 Solve model

We recall that we deal with a stationary model in this chapter, which significantly simplifies the model solution. In general, solving a model in Iris consists of three steps: (i) linearising the model around its steady state, (ii) solving for forward-looking variables, and (iii) creating a state-space representation (`idoc sspace()`).

Because the methods we are going to use is numerical, we begin by assessing initial values for the model parameters. In the `read_model.m`, create a structure `P`, which will store the initial values

[2]Please see the IRIS web page, which offers an on-line support.

```
    P          =  struct{};

    % Steady-state parameters
    P.beta     =  0.99^(1/4);
    P.eta      =  1;
    P.gamma    =  0.40;

    % Transitory parameters.
    P.chi      =  0.80;
    P.kappaw   =  0.05;
    P.kappap   =  0.10;
    P.rhox     =  0.80;

    % Policy parameters.
    P.rhor     =  0.80;
    P.mup      =  1.50;
    P.mug      =  0.50;
```

To check that the structure is correctly created, run `read_model.m`, and type `P` in to the command window to print the parametrisation. If everything is correct, the next step is to assign the parameters to the model structure `m`. It is done by the Iris command `assign()`.

```
m = assign(m,P);
```

Again, you can double check the assigned parametrisation by typing in the command window `get(m,'parameters')`, which is another Iris command that prints the set of parameters associated with the model. We will talk about the use of `get()` a little later.

The `assign()` command works for a set of parameters, but can also be used for subsets or individual parameters. For example, you can assign structural parameters and shock standard errors in two separate sets, e.g. `P1` and `P2`. If you are up to changing a single parameter, Iris offers a fast way of changing it directly in the uploaded model structure. The syntax is `m.`*parameter_name* `=` *new_value*. For example, if you want to change the the value of policy parameter `mup` from 1.50 to 0, you simply type: `m.mup =`

`0`. But you have to remember that if the change is made in the command window and not in the `read_model.m` file, it is only temporary, and when you run the file again, the parameter will return to its preset value in the file. But you will find the possibility of the fast change in parametrisation useful when you experiment with the model (testing its sensitivity to shocks, etc). When changing parameter values, you have to know the nature of the changed parameter. If it is a parameter that determines the steady state, after changing it, you have to remember to re-solve (re-evaluate) the steady state (`sstate()`) and the model (`solve()`) again. If it is a transitory parameter, i.e., it only affects model dynamics but not the steady state, after changing it you only have to re-solve the model (`solve()`), and do not need to re-solve (re-evaluate) the steady state.

Having the initial parameter values, we can finally solve for the steady state. We use the command `sstate()`, which uses numerical procedures to get the steady state: `sstate()`.

```
m = sstate(m,'fix',{'X','dp','dw','g','r','dy_','dp_','dw_'});
```

To help the numerical algorithm, it is always good to fix the steady-state values that we already know. Here it is for variables `X, dp, dw, g, r, dy_, dp_, dw_`.

The final step is solving the model using the Iris command `solve()`.

```
m = solve(m);
```

Iris will give you the model structure `m`, which is ready to be worked with: simulated, estimated, etc.

A simple check of the solution property is by typing `m` in the command window, which prints out

```
    non-linear model object: 1 parameterisation(s)
    solution(s) available for a total of 1 parameterisation(s)
```

If the model solution is stable, it is available, and you will get the message `solution(s) available`. If the solution is not available, it means that the model could not be solved.[3]

Now it is a good opportunity to talk about the command `get()`. It is one of the most frequently used Iris commands. It gives you information about the model properties, e.g., list of parameters, steady state values, list of stationary and unit-root variables, initial conditions, etc. The command has many options and for their list we refer to Iris help (`idoc model/get`).

The command `get()` helps you to extract useful information about the model structure. If you type `idoc model/get` in your Matlab command window, you receive a list of options of what information you can get. The most commonly used are:

`get(m,'parameters')` List values of model parameters.

`get(m,'xlist')` List of transition (state) variables.

`get(m,'ylist')` List of observable variables.

`get(m,'SState')` Steady-state values.

`get(m,'Omega')` Variance-covariance matrix of model shocks.

`get(m,'StationaryList')` List of stationary (transitory and observable) variables

`get(m,'NonstationaryList')` List of non-stationary variables.

Because we are also going to work with the model structure created here in the following chapters, it is useful to save it as a mat-file:

```
savestruct('EHL2000.mat',m);
```

It is an Iris structure, and it can be uplodaded (`loadstruct()`)at any other occasion and the model structure is ready to be worked with. It will save you steps of uploading the model file, parametrising, and solving the model again.

[3]The model structure in Iris is very flexible. Advanced users, may use the possibility to assign multiple parametrisations of the model, which can be useful in evaluating solution stability regions, etc.

1.3 Simulate model

Having declared the model dynamic equilibrium, and solving it, we are now ready to work with the model. In this section, we show you how to run an intervention analysis, and how to simulate artificial time series of the model variables. Because the model structure is kept in the Matlab's work memory, each model operation is simple and quick to perform.[4]

1.3.1 Impulse responses

The example below produces an impulse response to a unitary monetary policy shock in the EHL(2000) model:

```
startdate  =  qq(1990,1);
enddate    =  qq(2008,2);
d          =  zerodb(m,startdate-1:enddate);
d.er(startdate+12) = 1;
s          =  simulate(m,d,startdate:enddate,...
              'deviation',true,'anticipate',false);
```

The first two lines define the time horizon for simulation: `startdate` and `enddate`. The third line of code, uses the command `zerodb` to create a zero database related to the model. It contains all model's `m` variables, including shocks, defined as time series on the time span `startdate-1:enddate`. The time span is expanded for the period `startdate-1`, which caries the information on initial conditions. If the reduced system is of the second order, then the zero database must count with two initial conditions, i.e. we create the database for periods `startdate-2:enddate`. Note that the time format is specific to IRIS (use of `qq()` or `mm()`), and significantly simplifies handling of time series. We talk about it in more detail in section 3.1. If you wish, you can define the time span simply by the number of observations, that is

```
startdate  =  2; % Counting w/t init.  condition
enddate    =  40;
```

[4]Presenting the results will take you more coding than actually simulating the model.

The third line of the code above sets the value of the monetary policy shock `er` to 1 at the time period `startdate+12`. It is zero otherwise. You can double check by plotting it

```
plot(d.er);
```

The reason for shocking the system at the period `startdate+12` is to demonstrate the effect of *anticipated* and *unanticipated* shocks.

The last line of the code above simulates the model. `s` is an output database structure, similar to `d`. It contains the impulse responses of the system. The use of `'deviation', true` states the option that the impulse response is produced as a deviation from the steady state. It is set to `true` if the model variables and observables are deviations from the steady state. Then there is another option `anticipate`, which relates to how exogenous shocks will be treated in the simulation (and perceived by rationally behaving economic agents in the model economy). If the option is set to `false`, the shock is a standard exogenous shock, which causes surprise. If you use `'anticipate', true` instead, the shock is anticipated and agents react to it well before it actually takes place (at `startdate+12`). In Figure 1.3.1 you can see that that the response is quite different. We put the economics behind aside.

You can experiment with anticipate/unanticipate options, and plot individual variable's responses by

```
plot(s.variable_name);
```

1.3.2 Monte Carlo experiments

You can also run a Monte Carlo experiment, in which you produce random draws for all shocks, and simulate the model. The Iris command `resample()` does all that for you. Here is an example:

```
 N  =  2;
 y  =  resample(m,[],startdate:enddate,N,'output','dbase');
```

Figure 1.1: Impulse responses to unanticipated and anticipated monetary policy shocks (EHL, 2000, model)

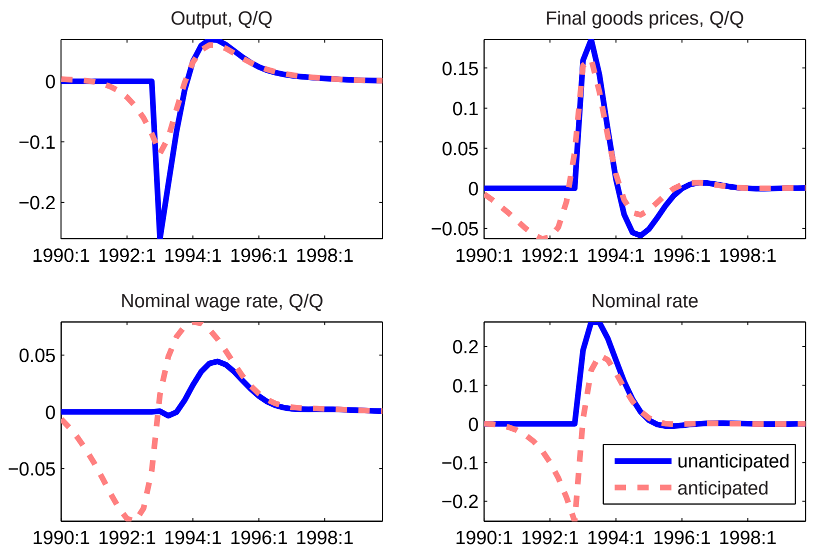
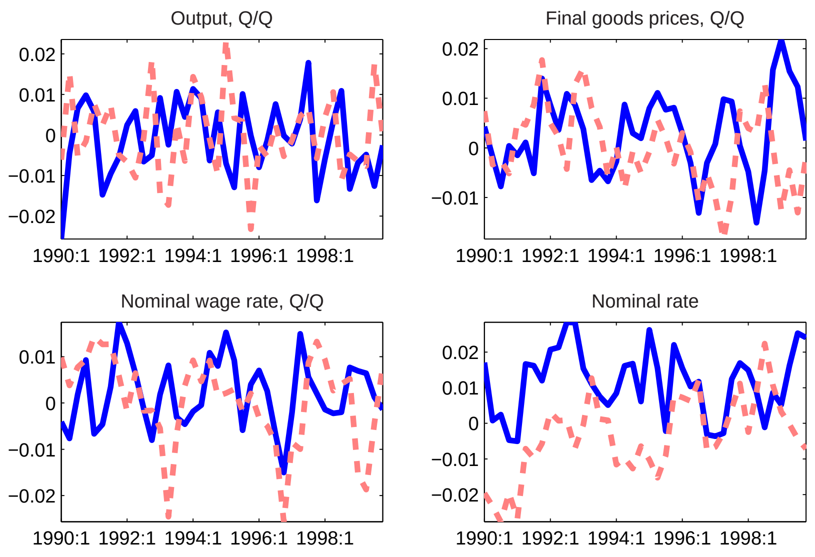


Figure 1.2: Monte Carlo simulation of EHL's(2000) model (N=2)



You have the option to simulate one or more realisations of model series. You just change the number of draws `N`. An example of two MC experiments is plotted in Figure 1.3.1.

The the command `resample()` can be used for Bootstrap experiments as well. You have to have the estimates of model shocks available (either from model estimation, or likelihood evaluation for a particular parametrisation). We keep the use of Bootstrap for section 4.1, where we analyse the data matching properties of the EHL(2000) model.

1.4 Matlab code

1.4.1 Read model

```
%% Get the model into Matlab.

%% Clear memory of everything.
clear all;

%% Read the model.
% Read the model code into Matlab as an IRIS model object.
% Set all std deviations to 0.01.
m = model('EHL2000.model','std',0.01);
disp(m);

%% Assign parameters.
% Create a parameter database, and assign this database
% to the model object.
% Assign also some of the known steady-state values.
% These are then fixed and taken as granted when we call
% the sstate function.

P = struct();

% Steady-state parameters.
P.beta = 0.99^(1/4);
P.eta = 1;
P.gamma = 0.40;

% Transitory parameters.
P.chi = 0.80;
P.kappaw = 0.05;
P.kappap = 0.10;
P.rhox = 0.80;

% Policy parameters.
P.rhor = 0.80;
P.mup = 1.5;
P.mug = 0.5;

P.X = 1;
P.dp = 0;
P.dw = 0;
P.g = 0;
P.r = 1/P.beta - 1;

P.dy_ = 0;
P.dp_ = 0;
P.dw_ = 0;
```

```
P.r_ = 0;

m = assign(m,P); disp(get(m,'parameters'));

%% Find steady state.
% Use Optimisation Toolbox to solve the model for steady state.
% Fix some of the known steady-state values.
% Note that there are other, more sophisticated options in IRIS
% how to handle the problem of finding steady state in more complex models.
m = sstate(m,'fix',{'X','dp','dw','g','r','dy_','dp_','dw_'});

% Display steady-state database.
S = get(m,'sstatelevel'); disp(S);

% Make sure all equations hold with the currently computed steady state.
disp(' '); [flag,discrep,list] = chksstate(m); if flag
   disp('Steady state OK.');
else
   error('Steady state invalid: "%s".\n',list{:});
end

%% Solve the model.
% Linearise the model around its steady state
% and solve the model by generlised Schur decomposition.
[m,npath] = solve(m);

if npath == 1
   disp('1st-order accurate solution OK.');
end

disp(' '); disp(m);

%% Save the model.
% Save the solved model object.
% SAVESTRUCT more efficient than SAVE.
savestruct('EHL2000.mat',m);
```

1.4.2 Simulate model

```
%% Read the model.
% Read the model as an IRIS model object.
m = loadstruct('..\Model\EHL2000.mat');
disp(m);

%% Ask questions about model
% Use the IRIS function get() to see the model's characteristics
isstationary(m)
get(m,'sstate')

list  = get(m,'ylist');
names = get(m,'yComments');

%% Simulate impulse responses
% Define simulation range, create a zero database, shock and simulate the
% system.
startdate = qq(1990,1);
enddate = qq(1999,4);
d=zerodb(m,startdate-1:enddate);

d.ex(startdate+12) = -1;

s1 =
simulate(m,d,startdate:enddate,'deviation',true,'anticipate',false);
s2 =
simulate(m,d,startdate:enddate,'deviation',true,'anticipate',true);

figure(); for i = 1:length(list);
    subplot(2,2,i)
    plot([s1.(list{i}), s2.(list{i})]);
    title(names(i),'interpreter','none');
    axis tight
end hold on legend('unanticipated','anticipated')


%% Monte Carlo experiments
% Simulate time series out of the model. The number of experiments controls
% for the number of realisations that you get for an individual variable.
N = 2; % number of experiments
y = resample(m,[],startdate:enddate,N,'output','dbase');

figure(); for i = 1:length(list);
    subplot(2,2,i)
    plot(y.(list{i}));
    title(names(i),'interpreter','none');
    axis tight
end
```

Chapter 2

Solving stationary models for steady state

As the problem of steady state requires a special attention, we were postponing the topic until now. In this chapter we discuss the principles of finding a steady state solution to structural models using Iris. We will discuss how the solution differs for stationary models: linear and non-linear.

There are two ways how to calculate the steady state for a stationary model. If you have available the Symbolic toolbox, you may want to try to find an analytical solution. We say "try", because for some models the analytical solution may not exist. The advantage of having the analytical solution is that it significantly speeds up computations. The second possibility is to solve for the steady state numerically by using the Optimisation toolbox. There is also a third approach – solve for the steady state in hand – but we do not cover it here.

2.1 Analytical solution

The first step is to adjust the original model in to the form, which will be used to solve for the steady state. It is basically a strip-off version of the original model structure with eliminated time subscripts, expectations terms, and stochastic parts.

Open a new file and save it as `Ireland.sstate`, with `sstate` as the file extension. The structure of such a file contains declaration of `!parameters`, and `!equations`. If `!equations` is followed by `!symbolic`, it declares that the following block of equation will be solved analytically. If this command is skipped, IRIS will solve the equation numerically. Each equation block is completed by `!solvefor`, which declares what variables it will be solved for from the system. Of course, the number of endogenous variables must be equal to the number of equations.

In some complicated cases, it may be that an explicit solution does not exist. In such a case we try to separate such steady state, and evaluate it numerically. A combination of analytical and numerical solution is possible in IRIS. A system of equations that determine the steady state can be divided in to subsets, and each subset can be solved individually. There is one condition that applies in such a case. The subsystems has to be ordered hierarchically. That is the first subsystem has to be solved independently from the subsystem that follows.

This is an example how such a steady state file looks like.

```
!parameters

!equations

!symbolic

!equations:transition

!solvefor
```

In the above example we divided the original model in to two subsystems: transition equations, and measurement equations. This is only for demonstration purposes. The steady state can be solved for the whole system.

The next step is the analytical solution itself:

```
sstatefile('Ireland.sstate','Ireland_sstate.m');
```

The IRIS command `sstatefile` will do the trick. Using the command, first, you have to declare the name of the file that describes the steady state to be solved for. Second, you declare the name of an m-file that this command will automatically create, and in which the solution will be stored. In our example, we solve `Ireland.sstate`, and the result will be stored in `Ireland_sstate.m`.

`Ireland_sstate.m` is a function type of file. You can open it and view its structure. It evaluates the model's steady state for a given set of parameters. You have to supply them. For example, we have the following set of initial parameters. We create a structure `P`, and then declare the parameters

```
P          =  struct{};

% Steady-state parameters
P.beta     =  0.99^(1/4);
P.eta      =  1;
P.gamma    =  0.40;

% Transitory parameters.
P.chi      =  0.80;
P.kappaw   =  0.05;
P.kappap   =  0.10;
P.rhox     =  0.80;

% Policy parameters.
P.rhor     =  0.80;
P.mup      =  1.50;
P.mug      =  0.50;
```

Now we can evaluate the steady state. We take the set of initial parameters, and evaluate the `Ireland_sstate.m` for them.

```
P = Ireland_sstate(P);
```

As the output, we get the original structure P expanded for the steady-state values. If you print P in your Matlab command window, you will see that now the structure P also contains the endogenous variables, e.g. `P.y = 0`, `P.infl = 0`, etc. The values associated with endogenous variable is the appropriate steady state.

The third, and last step is to assign the parametrisation and steady state to the original dynamic model, and solve the model. The solution takes linearisation, and solution for rational expectations. The resulting workhorse model is stored as a state-space system.

```
m = assign(m,P);
m = solve(m);
```

The `assign` command assigns the parametrisation `P` to the model structure `m`. The `solve()` command then solves the model. If you now type `m` in your Matlab command window, it will print the properties of the model structure

```
nonlinear model object: 1 parameterisation(s)
solution(s) available for a total of 1 parameterisation(s)
```

2.2 Numerical solution

An alternative approach to obtain a model steady state is a numerical solution. This solution is always available, and users that do not have the Symbolic toolbox may always use it. To obtain a numerical solution takes two steps. First we have to parametrise the model. For our example, we can use the same parameters as before. You have to create the structure P again. Then the parameter structure `P` has to be assign to the model `m`

```
m = assign(m,P);
```

To check that the parameters have been assigned correctly, type `get(m,'parameters')`. `get()` is another useful IRIS command. The parameter `'parameters'` tells IRIS to print out model parameters. `get()` is one of the most frequently used commands in IRIS. It has many more useful options see the help to the command: `idoc model/get()`.

The last step is to solve the model. The command for a numerical solution is simply

```
m = sstate(m);
```

The output is a model `m` structure, which is already linearised and solved for rational expectations, and it is ready to be used for simulation, estimation, etc.

For large models the numerical solution might be the only option. However, it might not be straightforward to use. For really large and non-linear models, to help the numerical algorithm to converge, you might have to use a similar approach as in the analytical solution case. You will have to create a special sstate-file, in which you divide the original model in to a number of subsystems, which will be sequentially solved for the steady state. Recalling that a numerical solution is obtained if you do not declare `!symbolic` in the `Ireland.sstate` file.

Chapter 3

Estimation

The goal of this chapter is to show how you estimate a structural model in IRIS. IRIS offers three basic methods in its toolbox: maximum likelihood, regularised maximum likelihood, and posterior mode maximisation. In this chapter, you learn how to apply these three methods to stationary models. All three methods are performed by the Iris command `estimate()`, and choosing an appropriate option. In estimating your model, you will typically follow this pattern:

1. Load model structure;
2. Load database structure;
3. Define time span for estimation;
4. Choose parameters to estimate;
5. Run the estimation.

By default, the command `estimate()` uses the Optimisation toolbox in Matlab. But Iris is flexible enough to maximise objective functions by other methods as well, e.g., you can use csminwel. You only have to declare it as one of the parameters in `estimate()`, see `idoc model/estimate`. But we leave this option for later as an advanced problem for coding.

3.1 Iris data format

Iris uses its own data format. Before estimation or data manipulation using any Iris commands, the data that you use must be transformed in to the time series object. But do not worry. The process is very straightforward and quick. First, you define the data span. Second, you upload raw data series (either from Excel, ASCII file, or any other external database). Third, you use the Iris command `tseries()`, which associates the raw data with the defined time subscripts, and creates the time series object. The advantage of such a structure is that it allows for an easy data manipulation: extracting sub-samples, changing frequencies, plotting, etc. When plotting a time series object, Matlab automatically treats the horizontal axis as the time domain.

An example below creates a quarterly time series from randomly drawn numbers. The sample code follows the three steps outlined above. The Iris command `qq(YYYY,Q)` is used to create the quarterly data range.[1]

```
startdate = qq(1990,1);
enddate   = qq(1990,4);

range = stardate:enddate;
y = rand(size(range));

Y = tseries(range,y);
```

If you print the time series `Y` in the command window, it will be a 4-by-1 time series object, where the random numbers are linked to the time domain.

```
 1990Q1:  0.24065
 1990Q2:  0.00885
 1990Q3:  0.67159
 1990Q4:  0.90481
```

The data manipulation is then made very easy. For example, if you wish to create a sub-sample from observations in the 2nd and 3rd quarter of 1990, you simply specify the time range:

```
Y1 = Y{qq(1990,2):qq(1990,3)};
```

[1]Similarly, a monthly range can be created: `mm(YYYY,MM)`.

Note that the range (index) of the sub-sample must be in curly braces {}. If you use parentheses (), then Iris will return data as a vector of numbers (without time subscript). If you estimate your model on a data sub-sample, a quite common mistake is the use parenthesis () instead of curly braces {}. Sometimes it happens that you will not notice that you made such a mistake, until you use the data to evaluate a likelihood function. For example if you use such a sub-sample in `estimate()`, the command will not function properly, and you get back the initial values associated with zero likelihood.[2] It may take some time before you find out that there is a problem in the time series object.

Similarly, if you have long time series, the Iris time format allows you to easily locate particular observations. E.g. in a sample of inflation `INFL` from 1990 to 2008, if you wish to know what the inflation rate was in 1995:IIIq, you simply type `INFL{qq(1995,3)}`.

3.2 Maximum likelihood

In this section, we estimate EHL(2000) model by the method of maximum likelihood (MLE). Again, we suggest to open a new m-file where you can code the sequence of commands that estimate the model, and save it in your Matlab workspace.

Following the step given in the chapter's introduction, we first upload the model structure created in Chapter 1: `EHL2000.mat`. We use the Iris command `loadstruct()`:

```
m = loadstruct('EHL2000.mat');
```

`m` is the model structure. You can use the command `get()` and check its properties (see `idoc model/get`).

The second step is to upload a database, and define the data time range. From the model file created in Chapter 1 `EHL2000.model`, we know that the model has three observables: output growth `dy_`, price inflation rate `dp_`, wage

[2]In recent Iris versions this problem might be fixed.

inflation rate `dw_`, and short term interest rate `r_`. We use the US data.[3] We skip the step of creating a database, the principles are described in section 3.1 above, and upload a database that is pre-prepared and saved in `US_data.mat`. We use the same command to upload a database structure as for uploading a model structure:

```
d = loadstruct('EHL2000.mat');
```

The third step is to define the time span for the data. For our example, the data range is from 1990:Iq to 2008:IIq:

```
d = loadstruct('US_data.mat');

startdate = qq(1990,1);
enddate   = qq(2008,2);
```

When the database is uploaded, it is always useful to check the properties of your data, in particular, if they are ready to be used in estimation. You should check if the data have correct units, if they are seasonally adjusted, and if they are really in the Iris time series format. If the data are not in the time series format, Iris estimation procedures will not function properly.

The fourth step is to declare parameters that we want to estimate. For demonstration purposes, we estimate the whole set of model parameters, except of the time preference parameter β. In practice, you can estimate only a subset of them, and the rest of parameters can be calibrated. If you wish to keep your parameters fixed at preferred values, simply do not declare them for estimation. Declaring a parameter for estimation consists of (i) setting up an initial parameter value (this is necessary to launch the Kalman filter; if `NaN` is supplied, then IRIS takes the parameter value assigned in the model structure `m` as the initial value - typically the one used to solve for the steady state), and setting a (ii) lower and (iii) upper bound for the parameter:

[3]Source: Fed. The tick for original time series is: TB3MS (3-month Treasury bill: Secondary market) for `r_`, PCEC96 (real personal consumption expenditures) for the construction of `dy_`, PCE (nominal personal consumption expenditures) for the construction of final goods price deflator, AHETPI (Average hourly earnings: Total Private Industries) for construction of `dw_`. The original data frequency is monthly, which we transform to quarterly.

```
E.parameter = [ init_value, Lower_Bound, Upper_bound ];
```

This is the parameter set that we estimate for the EHL(2000) model on the US data:

```
    E         =  struct();
    E.chi     =  [0.80,0.50,0.95];
    E.kappaw  =  [0.05,0.01,1];
    E.kappap  =  [0.10,0.01,1];
    E.rhox    =  [0.80,0.01,0.95];
    E.rhor    =  [0.90,0.01,0.99];
    E.mup     =  [1.50,1.10,5];
    E.mug     =  [0.50,0.01,5];
```

The information on parameters is stored in the structure `E`. Iris uses the Kalman filter to evaluate the likelihood. Because we are using routines form Matlab Optimisation toolbox, it is useful not to set lower and upper bounds on the exact boundaries of the parameter space. It is because of the nature of most of the optimisation procedures that, for some reason, evaluate the boundaries first, and which may lead to numerical problems. A typical example is the estimation of the parameter on the inflation rate in the Taylor rule. If you allow for unitary values, the optimisation procedure may crash because if it hits 1, the Taylor principle will be violated, the model becomes unstable and the optimisation procedure may not converge. For exactly this reason, you see the lower bound for `mup` set to 1.10. For similar reasons, the lower and upper bounds for the AR(1) parameters are set to something like 0.10 and 0.95, respectively.

You also declare the standard errors for estimation. For estimating the standard errors, IRIS offers two options: (i) take one standard error as a numereaur, and estimate the rest in relative terms, (ii) estimate all standard errors in absolute terms. The choice between the two options depends on user's preferences, and likelihood fuzziness. Estimating the standard errors in the relative terms might be computationally convenient, and in some cases speeds up optimisation. We use this approach in our example. The naming convention for standard errors estimation in Iris is `E.std`*`_name_of_shock`*.

Otherwise the standard errors are declared for estimation in exactly the same way as any other parameter.

```
    E.std_ec  =  [0.01,0.001,0.20];
    E.std_ew  =  [0.01,0.001,0.20];
    E.std_ep  =  [0.01,0.001,0.20];
    E.std_ex  =  [0.001,0.0001,0.02];
    E.std_er  =  [0.01,0.001,0.20];
```

Here we are estimating the standard errors for five shocks: `ec, ew, ep, ex, er`.

The fifth and last step is the estimation step. The maximum likelihood estimation is the default option in the Iris command `estimate()`. You run

```
[P,mloglik,Grad,Hess,m,se2,F,pe,A,Pa,setup,pred,smooth] =
  estimate(m,d,startdate:enddate,E,'relative',true,'deviation',true);
```

As inputs, you have to supply the model `m`, database `d`, data range `range`, set of parameters to be estimated `E`. Further you have to specify that the standard errors will be estimated in relative terms of each other `'relative',true`. If the model and data are defined in deviations from steady state, you also have to declare it by setting the option `'deviation',true`. As an output you receive a database of estimated parameters `P`, the value of the likelihood function at its maximum `mloglik`. You also obtain `Gradient` and `Hessian` at optimum, an updated model structure for newly estimated parameters `m`,[4] estimated variance factor `se2`, and predicted `pred` and smoothed `smooth` variables. The variance factor is estimated if you use the option `'relative', true`. If you multiply the standard errors estimated and stored in `P`, you get the absolute values. We recall that, in the model structure `m`, which is the output of `estimate`, the standard errors are already re-scaled, and are ready to be used.

[4]Estimated standard errors in such an updated model structure are already in absolute terms. That is if the standard errors are estimated in relative terms, in the output model structure they are multiplied by the variance factor `se2` and thus re-scaled in to their correct magnitudes.

3.3 Regularised maximum likelihood

The second estimation method that Iris offers is the regularised maximum likelihood (RMLE).[5] If a model contains parameters that lack identification, the likelihood function is flat in their dimension. A possible sign of such a problem is a non-converging algorithm searching maximum. But the lack of identification can be solved by regularising the likelihood function by a quadratic penalty function.[6] The penalty function helps to shape up the likelihood in the singular directions. It is equivalent to imposing normal priors on all initial parameter values. If the information on a particular parameter is insignificant, its estimate is pushed towards the initial value (or towards zero; depending on user's preferences).

You run the regularised likelihood in IRIS by declaring the option `'penalty'` in `estimate()`. `'penalty'` is followed by a numeric value, which multiplies the parameters specific penalty function. Because of the different magnitudes of parameters, the penalty has to be normalised (made parameter specific). For example, the penalty can be normalised so that if t-ratio for any estimate will not be greater than 2, it will be pulled towards its initial value. Technically it means, that the likelihood that is nearly flat will be shaped up by a normal prior centered around the parameter's initial value, with variance such that the t-ratio implied by the prior is 2. To run the RMLE you have to supply the parameter specific penalty as the fourth parameter when declaring the parameters for estimation, and then in the function `estimate()`, you use option `'penalty',1`.

For the EHL(2000) model, the information on estimated parameters is below. We define a variable `t_ratio`, which will be used to control the curvature of the prior. In our exercise we use such a prior curvature that gives t-ratio of 2. Note the penalty expression as the fourth parameter in `E`.

[5]We refer to Ljung (1999) for details on this method.
[6]It is similar idea as the ridge regression.

```
t_ratio   =  2

E         =  struct();
E.chi     =  [0.80,0.50,0.95,(t_ratio/0.80)^2/2];
E.kappaw  =  [0.05,0.01,1 ,(t_ratio/0.05)^2/2];
E.kappap  =  [0.10,0.01,1 ,(t_ratio/0.10)^2/2];
E.rhox    =  [0.80,0.01,0.95,(t_ratio/0.80)^2/2];
E.rhor    =  [0.90,0.01,0.99,(t_ratio/0.90)^2/2];
E.mup     =  [1.50,1.10,5 ,(t_ratio/1.50)^2/2];
E.mug     =  [0.50,0.01,5 ,(t_ratio/0.50)^2/2];

E.std_ec  =  [0.01,0.001,0.20 ,(t_ratio/0.01)^2/2];
E.std_ew  =  [0.01,0.001,0.20 ,(t_ratio/0.01)^2/2];
E.std_ep  =  [0.01,0.001,0.20 ,(t_ratio/0.01)^2/2];
E.std_ex  =  [0.001,0.0001,0.02,(t_ratio/0.01)^2/2];
E.std_er  =  [0.01,0.001,0.20 ,(t_ratio/0.01)^2/2];
```

Now you use the option `'penalty'` in the `estimate` command:

```
[P,mloglik,Grad,Hess,m,se2,F,pe,A,Pa,setup,pred,smooth] =
   estimate(m,d,startdate:enddate,E,'relative',true,...
   'deviation',true,'penalty',2);
```

The second way how to use the regularised likelihood is to apply it only on specific parameters. The likelihood function can be shaped only over the parameters that are expected to be weakly identified or they are not identified at all. Here is an example:

```
t_ratio    =  2

E          =  struct();
E.chi      =  [0.80,0.50,0.95,0];
E.kappaw   =  [0.05,0.01,1 ,0];
E.kappap   =  [0.10,0.01,1 ,0];
E.rhox     =  [0.80,0.01,0.95,0];
E.rhor     =  [0.90,0.01,0.99,0];
E.mup      =  [1.50,1.10,5 ,(t_ratio/1.50)^2/2];
E.mug      =  [0.50,0.01,5 ,0];

E.std_ec   =  [0.01,0.001,0.20 ,0];
E.std_ew   =  [0.01,0.001,0.20 ,0];
E.std_ep   =  [0.01,0.001,0.20 ,0];
E.std_ex   =  [0.001,0.0001,0.02,0];
E.std_er   =  [0.01,0.001,0.20 ,0];
```

The parameter `mup` is estimated using RMLE, the remaining parameters are estimated by the (constrained) MLE.

3.4 Prediction error minimisation

Except of likelihoodfunction, `estimate` allows for optimisation of different objective functions. The user has the option to supply her or his own preferred criteria. This section is devoted to (one-step-ahead) prediction error minimisation (PEM), which is built in Iris.

Asymptotically, the prediction error minimisation is equivalent to maximum likelihood, but in short samples this objective can provide different parameter estimates (in particular starndard errors), and, of course, different forecasting performance. If you are after tuning the model for forecasting, this might be the criteria to look at. However, it might not be the best criteria if you are seeking an economically interpretable value of your parameter estimates. In small samples, the estimates can be quite different from what you would expect, and also what you get by the MLE.

The set-up for PEM is exactly the same as in the MLE case. To make a distinction, you only use the option `'objective',prederr`. The complete syntax for PEM is

```
[P,mloglik,Grad,Hess,m,se2,F,pe,A,Pa,setup,pred,smooth] =
     estimate(m,d,range,E,'relative',true,'deviation',true,...
     'objective','prederr');
```

3.5 Posterior mode maximisation

The final option of the `estimate()` command that we cover in this chapter is the posterior mode maximisation (PMM). The objective function in this maximisation is a product of parameters' prior probability densities and data information. We refer to Hamilton (1994) for more on Bayesian estimation.

Running PMM is technically the same as running the MLE. Th eonly difference is that you supply their prior densities, when you declare parameters for estimation. A prior distribution comes as the fourth parameter in the set `E`. So for PMM, `E` contains: (i) parameter's initial value, (ii) lower, (iii) upper bound, and (iv) prior distribution (including parameters of the distribution, eg., mean and variance). If it happens that your prior is not laying fully within the lower-upper bound interval, `estimate()` will automatically truncate the distribution (the tails outside the lower-upper bound interval are cut off, and the distribution is re-scaled so that the integral of the prior distribution within the interval is 1).

Below is an sample set-up of PMM. We use the normal distribution for all parameters (including standard errors), centered around initial values, and with arbitrary standard deviation. Note that now when you are supplying the prior as an external function, you have to substitute the square brackets [ ], for curly braces {}. This is the way how you combine numerical values with functions in one string in Matlab.

Table 3.1: Estimates depending on the estimation criteria

	MLE	RMLE	RMLE(2)	PEM	PMM
χ	0.925	0.894	0.926	0.928	0.921
κ_w	0.018	0.029	0.017	0.049	0.025
κ_p	0.059	0.094	0.059	1.903	0.038
ρ_x	0.651	0.91	0.652	0.758	0.638
ρ_r	0.956	0.906	0.967	0.801	0.914
μ_p	1.1	1.387	1.396	2.026	1.46
μ_g	2.552	0.601	3.48	0.006	0.742
σ_{ec}	0.004	0.004	0.004	0.2	0.004
σ_{ew}	0.001	0.002	0.002	0.035	0.002
σ_{ep}	0.003	0.004	0.004	0.148	0.004
σ_{ex}	0.003	0.0004	0.003	0.088	0.004
σ_{er}	0.0004	0.001	0.0004	0.002	0.0005

The estimates are obtained for the EHL(2000) model, estimated on the US data from 1990:Iq to 2008:IIq.

```
E         =  struct();
E.chi     =  {0.80,0.50,0.95,@(x) normpdf(x,0.80,0.2)};
E.kappaw  =  {0.05,0.01,1 ,@(x) normpdf(x,0.05,0.2)};
E.kappap  =  {0.10,0.01,1 ,@(x) normpdf(x,0.10,0.2)};
E.rhox    =  {0.80,0.01,0.95,@(x) normpdf(x,0.80,0.2)};
E.rhor    =  {0.90,0.01,0.99,@(x) normpdf(x,0.90,0.2)};
E.mup     =  {1.50,1.10,5 ,@(x) normpdf(x,1.50,0.2)};
E.mug     =  {0.50,0.01,5 ,@(x) normpdf(x,0.50,0.2)};];

E.std_ec  =  {0.01,0.001,0.20 ,@(x) normpdf(x,0.01,0.005)};];
E.std_ew  =  {0.01,0.001,0.20 ,@(x) normpdf(x,0.01,0.005)};];
E.std_ep  =  {0.01,0.001,0.20 ,@(x) normpdf(x,0.01,0.005)};];
E.std_ex  =  {0.001,0.0001,0.02,@(x) normpdf(x,0.001,0.0005)};];
E.std_er  =  {0.01,0.001,0.20 ,@(x) normpdf(x,0.01,0.005)};];
```

And again we use the `estimate()` command:

```
[P,mloglik,Grad,Hess,m,se2,F,pe,A,Pa,setup,pred,smooth] =
  estimate(m,d,startdate:enddate,E,'relative',true,'deviation',true);
```

Iris offers a decomposition of the Hessian matrix evaluated at the posterior mode. It is a very handy tool. After running the PMM, print the Hessian matrix `Hess` in the command window. You will get two matrices. The first is the Hessian at the posterior mode, and the second matrix is the Hessian evaluated for the prior information. Subtracting the latter from the former, you get the value of the Hessian due to the data. Thus you learn, in a very elegant and straight forward way, how much information you are gaining from the data, and how much your estimates are influenced by the prior information.

For comparison, we report in Table 3.1 estimates that we obtain for all five estimation criteria discussed in this chapter.

3.6 Matlab code

```
%% Use a range of methods to estimate non-steady-state parameters

%%
clear all;

%% Load the model and data.

m = loadstruct('..\Model\EHL2000.mat');
d = loadstruct('..\Data\model_data_gaps.mat');
startdate = qq(1990,1);
enddate = qq(2008,2);

%% Constrained maximum likelihood.
% Specify a database with the parameters that are to be
% estimated as 1x3 vectors:
%
%  E.parameter_name = [initial-value, lower-bound, upper-bound];
%

E = struct();

E.chi    = [0.8,0.5,0.95];
E.kappaw = [0.05,0.01,1];
E.kappap = [0.10,0.01,1];
E.rhox   = [0.8,0,0.95];
E.rhor   = [0.9,0,0.99];
E.mup    = [1.5,1.1,5];
E.mug    = [0.5,0,5];

E.std_ec = [0.01,0.001,0.20];
E.std_ew = [0.01,0.001,0.20];
E.std_ep = [0.01,0.001,0.20];
E.std_ex = [0.001,0.0001,0.02];
E.std_er = [0.01,0.001,0.20];

[P_mle,mloglik,Grad,Hess,m_mle,se2_mle,F,pe,A,Pa, ...
            setup,pred,smooth] = estimate(m,d,startdate:enddate,E,...
                                 'relative',true,'deviation',true);
```

```
%% Regularised maximum likelihood

% A.) Regularised likelihood in all dimensions
t_ratio = 2;

E = struct();

E.chi    = [0.8,0.5,0.95,(t_ratio/0.80)ˆ2/2];
E.kappaw = [0.05,0.01,1 ,(t_ratio/0.05)ˆ2/2];
E.kappap = [0.10,0.01,1 ,(t_ratio/0.10)ˆ2/2];
E.rhox   = [0.8,0,0.95  ,(t_ratio/0.80)ˆ2/2];
E.rhor   = [0.9,0,0.99  ,(t_ratio/0.90)ˆ2/2];
E.mup    = [1.5,1.1,5   ,(t_ratio/1.50)ˆ2/2];
E.mug    = [0.5,0,5     ,(t_ratio/0.50)ˆ2/2];

E.std_ec = [0.01,0.001,0.20  ,(t_ratio/0.01)ˆ2/2];
E.std_ew = [0.01,0.001,0.20  ,(t_ratio/0.01)ˆ2/2];
E.std_ep = [0.01,0.001,0.20  ,(t_ratio/0.01)ˆ2/2];
E.std_ex = [0.001,0.0001,0.02,(t_ratio/0.001)ˆ2/2];
E.std_er = [0.01,0.001,0.20  ,(t_ratio/0.01)ˆ2/2];

[P_rmle1,mloglik,Grad,Hess,m_rmle1,se2_rmle1,F,pe,...
    A,Pa,setup,pred,smooth] = estimate(m,d,startdate:enddate,E,...
                  ’relative’,true,’deviation’,true,’penalty’,1);



% B.) Partialy regularised likelihood
t_ratio = 2;

E = struct();

E.chi    = [0.8,0.5,0.95, 0];
E.kappaw = [0.05,0.01,1 , 0];
E.kappap = [0.10,0.01,1 , 0];
E.rhox   = [0.8,0,0.95  , 0];
E.rhor   = [0.9,0,0.99  , 0];
E.mup    = [1.5,1.1,5   ,(t_ratio/1.50)ˆ2/2];
E.mug    = [0.5,0,5     , 0];

E.std_ec = [0.01,0.001,0.20  , 0];
E.std_ew = [0.01,0.001,0.20  , 0];
E.std_ep = [0.01,0.001,0.20  , 0];
E.std_ex = [0.001,0.0001,0.02, 0];
E.std_er = [0.01,0.001,0.20  , 0];

[P_rmle2,mloglik,Grad,Hess,m_rmle2,se2_rmle2,F,pe,...
 A,Pa,setup,pred,smooth] = estimate(m,d,startdate:enddate,E,...
                  ’relative’,true,’deviation’,true,’penalty’,1);
```

```
%% Prediction error minimisation
E = struct();

E.chi    = [0.8,0.5,0.95];
E.kappaw = [0.05,0.01,1];
E.kappap = [0.10,0.01,2];
E.rhox   = [0.8,0,0.95];
E.rhor   = [0.9,0,0.99];
E.mup    = [1.5,1.1,10];
E.mug    = [0.5,0,5];

E.std_ec = [0.01,0.001,0.20];
E.std_ew = [0.01,0.001,0.20];
E.std_ep = [0.01,0.001,0.20];
E.std_ex = [0.001,0.0001,0.1];
E.std_er = [0.01,0.001,0.20];

[P_pem,mloglik_pem,Grad,Hess_pem,m_pem,se2_pem,F,pe,...
  A,Pa,setup,pred,smooth] = estimate(m,d,startdate:enddate,E,...
             'relative',true,'deviation',true,'output',dbase,...
                          'objective','prederr','maxiter',5000);


%% Posterior mode maximisation
E = struct();

E.chi    = {0.8,0.5,0.95,@(x) normpdf(x,0.80,0.2)};
E.kappaw = {0.05,0.01,1, @(x) normpdf(x,0.05,0.2)};
E.kappap = {0.10,0.01,2, @(x) normpdf(x,0.10,0.2)};
E.rhox   = {0.8,0,0.95,  @(x) normpdf(x,0.80,0.2)};
E.rhor   = {0.9,0,0.99,  @(x) normpdf(x,0.90,0.2)};
E.mup    = {1.5,1.1,10,  @(x) normpdf(x,1.50,0.1)};
E.mug    = {0.5,0,5,     @(x) normpdf(x,0.50,0.2)};

E.std_ec = {0.01,0.001,0.20, @(x) normpdf(x,0.01,0.2)};
E.std_ew = {0.01,0.001,0.20, @(x) normpdf(x,0.01,0.2)};
E.std_ep = {0.01,0.001,0.20, @(x) normpdf(x,0.01,0.2)};
E.std_ex = {0.001,0.0001,0.1,@(x) normpdf(x,0.001,0.2)};
E.std_er = {0.01,0.001,0.20, @(x) normpdf(x,0.01,0.2)};

[P_mode,mloglik_mode,Grad,Hess_mode,m_mode,se2_mode,F,pe,...
  A,Pa,setup,pred,smooth] = estimate(m,d,startdate:enddate,E,...
                'relative',true,'deviation',true,'output',dbase);

% This file is used later (for forecasting)
savestruct('../Model/EHL2000_mode.mat'   ,m_mode);
```

Chapter 4

Evaluation

In this chapter, you learn the basic commands in Iris that you use for evaluation of the model data fit. We show how to check the basic data matching properties of the structural model. We focus the on the second moments implied by the model, and compare them to the second moments implied by an unrestricted, statistical model, which is treated as a benchmark. The spectral analysis is covered here as an advanced topic. We again compare the model implied spectral densities with the benchmark model. It is a complex way for checking the properties of a structural model.

4.1 Comparison second moments

Comparing how well a model matches data moments belongs among basic tools for assessing the model empirical fit. In Iris, the problem of computing the complete set of second moments is solved by the command `acf()`. It uses known analytical expressions to compute model implied second moments. In this section, we compare these model properties, to an unrestricted VAR(4) model estimated on the US data.

You can analytically solve and evaluate the implied second moments for any reduced form model (see e.g. Hamilton 1994, p ???). `acf()` does exactly that. It is much faster, and you avoid any Monte carlo experiments.[1] You

[1]It might a disappointment if you are that kind of person that likes MC experiments.

only supply a model structure, and the order of autocorrelation function (default is 0) to `acf()`, and as an output, you obtain autocovariance (AC) and autocorrelation (AR) matrices. If the order of AC and AR functions is non-zero, the AC and AR matrices are 3-dimensional. The general structure of AC matrix is

$$AC_k = \begin{pmatrix} cov(x_t^1, x_{t-k}^1) & cov(x_t^1, x_{t-k}^2) & ... & cov(x_t^1, x_{t-k}^n) \\ cov(x_t^2, x_{t-k}^1) & cov(x_t^2, x_{t-k}^2) & ... & cov(x_t^2, x_{t-k}^n) \\ ... & ... & ... & ... \\ cov(x_t^n, x_{t-k}^1) & cov(x_t^n, x_{t-k}^2) & ... & cov(x_t^n, x_{t-k}^n) \end{pmatrix} \tag{4.1}$$

It is useful to remember the basic logic of structure (4.1), because you need to know it when summarising or plotting the results (in particular for high-dimensional models). The elements of the AC matrix are the second moments among n variables. The third dimension to the matrix comes through time lag $k = 0, ..., p \geq 0$. If $k = 0$, its default value, the AC matrix is a standard 2D matrix. Of course, the AR matrix is of the same structure as AC.

Similarly as in the previous chapters, we recommend to open an m-file where you run the sequence of commands from, and work with the results. In the new file again, first you have to upload the model and data structures: `loadstruct('EHL2000.mat')` and `loadstruct('US_data.mat')`. And set the time horizon for simulations: `starthist`, and `endhist`.

Now everything is ready to compute the AC and AR functions for the EHL(2000) model

```
order   = 2;
[AC,AR] = acf(m,order);
```

In the Matlab command window, print the AC and AR matrices. Model implied variances lay on the diagonal of the AC matrix for $k = 0$. You can print them on the screen by executing `diag(AC{1})`. The parameter `{1}` here relates to the part of the AC matrix that corresponds to $k = 0$. The off-diagonal elements are cross-covariances.

You do not already avoid Monte Carlo (MC) simulations (or Bootstrap) if you seek the intervals of significance for the estimates. Unfortunately, there is not a simple analytical solution available. Obtaining, the confidence intervals is straightforward, but as any MC or Bootstrap experiments, it takes time to run. In principle, the procedure takes three steps:

1. Simulate the structural model (`resample()`)
2. Estimate unrestricted VAR on the simulated model variables (`rvar()`)
3. Evaluate autocovariance and autocorrelation functions (`acf()`)

4.1.1 Monte Carlo experiment

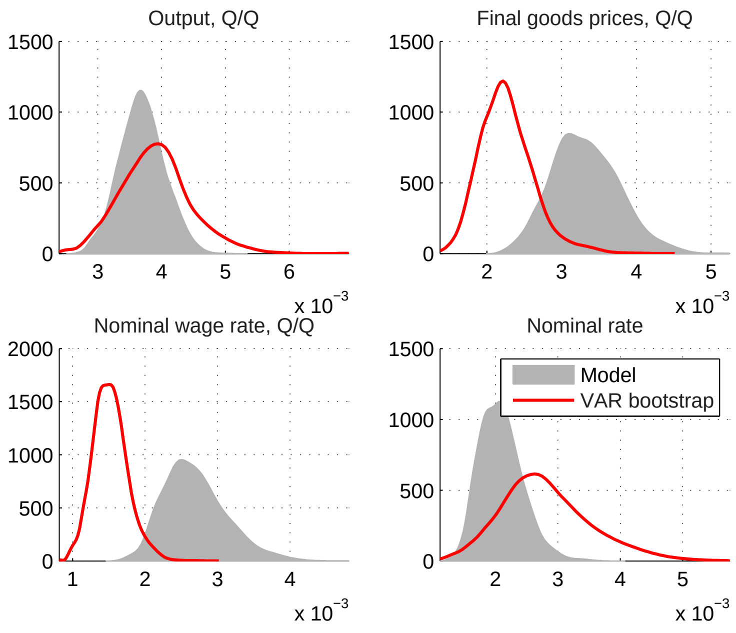
For demonstration purposes, we take the Monte Carlo approach to simulate the model. The code below follows the three steps. First, we simulate the EHL(2000) model as in Chapter 1 for 1,000 times using the command `resample()`. Then we use the database of model observable variables, and estimate a VAR(1) model on them. We use the command `rvar()` for that. The choice of the order of the VAR model depends on the order of the reduced form of the original structural model. In our case, it is 1. As the third step, we evaluate the AC and AR functions implied by the estimated VAR by using the `acf()` command again

```
ndraw = 1000;

% Monte Carlo simulate the model.
Ym = resample(m,d,starthist-4:endhist,ndraw);
Wm = rvar(Ym,ylist,starthist:endhist,'order',4);

[C_mc, R_mc] = acf(Wm,'order',1);
```

The strength of Iris lays in its simplicity and efficiency of coding. Note that to run Monte Carlo or Bootstrap experiments, you do not need to code any loops, and that all simulations are done within the single execution of the function `resample()`.

The best way to analyse and present the data matching results is plotting them. In Figure 4.1, we show the distribution of standard errors. The shadow area depicts the model implied distribution. Similarly, Figure 4.2 shows the cross-correlations. Remember that they are estimates that hold asymptotically. To visualise how well the model matches the data moments, it is useful to plot them next to data implied distributions.

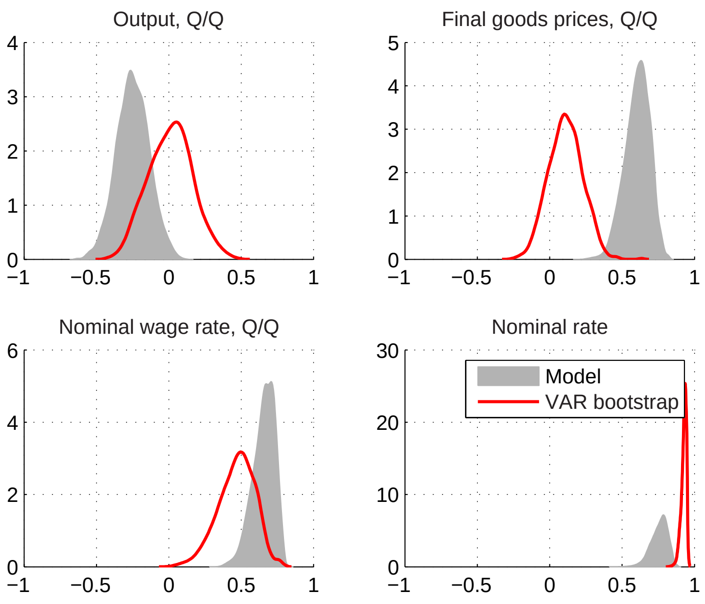
Figure 4.1: Standard deviations: model vs. data



4.1.2 Bootstrap

The second moments of the data are obtained in a similar way as for the model. We estimate a VAR(p) on the data, compute second moments, and then simulate the model to obtain their distributions. In the code below, the order of VAR model is picked arbitrarily, but you can use a selection criteria to pick the right order. To practice a different method, we show you how to use the Bootstrap technique to generate distributions in Iris. The VAR exercise serves well to this purposes, because Bootstrap is based on random draws from residuals. The code bellow bootstraps a VAR(4) model on the US data:

Figure 4.2: Autocorrelation: model vs. data



```
Yb = resample(m_var4,data_var,ndraw,...
                                'distribution','bootstrap','wild',true);
Wb = rvar(Yb,Inf,'order',4);
[C_var4,R_var4] = acf(Wb,'order',1);
```

DOPLNIT!!!! The Bootstrap experiment is performed by the `resample()` command and employing the option `'bootstrap','wild'`. The 'wild' option allows for a possibility of heteroscedastic residuals, which makes the Bootstrap more robust (see **??**).

The Bootstrap results are plotted together with the structural model results in Figures 4.1 and 4.2.

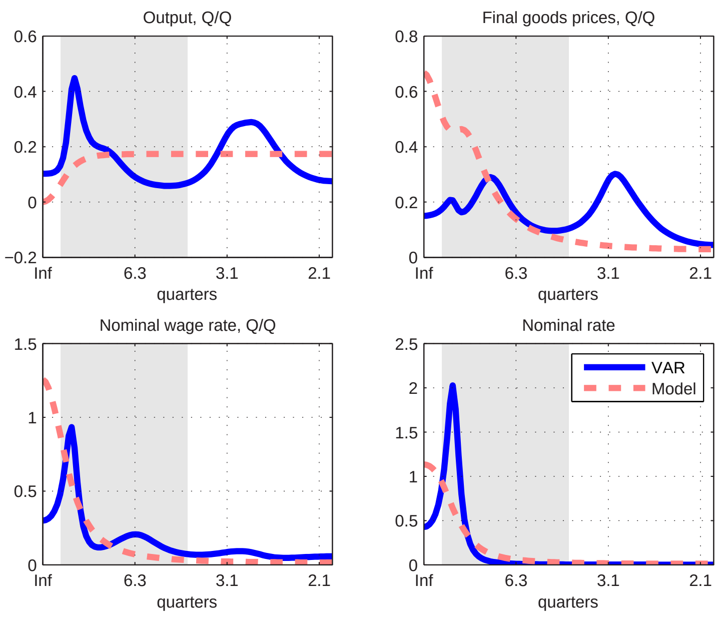
4.2 Spectral analysis

A spectral analysis provides a complex view on models and their data matching properties. A spectral density summarises information about second moments. We refer to Hamilton (1994, chapter 6) for details. Iris has the function `xsf()`, which performs the analysis. The input arguments are (i) the model to be evaluated, (ii) frequency for evaluation. As an output you receive a power spectrum function, and spectrum density function. An example of the use of `xsf()` is below

```
% VAR
frq = 0 : pi/100 : pi;
[m_var4,data_var] =
rvar(d,ylist,starthist:endhist,'order',4,'comment',ylabels);
[psz,sdz]          = xsf(m_var4,frq);

% EHL2000 Model
[psm,sdm] = xsf(m,frq);
```

The model implied and data (approximated by a VAR model) spectra are plotted in Figure 4.3. The continuous line represents the model spectrum density, and dashed line is the data spectrum density. The gray area highlights business cycle frequencies.

Figure 4.3: Model and data spectra



The gray area highlights business cycle frequencies.

4.3 Matlab code

```
%% Model evaluation: Second momements, and spectral analysis

%% Clear memory of everything.
clear all;
close all;

%% Set up working environment
m = loadstruct('../Model/EHL2000_rmle.mat'); % Model
d = loadstruct('../Data/US_data_gaps.mat'); % Data

starthist = qq(1990,1);
endhist = qq(2008,2);

ylist   = get(m,'ylist'); % list of measurement variables
ylabels = get(m,'ycomments'); % comments attached to measurement variables

%% Spectral density
% Estimate reduced-form VAR.
% Compute power spectrum and spectral density.
% Compute the same characteristics implied by the model.

% VAR
frq = 0 : pi/100 : pi;
[m_var4,data_var] = rvar(d,ylist,starthist:endhist,'order',4,'comment',ylabels);
[psz,sdz]        = xsf(m_var4,frq);

% EHL2000 Model
[psm,sdm] = xsf(m,frq);

% Plotting
figure('defaultaxesfontsize',8,'defaultlinelinewidth',1,...
      'defaulttextinterpreter','none');
N = 4;
for i = 1 : N
  subplot(2,2,i);
  plot(frq,[vec(sdz(i,i,:)),vec(sdm(i,i,:))]);
  hold on;
  set(gca,'xlim',[0,pi]);
  grid on;
  xlabel quarters;
  title(ylabels{i});
  xtickfrq2per(gca());
  if i == N
      legend('VAR','Model','location','northeast');
  end
  highlight(gca(),2*pi./[32,4]);
end
```

```
%% ACF
% Model implied second momemnts
[C,R,acf_list] = acf(m,'order',1);
disp(size(C));
disp('1nd-order correlation matrix:');
disp(acf_list(1:4));
disp(R(1:4,1:4,2));

%% Bootstrap and Monte Carlo simulation
% Comparing the model to the data
ndraw = 10;

% Bootstrap resample from the estimated VAR.
Yb = resample(m_var4,data_var,ndraw,'distribution','bootstrap','wild',true);
return
Wb = rvar(Yb,Inf,'order',4);
[C_var4,R_var4]  = acf(Wb,'order',1);

% Monte Carlo simulate the model.
Ym = resample(m,d,starthist-4:endhist,ndraw);
Wm = rvar(Ym,ylist,starthist:endhist,'order',4);
[C_mc, R_mc] = acf(Wm,'order',1);


%% PLOTTING THE RESULTS
%% Standard errors
f = [];
mdlcolour = 0.7*[1,1,1];

f(end+1) = figure('defaultlinelinewidth',1,'defaultaxesfontsize',10);
for i = 1 : N
  subplot(2,2,i)
  hold on;
  % model distribution
  mm = vec(sqrt(C_mc(i,i,1,:)));
  mm(2:end) = mm(2:end) - mean(mm(2:end)) + mm(1);
  [fx,x] = ksdensity(mm(2:end),'support','positive');
  area(x,fx,'facecolor',mdlcolour,'edgecolor',mdlcolour);
  % VAR distribution
  ww = vec(sqrt(C_var4(i,i,1,:)));
  [fx,x] = ksdensity(ww,'support','positive');
  plot(x,fx,'linewidth',1.5,'color','red');
  ylim = get(gca,'ylim');
  set(gca,'ylimmode','manual','xgrid','on','ygrid','on','xlim',...
       [min([mm;ww]),max([mm;ww])]);
  title(ylabels{i});
  if i == 4, l = legend('Model','VAR bootstrap','location','northeast'); end
end
```

```
%% Autocorrelations

f(end+1) = figure('defaultlinelinewidth',1,'defaultaxesfontsize',10);
for i = 1 : 4
  %nextsubplot(3,3);
  subplot(2,2,i)
  hold on;
  % model distribution
  mm = vec(R_mc(i,i,2,:));
  mm(2:end) = mm(2:end) - mean(mm(2:end)) + mm(1);
  [fx,x] = ksdensity(mm(2:end),'support',[-1,1]);
  area(x,fx,'facecolor',mdlcolour,'edgecolor',mdlcolour);
  % VAR distribution
  ww = vec(R_var4(i,i,2,:));
  [fx,x] = ksdensity(ww,'support',[-1,1]);
  plot(x,fx,'linewidth',1.5,'color','red');
  ylim = get(gca,'ylim');
  set(gca,'ylimmode','manual','xgrid','on','ygrid','on','xlim',[-1,1]);
  title(ylabels{i});
  if i == 4, l = legend('Model','VAR bootstrap','location','northeast'); end
end
```

Chapter 5

Forecasting

In this chapter you learn the basic Iris commands to produce forecasts from a structural model. The forecasting exercise by itself is very simple, it basically consists of simulating the model out of sample. It takes more coding to present the forecast than forecasting itself. You will also learn how to produce confidence intervals, fan charts, and as an advanced topic, you learn the principles of imposing judgement on the forecasts. This exercise is particularly useful for Iris users in central banks.

5.1 Point forecasts

The point forecast are produced by simulating the structural model out of sample. But to get the forecast right, we need to have correct initial conditions: historical shock and state variables estimates. If you estimate your model before forecasting out of it, you should save the database with the smoothed estimates. If you calibrate the model, you have to back out the smoothed estimates first, and then forecast. For demonstration purposes we take the latter approach here. But in the sake of using realistic values for standard deviations of the shocks, we use the estimates from Chapter 4.

5.1.1 Initial conditions

To produce forecasts we need to know the estimates of historical shocks and model state variables, because it is their cumulative effect that drives the forecasts. If the model is satisfactorily parametrised, we evaluate the historical database by using the Iris command `filter()`. In a single run of the Kalman filter it estimates the model state variables, shocks, and evaluates the likelihood for the given set of parameters.

As in all exercises before, we recommend to open a new m-file and execute the sequence of commands from there, rather than typing them directly in to the Matlab command window. You start the code by loading the model and data, and by setting up the time dimension for the simulation and forecast. The historical range spans from 1990:Iq to 2008:IIq. The forecast horizon is from 2008:IIIq to 2011:IVq.

```
m = loadstruct('HNL2000.mat');
d = loadstruct('US_data.mat');

starthist = qq(1990,1);
endhist = qq(2008,2);

startfcast = qq(2008,3);
endsfacst = qq(2011,4);
```

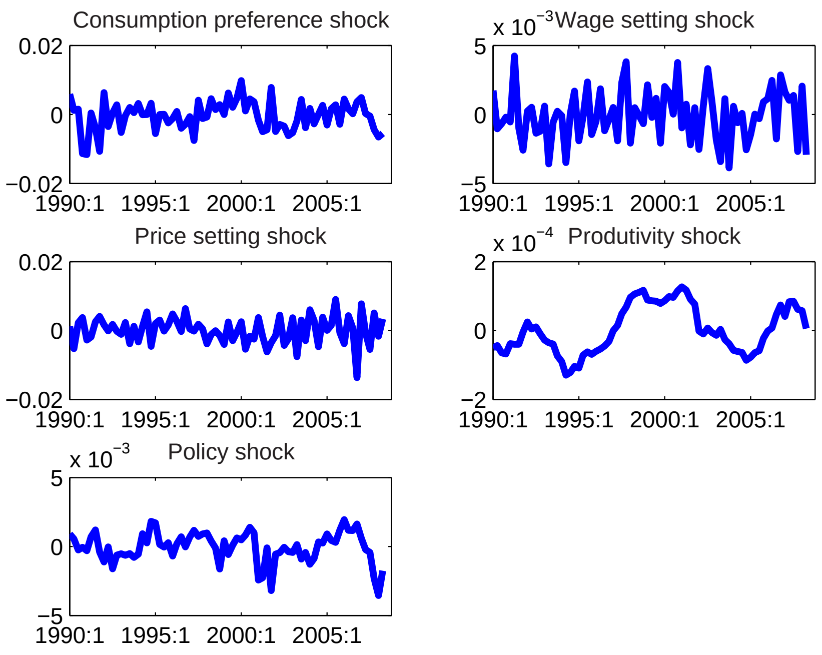
Now the basic working environment is set and we are ready to run the Kalman filter:

```
[obj,pe,pred,smooth] =
   filter(m,d,starthist+1:endhist,'deviation',true,'output','dbase');
```

The set of the smoothed estimates of shocks and states is stored in the output database called `smooth`. In their further use, we will denote them as `init_cond = smooth.mean`. Similarly the set of predicted values (coming from the prediction step of the Kalman filter) is stored in the `pred` database. We plot the smoothed estimates of the shocks in Figure 5.1.

An equivalent result can be obtained by using the command `simulate()`. You usually use it if you are interested in the decomposition of model variables in to the contribution of individual shocks. You declare `'contributions',true`

Figure 5.1: Estimates of historical shocks



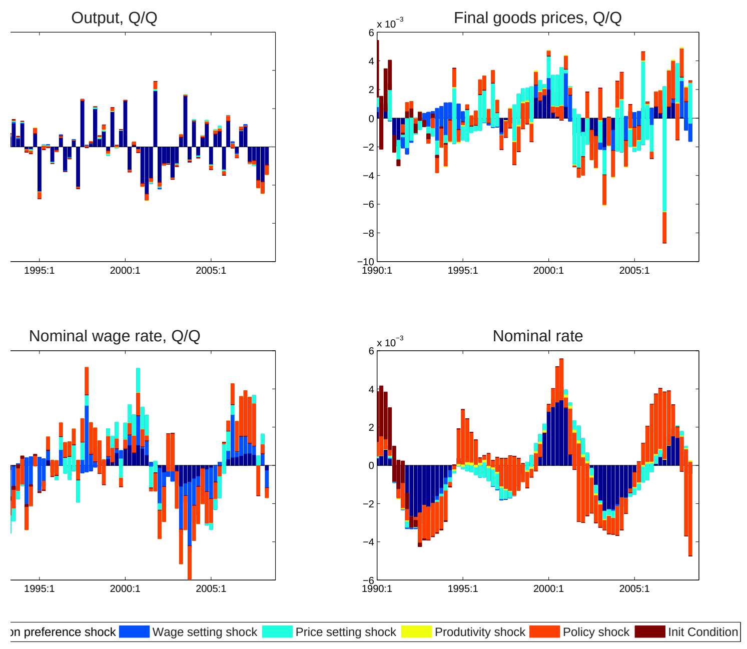
Note: The estimates are obtained for EHL(2000) model, estimated in the US data sample from 1990:Iq to 2008:IIq.

as one of the options, and the output will be a database with decomposed variables. An example of such decomposition is shown in Figure 5.2. Here is the Iris code that produces the figure:

```
s = simulate(m,fcast_init,startdate:enddate,...
        'anticipate',false,'deviation',true,'contributions',true)';
```

If you check the dimension of the decomposed variables, you will note that except of the contribution of exogenous shocks, the variable has one extra dimension, The extra dimension carries the effect of (historical) initial condition. That is why `size(s.dy_)` will give you the dimension (x-by-6) instead of (x-by-5), for the EHL(2000) model, which has 5 structural shocks. You can see the effect of initial condition in Figure 5.2.

Figure 5.2: Historical decomposition of observable variables



Note: The estimates are obtained for EHL(2000) model, estimated in the US data sample from 1990:Iq to 2008:IIq.

5.1.2 Forecasts

When we have the shocks and state variables estimated, to get a point forecast takes only one step. Given the starting conditions, the model is simulated forward. For that we can employ the Iris command `filter()` again. As input, we use the database with initial conditions obtained above, and specify the forecasting horizon `startfcast:endfcast`:

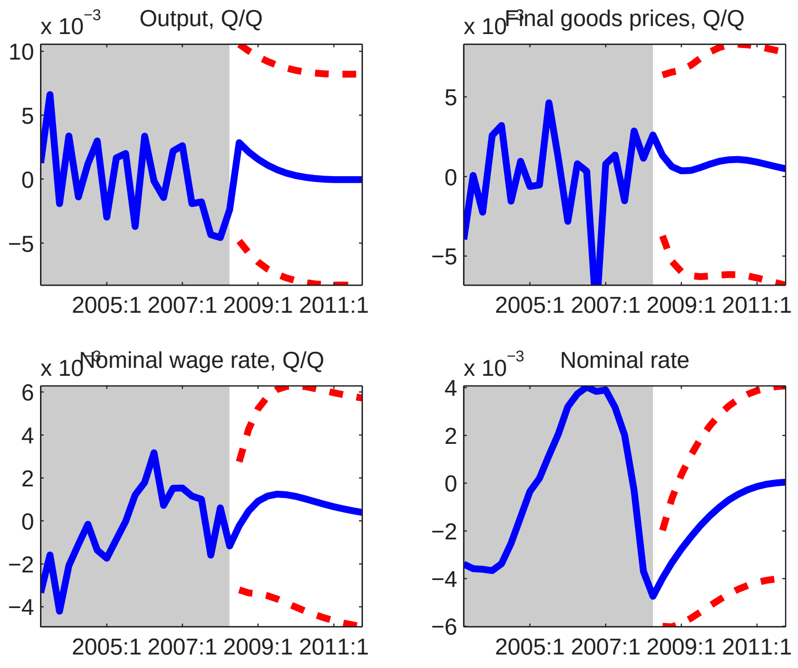
```
[obj,pe,pred,smooth] = filter(m,init_cond,startfcast:endfcast,...
                               'deviation',true,'output','dbase');
mfcast = smooth.mean;
```

The model forecasts are stored in the database named `mfcast`, and we plot them in Figure 5.3.

5.2 Confidence intervals

The simplest way to construct confidence intervals, and the fastest one, is to use the Iris command `fmse()`, which uses an analytical solution for the forecast mean-square error estimates on as specified forecast horizon.

Figure 5.3: Point forecasts of the EHL(2000) model



Note: The shadow area highlights the historical sample. The last point in that area is the last available data observation. The forecast horizon spans from 2008:IIIq to 2011:IVq. The dashed lines set 95% confidence intervals for the point forecast.

The `fmse()` uses the distribution of estimated (or calibrated) exogenous shocks. The syntax is

```
[CI,listCI,ci] = fmse(m,startfcast:endfcast);
```

`ci` stores the model implied forecast mean-square errors. The time dimension to the series represents the forecast horizon. Each forecast error is then conditional on the information available at the last observation.

In Figure 5.3 we plot the point forecast together with the 95% confidence bands constructed as $\pm$`2*ci`.

5.3 Matlab code

```
%% Using the model for forecasting

%% Clear memory and screen of everything.
close all
clear all
format short
clc

%% Set up working environment
% Read estimated model and data. Set-up dates for historical simulations
% and forecast horizon.
m = loadstruct('../Model/EHL2000_rmle.mat'); % Model
d = loadstruct('../Data/US_data_gaps.mat');  % Data

startdate = qq(1990,1);
enddate   = qq(2008,2);
startfcast= qq(2008,3);
endfcast  = qq(2011,4);

ylist   = get(m,'ylist');
ylabels = get(m,'yComments');
elist   = get(m,'elist');
elabels = get(m,'eComments');

%% Initial conditions
% Estimate model state variables and historical shocks to get initial
% conditions for the forecasts. If we seek a story-telling forecast, we
% can decompose the model series in to shock contributions.

% Initial conditions
[m,smooth,se2,delta,pe,F] = filter(m,d,startdate:enddate,...
                          'relative',false,'deviation',true);
fcast_init = smooth.mean;

% Historical decompositions
s = simulate(m,fcast_init,startdate:enddate,...
          'anticipate',false,'deviation',true,'contributions',true)';

figure()
for i = 1:length(ylist)
    subplot(2,2,i)
    bar(s.(ylist{i}),'stack');
    title(ylabels{i});
    if i == 4, l = legend(elabels,'location','Best'); end
end
```

```
%% Point forecast
% Producing point forecast. Simulating the model out of sample
% Forecast
%f = simulate(m,fcast_init,startfcast:endfcast,...
%     ’anticipate’,false,’deviation’,true)’;
[m,smooth,se2,delta,pe,F] = filter(m,fcast_init,startdate:endfcast,...
                                 ’relative’,false,’deviation’,true);
f = smooth.mean;

%[unc,con] = forecast(m,fcast_init,startfcast:endfcast,[],...
             ’deviation’,true,’anticipate’,false);
%unc = unc.mean;
%con = con.mean;
%f   = dboverlay(fcast_init,unc);
%return

%figure()
%for i = 1:length(ylist)
%    subplot(2,2,i)
%    plot([unc.(ylist{i}) con.(ylist{i})])
%    title(ylabels{i});
%    axis tight;
%end





%% Confidence intervals
[CI,listCI,ci] = fmse(m,startfcast:endfcast);


figure()
for i = 1:length(ylist)
    subplot(2,2,i)
    UB = f.(ylist{i})+2*ci.(ylist{i});
    LB = f.(ylist{i})-2*ci.(ylist{i});
    plot([UB LB],’-- r’)
    hold on
    axis tight;
    plot(enddate-20:endfcast,f.(ylist{i}))
    title(ylabels{i});
    grayarea(enddate-20:enddate);
    hold on
end
```

Bibliography