Manual for

PROMES

a projection method solver for Matlab

Sijmen Duineveld

Promes, version 05.0.0.a
December 20, 2021

Contents

I Manual **4**

1 Introduction **5**
- 1.1 Core of the toolbox 5
- 1.2 Getting started 6
- 1.3 Installation 6
- 1.4 Algorithms 8
- 1.5 Remarks 10
- 1.6 Acknowledgments, sources, and license 11

2 Introduction to projection methods **12**
- 2.1 Simple Life Cycle model 12
- 2.2 Projection explained 13
- 2.3 Spline methods 15
- 2.4 Monomial basis function 16
- 2.5 Chebyshev polynomials 17
- 2.6 Smolyak algorithm 19

3 Basic Procedure **21**
- 3.1 Deterministic Brock-Mirman model 22
- 3.2 Approximation of the policy function 22
- 3.3 Basic Procedure 23

4 Detailed Procedure **28**
- 4.1 Standard RBC model 28
- 4.2 Step 0: Matlab Settings 28
- 4.3 Step 1: Initial Block 30
- 4.4 Step 2: Construct the grid 31
- 4.5 Step 3: Model function and handle 34
- 4.6 Step 4: Initial guess for the policy function 40
- 4.7 Step 5: Solving the policy function 42
- 4.8 Step 6: Evaluating the policy function 43
- 4.9 Performance 45
- 4.10 Sensitivity Gauss-Hermite quadrature 47

5 Multiple policy variables **48**
5.1 Housing model . . . 49
5.2 Policy function . . . 49
5.3 Residuals . . . 51

II Theoretical description of algorithms **52**

6 General Approach **53**
6.1 Defining a grid . . . 54
6.2 Collocation . . . 54

7 Algorithms **56**
7.1 Splines . . . 56
7.2 General: construction of polynomials . . . 58
7.3 Monomials . . . 61
7.4 Complete Chebyshev polynomials . . . 62
7.5 Smolyak's algorithm . . . 64

III Technical descriptions of functions **68**

8 Model file **69**
8.1 Formats for policy function evaluation . . . 70
8.2 Function `get_pol_var` . . . 71

9 Solving the model **72**
9.1 Function `solve_proj` . . . 72

10 Construction of grid **74**
10.1 Function `prepgrid` . . . 75
10.2 Function `gridstruct` . . . 75
10.3 Function `gridstruct_smolyak` . . . 76
10.4 Example `grid_example` . . . 77
10.5 Function `constr_vecs` . . . 80
10.6 Function `constr_grid` . . . 81
10.7 Function `poly_elem_ani` . . . 82
10.8 Function `constr_univar_basis` . . . 82
10.9 Function `get_poly_ani` . . . 83
10.10 Function `scal_mat_up` . . . 83
10.11 Function `sc_cheb_up` . . . 84
10.12 Function `scal_mat_dw` . . . 84
10.13 Function `sc_cheb_dw` . . . 85
10.14 Function `chebnodes` . . . 85

IV Example models 86

11 Deterministic Brock-Mirman model 87

12 Standard RBC model 88
12.1 Model 88
12.2 Gauss-Hermite quadrature 90
12.3 Steady state 90

13 Housing model 91

Bibliography 93

Part I

Manual

1. Introduction

The `Promes` toolbox solves Dynamic Stochastic General Equilibrium models using projection methods. `Promes` is an acronym for Projection method solver. The toolbox is written for Matlab, and tested with Matlab 2016b and 2019a.

The toolbox offers several algorithms to solve a model. The policy function can be approximated with either a spline, a complete polynomial, or a Smolyak polynomial[1]. The toolbox is both fast[2] and accurate. For example, for a standard Real Business Cycle (RBC) model a maximum error of 10^{-6} can be achieved in less than 0.05 seconds with each of the basis functions. A maximum error of 10^{-13} can be achieved in 0.3 seconds with a complete Chebyshev polynomial. Due to the curse of the dimensionality, computation times do increase strongly with the number of state variables. For example, a model with four continuous state variables and two policy variables is solved in 5 seconds with a spline (Duineveld, 2021)[3].

1.1 Core of the toolbox

Three functions form the core of the `Promes` toolbox. The first function is `prepgrid`, which constructs a grid taking the grid parameters and the algorithm as input. The second is `get_pol_var`, which evaluates the policy function taking the state variables as input. This function simplifies programming the model file and simulations. The third is `solve_proj`, which solves the model, given the algorithm, an initial guess, and the grid. The initial guess for the policy function should give the policy variable(s) at the gridpoints. The toolbox will internally construct the appropriate policy function, either a spline or a polynomial. These three functions are the only functions the modeler has to call directly when solving and simulating a model. The functions are placed in the main folder 'PROMES_v05.0.0', and are explained in Part III.

With these three functions solving a DSGE model with projection methods becomes relatively easy. The main task of the modeler is to program the model function. The model function has to compute the residuals of the objective function, given a grid and the policy function. The requirements for a model file are explained in Chapter 8. One has to pay special attention to the formats for evaluating the policy function (see Section 8.1).

[1]The Smolyak algorithm is implemented with the code written by Rafa Valero (2021). Smolyak Anisotropic Grid (https://www.mathworks.com/matlabcentral/fileexchange/50963-smolyak-anisotropic-grid), MATLAB Central File Exchange. Retrieved November 5, 2021. The algorithm is described in Judd et al. (2014).

[2]All reported computation times are on a Windows 10 PC with Matlab 2019a and a Ryzen 2700x processor, and without any parallel computing.

[3]This paper is available at www.saduineveld.com/research

1.2 Getting started

To get started with the toolbox it is recommended to go through the examples. The code of these examples can be found in the folder 'PROMES_v05.0.0\Examples'. Chapter 2 gives an introduction to projection methods, and explains the basic features of the toolbox. It describes a very simple non-recursive model, which is solved in the program `main_exp_proj`. This example plots the exact solution and the projection approximation.

For those familiar with projection methods it is recommended to start with Chapter 3, which describes a 6-step procedure to solve a model with the `Promes` toolbox. This procedure is explained with a very simple recursive model, the deterministic Brock-Mirman model. The program `main_det_bm_proj` solves this model following the six steps. This program also plots the policy function, and the errors.

A more detailed step-by-step guide can be found in Chapter 4, which is based on a standard Real Business Cycle (RBC) model with stochastic shocks. The code for this model is the script `main_stnd_rbc_proj`. As in the previous example the program will plot the policy function. In addition it will plot two stochastic simulations, and compute the errors.

The last example is the program `main_housing_proj`, which shows how to solve a model with multiple policy functions. It is an RBC model with housing as an extra asset.

In Part II the algorithms are described theoretically. In Part III the technical and coding details of the toolbox are discussed. In Part IV the equations of the example models are derived.

1.3 Installation

For the installation download the 'Promes_v05.0.0.a.zip' file from https://www.promestoolbox.com/[4], and unpack the file in a folder. This will add the folders 'PROMES_v05.0.0' and 'TOOLS' to the destination folder. The folders and files of the `Promes` toolbox are in the folder 'PROMES_v05.0.0' and are shown in Figure 1.1.

In order to use the `Promes` toolbox one needs to add the folder 'PROMES_v05.0.0' and the subfolders 'grid_subfun' and 'smolyak_subfun' to the search path. After unpacking the `zip` file in the folder 'C:\Myfolder' one can add 'PROMES_v05.0.0' and all its subfolders to the searchpath with the Matlab command:

```
addpath(genpath('C:\Myfolder\PROMES_v05.0.0'));
```

The folder 'PROMES_v05.0.0' also has a subfolder 'Examples' which contains five examples:

[4]Older versions are available at https://github.com/saduineveld/Promes_toolbox. The latest release is also found on https://www.saduineveld.com/tools.

Figure 1.1: PROMES folders and files

```
Folder: PROMES_v05.0.0
├── get_pol_var.m
├── prepgrid.m
├── solve_proj.m
├── Folder: grid_subfun
│   ├── chebnodes.m
│   ├── constr_grid.m
│   ├── constr_univar_basis.m
│   ├── constr_vecs.m
│   ├── get_poly_ani.m
│   ├── gridstruct.m
│   ├── gridstruct_smolyak.m
│   ├── poly_elem_ani.m
│   ├── sc_cheb_dw.m
│   ├── sc_cheb_up.m
│   ├── sc_mat_dw.m
│   └── sc_mat_up.m
├── Folder: smolyak_subfun
│   ├── Smolyak_Elem_Anisotrop.m
│   ├── Smolyak_Elem_Isotrop.m
│   ├── Smolyak_Grid.m
│   └── Smolyak_Polynomial.m
└── Folder: Examples
    ├── grid_example.m
    ├── main_det_bm_proj.m
    ├── main_housing.m
    ├── main_exp_proj.m
    ├── main_stnd_rbc_proj.m
    └── Folder: STND_RBC_mod
```

- `main_exp_proj` is used to explain the basics of projection methods, and the main features of the toolbox in Chapter 2;
- `grid_example` illustrates the construction of the grid, and is explained in Chapter 10;
- `main_det_bm_proj` demonstrates a basic procedure to use to toolbox as described in Chapter 3 with a very simple model, the deterministic Brock-Mirman model;
- `main_stand_rbc_proj` demonstrates some details of the toolbox as described in Chapter 4 with a standard RBC model with stochastic shocks. The subfolder 'STND_RBC_mod' contains additional functions needed to obtain the results;
- `main_housing_proj` shows how to solve a model with multiple policy functions as described in Chapter 5 with an RBC model that includes housing as an extra asset.

The examples `main_stand_rbc_proj` and `main_housing_proj` require the addition of the folder 'TOOLS' to the searchpath. This folder contains the function `hernodes`, and the perturbation toolbox `CSD`. The latter can be used to obtain an initial guess for the policy function.

1.4 Algorithms

This section gives a brief overview of the algorithms. More details are described in Chapter 7. Projection method algorithms consist of three main choices[5]: the basis functions, the projection condition and the solution method. An overview of the implemented algorithms and their three choices is given in Table 1.1. The basis functions are used for the approximation of the policy function. The basis functions in this toolbox are splines, complete polynomials, and Smolyak-Chebyshev polynomials.

The algorithms starting with `'spl'` use a spline[6] with equidistant nodes. A spline is a piece-wise polynomial. The algorithm names starting with `'mono'`, and `'cheb'` approximate the policy function with complete polynomials. The one called `'mono_mse'` uses equidistant nodes, and monomial basis functions $(1, x, x^2, x^3, \ldots)$, whereas the ones called `'cheb'` use complete Chebyshev polynomials with Chebyshev nodes. The algorithms starting with `'smol'` use Smolyak's algorithm, which relies on a sparse grid and a sparse Chebyshev polynomial[7].

The second choice is the projection condition. This toolbox includes collocation, Galerkin's method, and minimization of the squared errors.

[5] Gaspar and Judd (1997, Table 3) mention a fourth choice, which is the integration method to compute the expected value of future states of the economy.

[6] Splines are determined by Matlab's `griddedInterpolant`.

[7] See Footnote 1.

Table 1.1: Overview of algorithms

Algorithm	Basis function	Proj. Cond.	Solution Meth.
'spl_dir'	Spline	Collocation	Direct Comp.
'spl_tmi'	Spline	Collocation	Time Iteration
'cheb_gal'	Compl. Chebyshev polyn.	Galerkin	Newton type
'cheb_tmi'	Compl. Chebyshev polyn.	Collocation	Time Iteration
'cheb_mse'	Compl. Chebyshev polyn.	Min. Sq. Err.	Trust-Region
'mono_mse'	Monomials (compl. polyn.)	Min. Sq. Err.	Trust-Region
'smol_dir'	Smolyak-Chebyshev polyn.	Collocation	Direct Comp.
'smol_tmi'	Smolyak-Chebyshev polyn.	Collocation	Time Iteration

Collocation solves the model at the gridpoints. With Galerkin's method the residual function is orthogonal to the basis functions, similar to the Method of Moments (Judd, 1998). Minimization of the squared errors is used to obtain the coefficients of a polynomial that is overidentified.

The third choice is the solution method for the objective function. The toolbox uses four methods. The first is a Newton-type of non-linear equation solver. When a Newton-type of solver is used to solve the model at the gridpoints (collocation) we call this Direct Computation. The third method is Time Iteration, which is specifically designed to solve recursive dynamic optimization problems. The fourth method uses a Trust-Region algorithm to solve least squares problems.

The toolbox does not use the Fixed Point algorithm (Miranda and Helmberger, 1988), because it requires a different format of the model file than the other algorithms. The current format requires the model file to return the Euler residuals as output. This contrasts with the Fixed Point algorithm, which requires the policy variables as output (Gaspar and Judd, 1997).

The algorithm using a spline with Time Iteration (`'spl_tmi'`) is the most robust, because splines preserve the shape of the policy function well, and Time Iteration is the only solution method that should theoretically converge to the saddle path stable solution (Judd, 1998). A spline with Direct Computation should be preferred over Time Iteration when the number of gridpoints is relatively low, and convergence is not an issue.

For small, well-behaved models complete Chebyshev polynomials with the Galerkin projection condition (`'cheb_gal'`) might be preferred as this algorithm performs best for the Standard RBC model described later. For models with a high number of state variables the Smolyak algorithm is recommended, because it is very effective at tackling the curse of the dimensionality. For the Smolyak algorithm the number of gridpoints grows only polynomially in the number of state variables, while it grows exponentially for the other algorithms.

The algorithms using Minimization of the Squared Errors (`'mse'`) are not recommended for two reasons. The first is that Minimization of the Squared Errors can get stuck in a local minimum (Judd, 1992). The second reason is that the gradients of the residual function can be highly correlated, which makes it difficult to get an accurate result (Judd, 1992).

Monomial basis functions in the algorithm 'mono_mse' are only included for demonstration purposes, although they might perform very well for low order approximations of simple models. Monomial basis functions are not recommended for two reasons (Fernández-Villaverde, Rubio-Ramírez, and Schorfheide, 2016). The first is that they are highly collinear, especially for high order approximations. The second reason is that monomials are not scaled to have a similar magnitude.

The algorithm 'cheb_tmi' is not recommended either, unless there is a specific reason not to use splines. It will in general be outperformed by splines with Time Iteration ('spl_tmi'). The reason is that complete Chebyshev polynomials are overidentified when rectangular grids are used[8]. This means the polynomial will not go through the solution at the gridpoints, while a spline will go through these points. In general, for small scale problems 'cheb_gal' will outperform 'cheb_tmi', but for larger problems 'cheb_tmi' is the better choice.

For complete polynomials we recommend to set the number of nodes equal to the order plus 1 (see also Fernández-Villaverde, Rubio-Ramírez, and Schorfheide, 2016). This is the minimum number of gridpoints for the algorithms using complete Chebyshev polynomials. For splines a low number of nodes usually suffices for reasonable accuracy. For example for the Standard RBC model only 3 nodes in each dimension results in errors of similar magnitude as the third order perturbation solution.

The accuracy of the results depends on the stopping criteria, especially with Time Iteration. For Time Iteration tighter stopping criteria will reduce the errors at the gridpoints, and can improve the accuracy significantly. This is discussed in more detail in Section 4.7 for the Standard RBC model.

1.5 Remarks

Matlab toolboxes

All methods require Matlab's `Optimization toolbox`. We use `fsolve` for the algorithms with Time Iteration ('tmi'), Direct Computation ('dir'), or Galerkin ('gal'). We use `lsqnonlin` for the algorithms using Minimization of Squared Errors ('mse'). Alternatively one could use his/her own equation solver or minimization routine. This requires replacing the functions `fsolve` and `lsqnonlin` in the function `solve_proj`.

Notes on typesetting

Names in general are referred to by single quotations, like a folder name 'Myfolder'. Variables, cell arrays, structure names, fields of structures, objects,

[8]The number of gridpoints grows exponentially but the number of coefficients grows polynomially.

and properties of objects in Matlab are referred to in the text with the mathematical font of Latex, for example variable x, structure par or the field of a structure $par.alpha$. In general we use double letters in our code such as xx, because this makes it easier to find them. In this manual we generally refer to variables by the single letter (x). Strings in Matlab code will be referred to in Matlab typesetting, for example `'thisstring'`. Names referring to toolboxes, code, functions or scripts are in Typewriter font, as in `myfunction`, where the `.m` extension of functions and scripts will be omitted.

Scripts versus functions

There are two main differences between a function and a script in Matlab. The first is that one cannot define a subfunction in a script. The second is that a script will use the current workspace, while a function has its own workspace, which is empty unless input arguments are defined or global variables are used[9]. With a function one can evaluate the variables in the workspace by placing a breakpoint.

The toolbox consists of functions, and most examples are also functions, except the files `grid_example` and `main_stnd_rbc_proj`. For the latter example all subfunctions are stored in the subfolder 'STND_RBC_mod'. For the other examples all subfunctions are included in the main file.

1.6 Acknowledgments, sources, and license

Acknowledgments

We acknowledge that the Smolyak algorithm is implemented with the code provided by Rafa Valero[10]. The algorithm of this code is described in Judd et al. (2014). In addition, I thank Wouter den Haan, Joris de Wind and Petr Sedlacek for the courses on solving DSGE models at the Tinbergen Institute. I would also like to thank Alfred Maussner, Christopher Heiberger, and Daniel Fehrle at the University of Augsburg for various discussions on solving DSGE models, and writing this toolbox. Furthermore, I thank Alfred Maussner and Joris de Wind for providing their codes on projection methods.

Sources on projection methods

Good references on projection methods are Judd (1998), and Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016). The paper by Duineveld (2021) evaluates the most efficient algorithms of the `Promes` toolbox for three models. For practical guidelines on implementing projection methods

[9] Global variables are not recommended for Matlab.

[10] Rafa Valero (2021) Smolyak Anisotropic Grid, MATLAB Central File Exchange, Retrieved November 5, 2021 (https://www.mathworks.com/matlabcentral/fileexchange/50963-smolyak-anisotropic-grid).

Heer and Maussner (2009) and Wouter den Haan's material on his website www.wouterdenhaan.com are useful sources.

Copyright and license

Feedback

All feedback is more than welcome at s.a.duineveld@outlook.com.

2. Introduction to projection methods

Projection methods are used to approximate an unknown function. We will explain the basic principles with a simple 2-period life-cycle model, called the Simple Life Cycle model. This simple model has $C(x) = e^x$ as its solution[11]. The program that solves this model is the function `main_exp_proj` in the folder 'PROMES_v05.0.0/Examples'.

2.1 Simple Life Cycle model

The objective of the agent is to maximize utility derived from consumption C and C_2:

$$\max_{C,C_2} U(C) + U(C_2)$$

The optimization is subject to the budget constraint:

[11] Judd (1998) also approximates e^x, but he derives it from a differential equation and imposes an initial condition. We use a simple discrete time model. It should be emphasized that the model does not have the recursive structure of infinite horizon problems, which oversimplifies some aspects. The other examples in this manual have the recursive structure.

$$C_2 = 2e^x - C \tag{2.1}$$

where x is the capital stock at the beginning of the first period.

With the Constant Relative Risk Aversion (CRRA) utility function $U(C) = \frac{C_t^{1-\nu}-1}{1-\nu}$ and restricting $C \geq 0$, $C_2 \geq 0$, and $\nu > 0$ the First Order Conditions (FOC) are sufficient to define a unique solution[12]. The FOCs yield[13]:

$$\left(\frac{C}{C_2}\right)^{-\nu} = 1 \tag{2.2}$$

The system of equations consisting of (2.2) and (2.1) has the explicit solution $C(x) = e^x$.

2.2 Projection explained

In general projection methods approximate a policy variable as a function of the state variables. The state variables describe the current state of the economy, and are sufficient to determine the future behavior of the system[14]. In this example x is the state variable, and we choose consumption C as the policy or choice variable. The objective of projection methods is to approximate the exact policy function $C(x)$ with $\hat{C}(x;\theta)$, where θ is a vector of parameters that defines a polynomial or a spline.

The objective of projection methods is to find the policy function that solves the dynamic optimization problem. We need a residual function $R(x;\theta)$ that computes the errors in the dynamic equation (2.2) for a given approximation of the policy function. Given some policy $\hat{C}(x;\theta)$ we can compute C_2 with the budget restriction (2.1). The residuals are:

$$R(x;\theta) = \left(\frac{\hat{C}(x;\theta)}{2e^x - \hat{C}(x;\theta)}\right)^{-\nu} - 1 \tag{2.3}$$

The objective of projection methods is to find the approximation that minimizes the residual function R by setting θ.

[12] In most economic problems the model is restricted to be convex such that the First Order Conditions are sufficient to obtain a unique solution. Note however that dynamic models are usually saddle path stable, which means they have both a stable and an unstable solution.

[13] We do not use the standard formulation $C^{-\nu} = C_2^{-\nu}$, because in that case the relative errors are larger for higher levels of consumption. From the perspective of the agent it would be optimal to allow for larger errors when consumption is high, due to the risk aversion. However, we take the modeler's perspective and prefer more equally distributed errors. To analyze the effect on the approximation one can change the residual function to $C^{-\nu} = C_2^{-\nu}$, and use a second order polynomial for the approximation (by setting `POL.order = 2`). The plots will show that the errors will be large for high levels of consumption.

[14] See https://en.wikipedia.org/wiki/State_variable.

First we have to choose some interval $\underline{x} \leq x \leq \overline{x}$ of the state variable, where we want to approximation to be good. We select a set of q gridpoints on this interval:

$$x = \begin{bmatrix} x_1 & x_2 & \cdots & x_q \end{bmatrix}^\intercal \tag{2.4}$$

We call this vector the initial grid, and it will be assigned to *GRID.xx* by the function `prepgrid`. We want to emphasize that the modeler does not have to construct the grid. He/she only has to supply the inputs for the grid, consisting of the number of state variables n, the lower and upper bounds *lb* ($\underline{x}$) and *ub* ($\overline{x}$), and the algorithm *algo*. One can specify further options with the optional input argument *grid_spec*. The function `prepgrid` will construct the required fields in the structure *GRID* using the default settings with a simple call:

```
GRID = prepgrid(nn,lb,ub,algo);
```

After constructing the grid we need to calculate the residuals for which we use the model function `res_exp`. This subfunction takes the parameters *par*, the structure *GRID* and the structure with the policy function *POL* as inputs, and gives the residuals *RES* as output:

```
%% Residual function Simple Life Cycle model
function [RES]  = res_exp(par,GRID,POL)

%Initial grid of state variable:
xx = GRID.xx;

% Evaluate policy function,
% at the initial grid:
if ~(strcmp(POL.algo,'spl_tmi') || ...
  strcmp(POL.algo,'smol_tmi') || ...
  strcmp(POL.algo,'cheb_tmi'))

  % standard: log(C) from policy function
  CC     =      get_pol_var(POL,xx,GRID);
else
  % time iteration: solver directly sets C_i
  % (at gridpoint x_i)
  CC    = POL.YY;
end

% Budget constraint gives C2:
C2 = 2*exp(xx) - CC;

% Euler residuals:
RES = (CC./C2).^-par.nu - 1;
```

```
26
27 end
```

To compute the residuals we evaluate the approximation of the policy variable $\hat{C}(x)$ at the initial grid $GRID.xx$, for a given policy function in POL. There are two options for this. For all algorithms except those using Time Iteration (`'tmi'`) we call `get_pol_var` with the state vector x as input (Line 14). For the solution method `'tmi'` we directly evaluate the policy variable at the gridpoints, which is stored in $POL.YY$ by the toolbox (Line 18). After calculating C_2 from the budget constraint in Line 19 we can compute the residuals RES as in equation (2.3). The function `solve_proj` computes the policy function that minimizes these residuals. The function `solve_proj` will assign the policy function to the structure POL.

For all algorithms an initial guess for the policy function needs to be supplied to the solver `solve_proj`. The initial guess $Y0$ is the policy function at the initial grid. For our initial guess we use a third order Taylor series of the exact solution $C\left(x\right) = e^{x}$ around some point x^{*}:

$$C_0\left(x\right) = C\left(x^*\right) + \left(x - x^*\right)e^{x^*} + \frac{\left(x - x^*\right)^2}{2}e^{x^*} + \frac{\left(x - x^*\right)^3}{6}e^{x^*} \tag{2.5}$$

Note that in general we approximate an unknown function, which prevents us from directly computing a Taylor series approximation. However, we can obtain a good initial guess for most models using perturbation methods, which gives us a Taylor series approximation of a dynamic system of equations.

2.3 Spline methods

Two algorithms use a spline, `'spl_dir'` and `'spl_tmi'`. For both methods we use collocation, which solves the policy variable $\hat{C}_i = C\left(x_i\right)$ at the initial grid (2.4). Given $\hat{C}_i$ the residual R at gridpoint $i = 1, \ldots, q$ is:

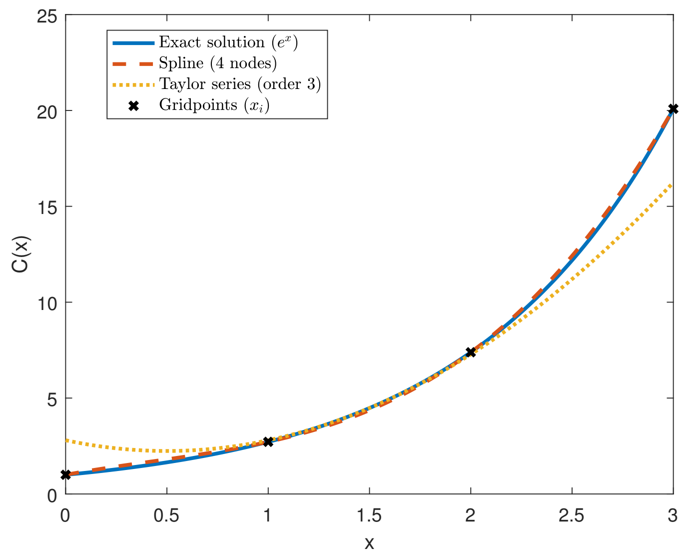
$$R\left(x_i, \hat{C}_i\right) = \left(\frac{\hat{C}_i}{2e^{x_i} - \hat{C}_i}\right)^{-\nu} - 1 \tag{2.6}$$

The objective (in vector notation) is:

$$R\left(x, C\right) = 0 \tag{2.7}$$

This is a system of q independent equations for q unknowns $\hat{C}_i$. We use $q = 4$ equidistant nodes in the interval $0 \leq x \leq 3$. This results in gridpoints $x = [0, 1, 2, 3]^{\mathsf{T}}$. We fit a spline through the solution at these gridpoints with Matlab's `griddedInterpolant`, and the interpolation method set to `'spline'`.

Figure 2.1: Spline approximation



The resulting policy function is plotted in Figure 2.1. We used 4 equidistant nodes and a cubic spline. The maximum error in $\hat{C}(x)$ is 0.26. The reason for this relatively large error is the small number of grid points. For comparison we have also included the third order Taylor series approximation (see equation (2.5)) around the point $x^* = 1.5$, which we used as an initial guess for the policy function. It is clear from the figure that the projection solution does well over the whole domain, while the Taylor series is inaccurate far away from the point $x^* = 1.5$. The maximum error for the third order Taylor series is 3.8.

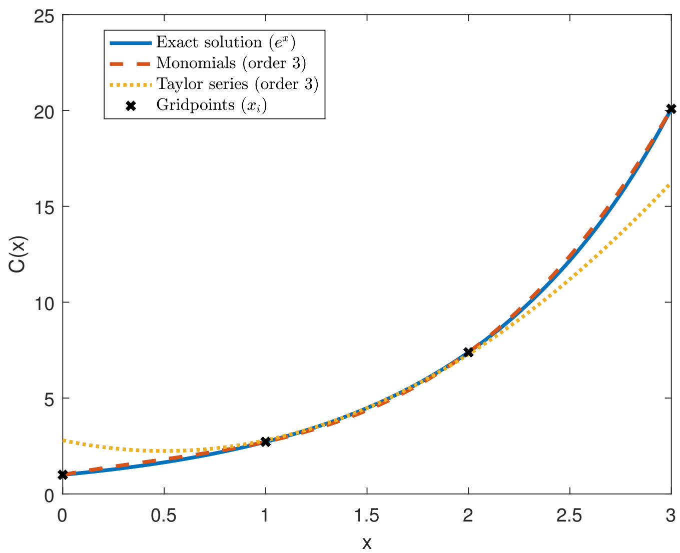
2.4 Monomial basis function

The algorithm `'mono_mse'` uses monomial basis functions to approximate the policy function. This algorithm is not recommended, but is a stepping stone to explaining approximation with Chebyshev polynomials. In our example we use the default order 3 monomial. The third order monomial basis functions consists of the terms $\Phi = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix}$. The third order polynomial approximation of $C(x)$ is:

$$\hat{C}(x;\theta) = \theta_1 + \theta_2 x + \theta_3 x^2 + \theta_4 x^3 \tag{2.8}$$

which means θ consists of $p = 4$ coefficients. We use $q = 4$ equidistant gridpoints on the interval $0 \leq x \leq 3$. As with the spline this results in the initial grid $x = [0, 1, 2, 3]^\intercal$, which the function `prepgrid` assigns to the field $GRID.xx$.

Figure 2.2: 3rd order monomial approximation



Having set the gridpoints we can evaluate the residual function (2.3) at those points using the approximation (2.8). The objective of the algorithm `'mono_mse'` is to minimize the sum of the squared residuals at the gridpoints:

$$\min_{\theta} \sum_{i=1}^{q} R\left(x_i;\theta\right)^2 \tag{2.9}$$

where q is the total number of gridpoints. The function `solve_proj` minimizes the sum of the squared errors by setting θ, which is assigned to $POL.theta$. The solver uses Matlab's `lsqnonlin` of the `Optimization Toolbox` to find the optimal values of θ. As the system is exactly identified the error at the gridpoints will be (close to) zero.

The resulting third order approximation is plotted in Figure 2.2. The maximum error in $\hat{C}\left(\theta\right)$ on the interval $0 \leq x \leq 3$ is 0.26. This error is of similar magnitude as the error in the spline approximation.

2.5 Chebyshev polynomials

The algorithms `'cheb_gal'`,`'cheb_tmi'`, and `'cheb_mse'` use complete Chebyshev polynomials. Chebyshev polynomials differ from monomial basis function in two aspects[15].

[15] Both aspects are taken care of by the toolbox, so the user does not need any knowledge of Chebyshev polynomials.

The first aspect is that for Chebyshev polynomials variables need to be scaled to the interval $[-1,1]$. The linear scaling down of variables from the interval $[\underline{x},\overline{x}]$ (lower and upper bound) to the interval $[-1,1]$ is:

$$\tilde{x}(x) = \frac{2x}{\overline{x}-\underline{x},} - \frac{\underline{x},+\overline{x}}{\overline{x}-\underline{x},} \tag{2.10}$$

where $\tilde{x}$ is the scaled down variable.

The second difference is that Chebyshev nodes are used instead of equidistant nodes. In our example the number of nodes is set at $q=4$. The Chebyshev nodes on the interval $[-1,1]$ are:

$$\tilde{x} = \begin{bmatrix} -0.924 & -0.383 & 0.383 & 0.924 \end{bmatrix}^{\mathsf{T}} \tag{2.11}$$

This vector is stored in $GRID.xx\_dw$ (see Section 10.13). The scaled up values (see Section 10.11) in the interval $[0,3]$ are:

$$x = \begin{bmatrix} 0.114 & 0.926 & 2.074 & 2.89 \end{bmatrix}^{\mathsf{T}}$$

which is assigned to $GRID.xx$. Note that these nodes are not linearly spaced.

As we use a third order approximation ($ord\_vec = 3$). The third order Chebyshev polynomial with one variable consists of $p=4$ terms:

$$\hat{C}(x;\theta) = \theta_1 + \theta_2\tilde{x}(x) + \theta_3\left(2\tilde{x}(x)^2 - 1\right) + \theta_4\left(4\tilde{x}(x)^3 - 3\tilde{x}(x)\right) \tag{2.12}$$

where $\tilde{x}(x)$ is the scaled down variable. We can write in short-hand notation:

$$\hat{C}(x;\theta) = \sum_{j=1}^{p} \theta_j \Omega_j(\tilde{x}(x)) \tag{2.13}$$

Alternatively we can write the polynomial as a matrix of polynomial terms:

$$\Omega(\tilde{x}(x)) = \begin{bmatrix} 1 & \tilde{x}(x) & 2\tilde{x}(x)^2 - 1 & 4\tilde{x}(x)^3 & -3\tilde{x}(x) \end{bmatrix}$$

which is a $q \times p$ matrix, where q is the number of datapoints. This allows us to formulate (2.13) using matrix multiplication:

$$\hat{C}(x;\theta) = \Omega(\tilde{x}(x))\,\theta \tag{2.14}$$

There are three algorithms with complete Chebyshev polynomials. The first is `'cheb_mse'`, which has the same objective (2.9) as with monomials. The second algorithm `'cheb_gal'` uses Galerkin projection. This means each coefficient θ_j for $j = 1, \dots, p$ is set such that the residuals (2.3) are orthogonal to the corresponding polynomial term $\Omega_j(\tilde{x}(x))$ in equation (2.13). Using matrix notation we have to solve a system of equations:

$$0 = R(x;\theta)^{\intercal}\,\Omega(\tilde{x}(x)) \tag{2.15}$$

where $R(x;\theta)$ is the $q\times 1$ residual vector at the gridpoints. This gives us a system of p equations in p unknowns. An alternative formulation of the objective for each θ_j is to solve the equation:

$$0 = \sum_{i=1}^{q} R(x_i;\theta)\,\Omega_j(\tilde{x}(x_i)) \tag{2.16}$$

The third algorithm is `'cheb_tmi'`. For this algorithm we use collocation, meaning that we first solve the policy variable at the gridpoints $\hat{C}_i = \hat{C}(x_i)$ for $i = 1, \dots, q$ as we did in for splines in Section 2.3, using residual function (2.6) and objective (2.7).

The second step of `'cheb_tmi'` is to fit the complete polynomial through the solution at the gridpoints. This is done by solving the linear system of equations:

$$\theta = \Omega(\tilde{x}(x))^{-1}\,\hat{C} \tag{2.17}$$

The result for the third order approximation with `'cheb_gal'` is plotted in Figure 2.3. The maximum error in $\hat{C}(\theta)$ on the interval $0 \leq x \leq 3$ is 0.18. As before the number of parameters θ and the number of nodes are the same ($p = q = 4$) resulting in (close to) zero errors at the gridpoints.

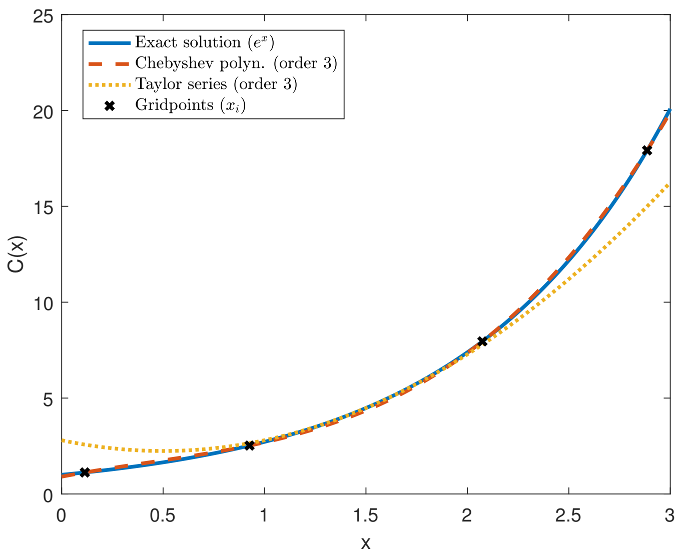
2.6 Smolyak algorithm

The Smolyak algorithm uses a sparse grid and sparse polynomial to approximate the policy function. The grid is constructed using nested sets with a special format. For this reason the only grid parameter than can be set is μ, which determines the total number of gridpoints, which is equal to the number of coefficients of the polynomial.

In the one dimension case the number of nodes is $q_\mu = 2^\mu + 1$, and the degree of the Chebyshev polynomial is 2^μ. The nodes correspond to the extrema of a univariate Chebyshev polynomial. For given μ with $i = 1, \dots, q_\mu$ the gridpoints are (Judd et al., 2014):

$$x_i = \begin{cases} 0 & \text{for } \mu = 0 \\ -\cos\left(\frac{i-1}{q_\mu - 1}\pi\right) & \text{for } \mu > 0 \end{cases}$$

Figure 2.3: 3rd order Chebyshev approximation



We set $\mu = 2$ which results in $m = 5$ gridpoints, and an order 4 polynomial (with $p = 5$ coefficients). The nodes are:

$$x = \begin{bmatrix} 0 & -1 & 1 & -0.707 & 0.707 \end{bmatrix}$$

The $m \times p$ matrix with polynomial terms is:

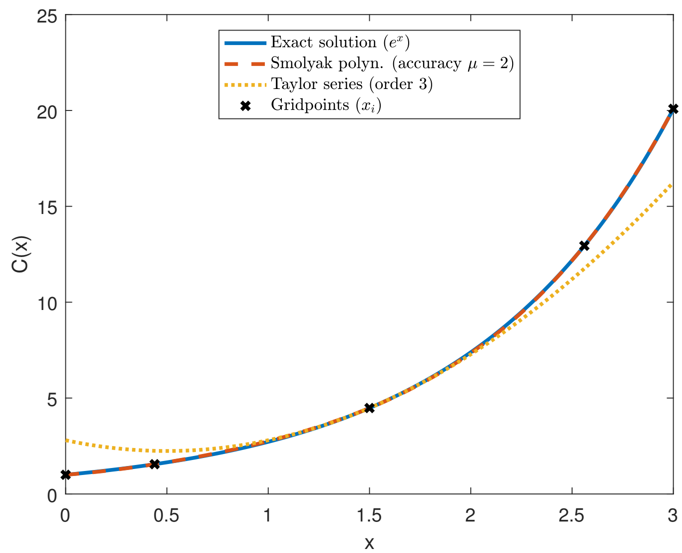
$$\Omega\left(\tilde{x}\left(x\right)\right) = \begin{bmatrix} 1 & \tilde{x}\left(x\right) & 2\tilde{x}\left(x\right)^2 - 1 & 4\tilde{x}\left(x\right)^3 - 3\tilde{x}\left(x\right) & 8\tilde{x}\left(x\right)^4 - 8\tilde{x}\left(x\right)^2 + 1 \end{bmatrix}$$

With `'smol_tmi'` we use collocation. We solve for the policy variable $\hat{C}_i$ at the gridpoints i using residual function (2.6) and objective (2.7) as with the spline algorithms and `'cheb_tmi'`. Given the solution $\hat{C}$ at the gridpoints the coefficients θ can be determined with equation (2.17) using matrix inversion. With `'smol_dir'` we directly solve for the coefficients θ using the objective (in vector notation):

$$R\left(x; \theta\right) = 0$$

The results are plotted in Figure 2.4. The maximum absolute error is 0.04 for an order four polynomial ($\mu = 2$). This error is considerably smaller than with third order polynomials.

Figure 2.4: Smolyak approximation



3. Basic Procedure

To set up, solve and evaluate a model with the `Promes` toolbox one typically needs to take 6 steps:

1. Set the parameters and solve the steady state;
2. Set the grid parameters, choose the algorithm, and construct the grid using the function `prepgrid`;
3. Create the model function, and function handle;
4. Make an initial guess for the policy function;
5. Solve the model, using the function `solve_proj;`
6. Evaluate the solution using the function `get_pol_var`.

To explain the above steps we use a very simple deterministic macroeconomic model, which we solve with the default settings of each algorithm. We describe the model in Section 3.1, define the approximated solution in Section 3.2, and the program code in Section 3.3. The program is the function `main_det_bm_proj`, which can be found in the folder 'PROMES_v05.0.0/Examples'.

3.1 Deterministic Brock-Mirman model

The Brock-Mirman model that we use is a special case of the deterministic, representative agent growth problem described by Judd (1998)[16]. The model is discussed in more detail in Chapter 11. The deterministic version of the model has only one state variable, which is capital K_t. The advantage of this model is that there exists an analytical solution to which we can compare the numerical solution.

The representative agent maximizes his/her discounted utility:

$$\max \sum_{t=1}^{\infty} \beta^{t-1} \log (C_t)$$

subject to:

$$K_{t+1} + C_t = K_t^{\alpha} \tag{3.1}$$

where C_t is consumption in period t, β is the discount factor, K_t is the capital stock at the beginning of the period, and K_t^{α} is the production function. Applying the recursive formulation and taking the first order conditions with respect to K_{t+1} and C_t we obtain the Euler equation:

$$\frac{1}{C_t} = \beta \frac{1}{C_{t+1}} \alpha K_{t+1}^{\alpha-1} \tag{3.2}$$

3.2 Approximation of the policy function

The objective is to find the policy function for consumption as a function of the state variable capital. The two equations (3.1) and (3.2) are sufficient to solve the model with projection methods.

We approximate the policy function for consumption as a function of capital $C_t = C(K_t)$. We use either a polynomial or a spline $\hat{C}_t = \hat{C}(K_t; \theta)$[17], where θ is a vector of coefficients determining the spline or polynomial.. To indicate the parameterization of the j-th iteration we write θ^j. To obtain the solution we minimize the residuals of the Euler equation (3.2) for a finite number of gridpoints.

To calculate the Euler residuals at these gridpoints we first calculate next period's capital stock using (3.1):

$$\hat{K}_{t+1} = K_t^{\alpha} - \hat{C}(K_t; \theta^q) \tag{3.3}$$

[16] Example in 16.4 starting on page 549.

[17] In practice we use both consumption and capital in logs $\hat{c}_t = \hat{c}(k_t; \theta)$, where smaller cases indicate logs.

where we used policy θ^q for period t choices. Next we substitute K_{t+1} into the Euler equation (3.2) and compute period $t+1$ consumption $\hat{C}\left(\hat{K}_{t+1};\theta^p\right)$.

Finally we multiply both sides of the equation with $\hat{C}\left(K_t;\theta^q\right)$ to normalize the Euler residuals[18]. The Euler residuals R are:

$$R\left(K_t;\theta\right) = \beta \frac{\hat{C}\left(K_t;\theta^q\right)}{\hat{C}\left(\hat{K}_{t+1};\theta^p\right)} \alpha \hat{K}_{t+1}^{\alpha-1} - 1 \tag{3.4}$$

Note that we use parameterization θ^q for period t choices, and θ^p for period $t+1$ choices. This distinction is only relevant for algorithms using Time Iteration (`'tmi'`). For the other algorithms these two parameterizations are the same, meaning $\theta^q = \theta^p$.

3.3 Basic Procedure

To solve a model with the toolbox one needs to go through the 6 steps described above. In this section we describe these steps in the example code `main_det_bm_proj` in the folder 'PROMES_v05.0.0/Examples'. This file is a function, which includes all the subfunctions of the model. These subfunctions calculate the Euler residual (`det_bm_res`), the steady state (`det_bm_ss`) and auxiliary variables (`det_bm_aux`). Each of the 6 steps of the example program are described in the following Subsections.

Step 0: Matlab settings

Before running the example we set some general Matlab settings, and add the relevant folders of the toolbox to the searchpath.

Step 1: Initial block

The initial block consists of two substeps, which are shown in Listing 3.1. In Step 1.A we set the parameters of the model, and in Step 1.B we solve the steady state[19].

Listing 3.1: Step 1 `main_det_bm_proj`

```
function main_det_bm_proj(sol_meth)
% Function gives projection solution for
% deterministic version of Brock-Mirman model,
% using default settings of Promes toolbox

```

[18]This scaling ensures that errors are more equally distributed. If we would not normalize the Euler residuals the errors in consumption would be larger for high levels of consumption, due to the risk aversion of the agent.

[19]See Chapter 11 for the derivation of the steady state.

```
%% STEP 1: INITIAL BLOCK
% STEP 1.A: Set parameters of the model
par.alpha        = 0.33; % production: K^alpha
par.beta         = 0.96; % discount factor

% STEP 1.B: Solve steady state
SS               = det_bm_ss(par);
```

Step 2: Construct the grid

The construction of the grid is shown in Listing 3.2. In this step one needs to set the parameters of the grid, which is explained in more detail in Chapter 10. The first three inputs are set as fields in the temporary structure *gin*. These are the number of state variables n, and the lower bound and upper bound of the state variables, *lb* and *ub* respectively. When constructing the grid with `prepgrid` one also needs to set the algorithm in *POL.algo*, because the algorithm determines the type of grid that is constructed. We use the default settings for each algorithm, but one can specify algorithm specific parameters in the optional field *grid_spec* (see for example Section 4.4).

Listing 3.2: Step 2 `main_det_bm_proj`

```
%% STEP 2: Construct the grid
% STEP 2.A: Set parameters & bounds of grid, in log(K)
gin.nn           = 1;% number of state variables

% Boundaries of grid at steady state +/- 20%:
gin.lb(1)        = -0.2 + log(SS.Kss);% lower bound
gin.ub(1)        = 0.2 + log(SS.Kss);% upper bound

% STEP 2.B: Choose algorithm:
% 'cheb_gal','cheb_tmi','cheb_mse'
% 'spl_tmi','spl_dir'
% 'smol_tmi','smol_dir'
% 'mono_mse'
POL.algo    = 'cheb_gal';

% STEP 2.C: Construct the grid
% with default settings
GRID    = prepgrid(gin.nn,gin.lb,gin.ub,POL.algo);
```

The function `prepgrid` will assign all the necessary properties to the structure *GRID*, given the algorithm. The structure *GRID* includes the initial grid xx, which is a $m \times n$ matrix, where every column is a state variable (or dimension), and every row a unique gridpoint. The explanation of the grid structure is found in Chapter 10 with an example in Section 10.4. It should be emphasized that the structure $GRID$ is specific for each algorithm. For example, a spline uses equidistant nodes, while complete Chebyshev polynomials use Chebyshev nodes. For polynomials the structure $GRID$ also includes the polynomial of the initial grid.

Step 3: Model function

The model function should calculate the Euler residuals, given the initial grid and the policy function. The model is shown in Listing 3.3. The file takes the structure with the parameters par, the grid structure $GRID$, and the policy structure POL as input arguments. Note that the modeler does not have to specify the policy function in the model file, because the function `solve_proj` will assign the appropriate policy function (a spline or polynomial) to the structure POL. More details on constructing a model file can be found in Chapter 8.

First the model function retrieves the state variable capital in logs k_t, which is LK in the code. Since there is only one state variable, it is the first and only column of the initial grid $GRID.xx$, as shown in Line 5. Next we evaluate the policy function given the state variable: $\hat{c}_t = c\,(k_t;\theta)$. For all algorithms except those using Time Iteration (`'tmi'`) the policy function is evaluated by calling the function `get_pol_var` as shown in Line 13 of Listing 3.3. This call takes the policy function POL, the state variable in period t (LK) and the structure $GRID$ as inputs. For the solution method `'tmi'` the solver directly sets the policy function c_t^i at all gridpoints i. These values are assigned to $POL.YY$ as shown in Line 18.

Next we calculate k_{t+1}, given k_t and $\hat{c}_t$ as in equation (3.3) (Line 22), and finally we need to calculate $\hat{c}_{t+1} = c\,(k_{t+1};\theta)$. For all algorithms except those with `'tmi'` we evaluate the policy function using the function `get_pol_var` with k_{t+1} as input argument (Line 29). For the solution method `'tmi'` we need to use the policy function of the old iteration θ^p. To ensure the old policy function is used for next period's choices the input argument *spec_opt* has to be set to `'old_pol'` (Line 33 and 34). With $\hat{c}_t$, $\hat{c}_{t+1}$ and $\hat{k}_{t+1}$ known we can compute the vector of Euler residuals as in (3.4) (Lines 38, 41, and 44).

Listing 3.3: Model function `main_det_bm_proj`

```
1  %% Deterministic B-M model file, or residual function:
2  function [RES] = det_bm_res(par,GRID,POL)
3
4  % Initial grid is stored in GRID.xx:
5  LK = GRID.xx;%log(K_t)
6
7  % Evaluate the policy function log(C):
8  if ~(strcmp(POL.algo,'spl_tmi') || ...
9    strcmp(POL.algo,'smol_tmi') || ...
10   strcmp(POL.algo,'cheb_tmi'))
11
```

```
  % standard: log(C) from policy function
  LC = get_pol_var(POL,LK,GRID);

else
  %tmi: solver directly sets log(C)
  % at grid points
  LC = POL.YY;
end

% Capital in next period (log):
LK_n = log( exp(par.alpha*LK) - exp(LC) );

% log(C_t+1) from policy function, given log(K_t+1):
if ~(strcmp(POL.algo,'spl_tmi') || ...
  strcmp(POL.algo,'smol_tmi') || ...
  strcmp(POL.algo,'cheb_tmi'))

  LC_n = get_pol_var(POL,LK_n,GRID);

else% for 'tmi':
  % use old policy function
  spec_opt_next = 'old_pol';
  LC_n = get_pol_var(POL,LK_n,GRID,[],spec_opt_next);
end

% log(RR_t+1): marginal prod. of capital (logs)
LR_n = log(par.alpha) + (par.alpha-1)*LK_n;

% RHS of Euler equation:
RHS          = par.beta * exp(-LC_n) .* exp(LR_n);

% Euler residual (scaled by C^-1):
RES          = exp(LC).*RHS - 1;

end
```

Finally, we need to create a function handle to the model function, which takes the structure *POL* as input argument:

Listing 3.4: Create function handle

```
%% STEP 3: Handle for objective function
% (ie. the model file)
fun_res      = @(POL)det_bm_res(par,GRID,POL);
```

This function handle is used as input for the solver `solve_proj`. The function `solve_proj` will assign the policy function to the structure *POL*.

Step 4: Initial guess

The initial guess $Y0$ for the policy function should give the value of the policy variable at the initial grid. This initial guess is an input argument for the solver `solve_proj`. For this simple model all algorithms will converge to the correct solution from a relatively poor initial guess. As the initial guess we use steady state consumption plus a small linear term in capital:

```
%% STEP 4: Initial guess policy function of log(c)
% steady state consumption +  small linear term in [
    log(K)-log(Kss)]:

Y0      = log(SS.Css) + 0.01*(GRID.xx-log(SS.Kss));
```

For more complex models one could use the perturbation solution as initial guess. Initial guesses are discussed in more detail in Section 4.6.

Step 5: Solve the model

To solve the model the function `solve_proj` is called. It takes as inputs the grid structure $GRID$, the policy structure POL, the function handle to the residual function fun_res, and the initial guess $Y0$.

```
%%% STEP 5: Solve the model
POL          = solve_proj(GRID,POL,fun_res,Y0);
clear Y0;
```

The structure POL at this points only needs to contains the algorithm in the field *POL.algo*. The function `solve_proj` will assign the appropriate policy function (spline or polynomial) to the structure POL. For the polynomial algorithms this will be the coefficients in the field *POL.theta*. For the spline algorithms the policy function is assigned to the field *POL.pp_y*. The spline *pp_y* is constructed with Matlab's `griddedInterpolant`.

Step 6: Evaluate the solution

To evaluate the solution we call the function `get_pol_var` with inputs POL, the state variables, and the structure $GRID$. In the example file we evaluate the policy at the initial grid:

```
%% STEP 6: Evaluate policy function:
LK = GRID.xx; % = initial grid
LC = get_pol_var(POL,LK,GRID);
```

In Section 4.8 we give an example of a simulation where `get_pol_var` is used to evaluate the policy function.

In our example we also plot the policy function. In addition we plot the error, which is the difference between the numerical solution $\hat{c}(k_t;\theta)$ and analytical solution $c(k_t)$. The maximum absolute errors are small, and range between the order 10^{-7} to 10^{-13} with the default settings. Smaller errors can be achieved by changing the stopping criteria as explained in Section 4.7.

4. Detailed Procedure

In this Chapter we discuss the procedure to solve a model in more detail. We use the example of a standard RBC model with stochastic shocks in Total Factor Productivity, which we call the 'Standard RBC example'. We first briefly describe the model in Section 4.1. The main program is the script `main_stnd_rbc_proj` in the folder 'PROMES_v05.0.0/Examples'. That script solves the model, and simulates time series. All model specific function are found in the subfolder 'STND_RBC_mod'. Each step in the process is described in more detail in Sections 4.3 to 4.8.

4.1 Standard RBC model

The model consists of two state variables, capital K_t and Total Factor Productivity (TFP) Z_t. Capital is determined endogenously, while TFP follows a stochastic process. We choose consumption C_t as policy variable, which we approximate as a function of the state variables. The model is captured by four equations[20]:

$$C_t + K_{t+1} = Z_t K_t^{\alpha} H_t^{1-\alpha} + (1-\delta) K_t \tag{4.1}$$

$$\chi H_t^{\frac{1}{\eta}} = C_t^{-\nu} Z_t (1-\alpha) K_t^{\alpha} H_t^{-\alpha} \tag{4.2}$$

$$C_t^{-\nu} = \beta E_t \left\{ C_{t+1}^{-\nu} \left[Z_{t+1} \alpha K_{t+1}^{\alpha-1} H_{t+1}^{1-\alpha} + 1 - \delta \right] \right\} \tag{4.3}$$

$$z_t = \rho_z z_{t-1} + \sigma_z \epsilon_t \tag{4.4}$$

where smaller cases indicate logs, ie. $z_t = \log(Z_t)$. The autocorrelation coefficient is ρ_z , and the shocks are scaled by σ_z. The shocks are standard normally distributed, ie. $\epsilon_t \sim \mathcal{N}(0,1)$.

4.2 Step 0: Matlab Settings

The only function of Step 0 is to prepare Matlab for running the script. It includes clearing all variables in the workspace, and adding folders to the searchpath.

[20] See Chapter 12 for details.

Clearing and set breakpoint

When using a script you typically want to clear all variables from the workspace using `clearvars`, which is not needed if a function is used. In the initial block of our program we close all figures (`close all`), clear the command prompt (`clc`), and ensure that we can access all local variables at the time an error occurs by setting `dbstop if error`.

Adding folders

Next, we need to add the folders 'PROMES_v05.0.0', and its subfolders 'grid_subfun', and 'smolyak_subfun' to the searchpath. In addition we add the folder 'TOOLS' and its subfolders 'CSD_v02.4.0' and 'CSD_v02.4.0\subfun' to the searchpath. We need the folder 'TOOLS' for the calculation of the Gauss-Hermite nodes with the function `hernodes`. We need the 'CSD_v02.4.0' folders to obtain the perturbation solution with the `CSD` toolbox.

Our model specific files are stored in the folder 'PROMES_v05.0.0\Examples\STND_RBC_mod'. These files include the model function `STND_RBC_proj`. The other relevant subfunctions of this model are `stnd_rbc_ss` which calculates the steady state, `stnd_rbc_aux` which calculates auxiliary variables of the model, and `stnd_rbc_sim` which is used to run simulations.

In our main program file `main_stnd_rbc_proj` Step 0 is:

```
% Solves standard RBC model with projection
% for a single variable policy function

%% STEP 0: Matlab settings
clearvars;
close all; %close all figures
clc; %clear command prompt
dbstop if error;%acces workspace if error

restoredefaultpath;
clear RESTOREDEFAULTPATH_EXECUTED;

% Add relevant folders of Promes toolbox:
addpath ('..');
addpath ('..\grid_subfun');
addpath ('..\smolyak_subfun');

% Add relevant folders of TOOLS
addpath ('..\..\TOOLS');
addpath ('..\..\TOOLS\CSD_v02.4.0');
addpath ('..\..\TOOLS\CSD_v02.4.0\subfun');
```

4.3 Step 1: Initial Block

The Initial Block consists of two parts:

A. Set parameters

B. Solve steady state

In our function `main_stnd_rbc_proj` we store all parameters in a structure called *par*. The steady state values are stored in a structure SS. The structure SS is used for the determination of the lower and upper bound of the capital stock, the initialization of the policy function, and the function that plots the policy function.

Step 1.A: Set parameters

We assign all the parameters of the model to the structure *par*. Using this structure as input argument of a function gives access to all parameters of the model. In our example we will use Gauss-Hermite quadrature (see the chapter on Numerical Integration in Judd, 1998), and we add the Gauss-Hermite nodes (*par.her.xi*) and its weights (*par.her.wi*) to the parameters.

In our example script `main_stnd_rbc_proj` Step 1.A is:

```
% STEP 1.A: Set parameters of the model
par.alpha        = 0.36;          %capital share income
par.beta         = 0.985;         % discount factor
par.delta        = 0.025;         % deprec. of capital
par.nu           = 2;             % risk aversion
par.eta          = 4;             % el. of lab. supply
par.chi          = 1;             % scalar disut. work

par.rho_z        = 0.95; % autocorr. coeff. TFP
par.sigma_z      = 0.01; % standard dev. shocks in TFP

par.her.gh_nod   = 5;% number of Gauss-Hermite nodes
[par.her.xi,par.her.wi] = hernodes(par.her.gh_nod);
% xi are roots, wi are weights
```

Step 1.B: Solve steady state

We want a good approximation of the policy function on a particular interval of the state variables, which is usually centered around the steady state. For this reason we compute the steady state. For our Standard RBC example we created the function `stnd_rbc_ss` that calculates the steady state analytically (see Section 12.3). This function takes the parameters and the steady state Total Factor Productivity ($Z_{ss} = 1$) as inputs. In the script `main_stnd_rbc_proj` our call to the steady state function is:

```
%% STEP 1.B: Solve steady state
SS                = stnd_rbc_ss(par,1);
% the 1 is steady state TFP
```

4.4 Step 2: Construct the grid

The construction of the grid is done using the following substeps:

A. Set basic grid parameters

B. Set algorithm & algorithm specific parameters

C. Construct grid

The grid parameters, and also the grid itself will be stored in the structure *GRID*. In Step 2.A three basic parameters of the grid need to be set. The algorithm has to be assigned to the structure *POL* in Step 2.B, before constructing the grid. In Step 2.B algorithm specific grid parameters can be set, such as the order of the polynomial or the number of nodes.

These inputs are fed into the function `prepgrid` (Step 2.C). This function constructs the structure *GRID*, and assigns all the required fields of the selected algorithm. More details on how the grid is constructed can be found in Chapter 10. That chapter also includes an example code, `grid_example`, that demonstrates the construction of a grid.

Step 2.A: Set basic grid parameters

The `prepgrid` function has the following inputs:

- *nn*: number of variables in the grid (scalar);
- *lb*: vector of lower bounds in each dimension (1 x *nn* vector);
- *ub*: vector of upper bounds in each dimension (1 x *nn* vector);
- *algo*: the algorithm, which is discussed in Step 2.B.
- *algo_spec* (optional): algorithm specific grid parameters

The first four inputs are necessary to create the structure *GRID*. In our model we have two state variables, capital (K) and Total Factor Productivity (Z). For the construction of the grid we use the logarithm of both variables. We set the lower and upper bound for each state variable symmetrically around this steady state. For capital we use the steady state in logs ± 0.1275, and for Total Factor Productivity we use 2.6 standard deviations around steady state.

```
%% STEP 2: Construct the grid
%Step 2.A: Initialize the grid
gin.nn       = 2;%number of state variables (K,Z)

%Set lower and upper bound for capital:
gin.lk_dev   = 0.1275;% deviation from kss
gin.lb(1)    = -gin.lk_dev + log(SS.kss);
gin.ub(1)    =  gin.lk_dev + log(SS.kss);

% Set lower and upper bound for log(z)
gin.lz_fac   = 2.6;%in multiple of stnd. deviation
gin.lb(2)    = -gin.lz_fac*...
        sqrt( par.sigma_z^2 / (1-par.rho_z^2) );
gin.ub(2)    =  gin.lz_fac*...
        sqrt( par.sigma_z^2 / (1-par.rho_z^2) );
```

Step 2.B: Set algorithm

The algorithm *algo* is set as a field in the structure *POL*. One can choose from eight algorithms to approximate the policy function. The algorithms are discussed in more detail in Chapter 7. The eight algorithms are based on four types of basis functions. These are splines, complete Chebyshev polynomials, Smolyak polynomials, and complete polynomials based on monomials.

- Splines: splines are used with the collocation projection condition. The objective is to set the residuals at each gridpoint to zero. The spline is defined by Matlab's `griddedInterpolant`. The interpolation method can be specified in the field *POL.meth_spl*. The two available solution methods are Time Iteration (`'spl_tmi'`) and Direct Computation (`'spl_dir'`);

- Complete Chebyshev polynomials: three algorithms are available. The first uses Galerkin's method (`'cheb_gal'`), which sets the residuals orthogonal to the polynomial terms. The second uses Time Iteration (`'cheb_tmi'`) to solve the policy variable at the gridpoints, and fits a spline to these points. The third uses Minimization of the Squared Errors (`'cheb_mse'`) to minimize the errors at the gridpoints;

- Smolyak polynomial: Smolyak's algorithm constructs a sparse grid, and sparse Chebyshev polynomial. The algorithm ensures that the number of nodes and coefficients grows only polynomially in the number of state variables, while the number of nodes grows exponentially for the other methods. The two available solution methods are Time Iteration (`'smol_tmi'`) and Direct Computation (`'smol_dir'`);

- Monomial basis functions ('mono_mse'): the coefficients of a complete polynomial are determined by Minimization of the Squared Errors of the residual function at the gridpoints.

In general 'spl_tmi' is a very robust choice. For simple, well-behaved models 'cheb_gal' is a good alternative. For a large number of state variables 'smol_tmi' is advised. In Section 7 under Algorithms more details are discussed.

The input *algo_spec* of the function `prepgrid` can be used to set algorithm specific grid parameters. These parameters are:

- *qq* (for all basis functions except Smolyak's algorithm): number of nodes in each dimension ($1 \times nn$ vector). This parameter allows for asymmetric grids with a different amount of gridpoints in each dimension;
- *ord_vec* (Chebyshev & monomials only): the order of the polynomial in each dimension ($1 \times nn$ vector). This parameter allows for asymmetric polynomials;
- *mu_vec* (Smolyak's algorithm only): the accuracy of the approximation in each dimension ($1 \times nn$ vector). This parameter allows for asymmetric grids and polynomials.

Example Standard RBC model

In our example we set *POL.algo* to 'cheb_gal', but this can be set to any of the other algorithms. In our example `main_stnd_rbc_proj` the code of Step 2.B is:

```
% STEP 2.B: Set algorithm:
POL.algo     = 'cheb_gal';
```

We set the following algorithm specific grid parameters:

```
% (OPTIONAL) Algorithm specific parameters
%CHEYBSHEV:
if strncmp(POL.algo,'cheb',4)
  %order of polyn. in each dim.
  algo_spec.ord_vec    = 5*ones(1,gin.nn);
  %# nodes in each dim.:
  algo_spec.qq         = POL.meth_spec.ord_vec+1;

%SPLINES:
elseif strncmp(POL.algo,'spl',3)
  %# nodes in each dim.:
  algo_spec.qq         = 7*ones(1,gin.nn);

%SMOLYAK:
elseif strncmp(POL.algo,'smol',4)
  %accuracy param. in each dim.:
```

```
  algo_spec.mu_vec     = 3*ones(1,gin.nn);

%MONOMIALS:
elseif strncmp(POL.algo,'mono',4)
  %order of polyn. in each dim.:
  algo_spec.ord_vec    = 3*ones(1,gin.nn);
  %# nodes in each dim.:
  algo_spec.qq         = POL.meth_spec.ord_vec+1;

else
  error('Invalid algo');
end
```

Step 2.C: Construct the grid

The construction of the grid is carried out by the function `prepgrid`, which requires at least four inputs. This function and all its subfunctions are explained in Chapter 10. Our call of the function `prepgrid` is:

```
% STEP 2.C: Construct the grid
GRID = prepgrid(gin.nn,gin.lb,gin.ub,POL.algo,
    algo_spec);
```

The function `prepgrid` will assign all the necessary fields to the structure $GRID$. The most important of these is the initial grid $GRID.xx$, which is an $m \times n$ matrix, where m is the total number of gridpoints, and n the number of state variables. Each row is a unique gridpoint, and every column represents a state variable. The first column $GRID.xx\,(:,1)$ contains capital in logs, and the second $GRID.xx\,(:,2)$ Total Factor productivity in logs. For the polynomial algorithms also the polynomial of the initial grid will be assigned to either XX_poly (for monomials) or in scaled down variables to XX_poly_dw (for Chebyshev and Smolyak algorithms).

It should be noted that the grid structure is specific to the algorithm. If one changes the algorithm a new grid structure should be constructed with `prepgrid`[21].

4.5 Step 3: Model function and handle

Step 3 consists of two parts:

A. Program model function

B. Create handle to model function

[21]To be more more precise, the grid is specific to the basis function, which is either a spline, a Chebyshev polynomial, a Smolyak polynomial, or based on monomials.

The creation of the model function is the crucial step. The model function should return a column vector with residuals as output. To solve the model with the solver `solve_proj` a function handle of this model function has to be created. This function handle should take the structure *POL* as input.

Step 3A: Program model function

The model function, which has to be programmed by the modeler, should calculate the Euler residuals at each gridpoint, given the policy function. It should be noted that the modeler does not have to specify the policy function. The toolbox will assign the policy function to *POL*. The model file should at least take the grid structure *GRID*, and the structure with the policy *POL* as inputs. Other inputs are also allowed. In our example we use the structure with the parameters *par* as additional input.

For the Standard RBC model the model function is `STND_RBC_proj`, which is shown in Listing 4.1. This file is found in the subfolder 'STND_RBC_mod'. The output are the residuals of the Euler equation (4.3), which are computed for a given policy function and initial grid. The code is explained further below.

Listing 4.1: Model function `STND_RBC_proj`

```
function [RES] = STND_RBC_proj(par,GRID,POL)
% Calculates Euler residuals for standard RBC model

LK = GRID.xx(:,1);%first state variable, log(K_t)
LZ = GRID.xx(:,2);%second state variable, log(Z_t)

%policy variable, log(C_t):
if ~(strcmp(POL.algo,'spl_tmi') || ...
  strcmp(POL.algo,'smol_tmi') ||...
  strcmp(POL.algo,'cheb_tmi') )

  %use initial grid for polynomials
  % (or ignore for spl_dir)
  spec_opt   = 'ini_grid';
  LC   = get_pol_var(POL,[LK,LZ],GRID,[],spec_opt);

else%for 'spl_tmi','cheb_tmi','smol_tmi':

  %LC is set directly for Time Iteration
  LC   = POL.YY;
end

%Capital in next period:
LK_n     = stnd_rbc_aux(par,LK,LZ,LC);


```

```
if strcmp(POL.algo,'spl_tmi') || ...
  strcmp(POL.algo,'smol_tmi') ||...
  strcmp(POL.algo,'cheb_tmi')

  % use old policy function in t+1
  % for Time Iteration
  % (pp_y_old or theta_old)
  spec_opt_next = 'old_pol';
else
  spec_opt_next = [];
end

%Allocate empty matrix for RHS of Euler equation:
rhs_l    = NaN(size(LK,1),par.her.gh_nod);

for ll = 1:par.her.gh_nod
  % Shock to TFP (using Gauss-Hermite nodes):
  EPS_n        = sqrt(2)*par.her.xi(ll);

  %log(Z_t+1):
  LZ_n         = par.rho_z*LZ + par.sigma_z*EPS_n;

  %log(C_t+1)
  LC_n         = get_pol_var(POL,[LK_n,LZ_n],GRID,[],
     spec_opt_next);

  %log(MPK_t+1) (marginal prod. of capital)
  [~,LMPK_n]     = stnd_rbc_aux(par,LK_n,LZ_n,LC_n);

  % RHS of Euler equation,
  % weighted by Gauss-Hermite weights
  rhs_l(:,ll) = par.her.wi(ll)/sqrt(pi)*par.beta*...
        exp(-par.nu*LC_n) .* ...
        (exp(LMPK_n)+1 - par.delta);
end

%Right hand side of Euler equation:
RHS = sum(rhs_l,2);

% Euler residuals (scaled by C^-nu):
RES      =  RHS./exp(-par.nu *LC) - 1;

end
```

Evaluating the policy function

In our code the policy and state variables are all defined in logs. The policy function is $\hat{c} = \hat{c}(k, z; \theta)$, where small cases indicate logs. For simplicity we ignore the log transformation, and write $\hat{C}(K, Z; \theta) = \exp[\hat{c}(k, z; \theta)]$.

To evaluate the policy function we differentiate between the solution method Time Iteration (`'tmi'`) and the other solution methods (see Chapter 8 for more details). In period t the policy variable is:

$$\hat{C}_t = \begin{cases} \hat{C}_t^j & \text{if sol. meth. is 'tmi'} \\ \hat{C}(K_t, Z_t; \theta) & \text{else} \end{cases} \tag{4.5}$$

where $\hat{C}_t^j$ is the solution at the initial grid $GRID.xx$. These values are assigned to $POL.YY$ by the `solve_proj`, and can be evaluated directly (Line 20).

For the other solution methods we explicitly evaluate the policy using $GRID.xx$, where the column vectors are capital and productivity (in logs) as shown in Line 4 and 5. For the algorithms using polynomials we save computation time by evaluating the period t policy function with the polynomial of the initial grid. This is done by setting `spec_opt = 'ini_grid'` in Line 14. This option is ignored for the algorithm `'spl_dir'`.

To evaluate the policy function in period $t+1$ we use:

$$\hat{C}_{t+1} = \begin{cases} \hat{C}\left(K_{t+1}, Z_{t+1}; \theta^{j-1}\right) & \text{if sol. meth. is 'tmi'} \\ \hat{C}(K_{t+1}, Z_{t+1}; \theta) & \text{else} \end{cases} \tag{4.6}$$

For Time Iteration we use the policy function of the previous iteration θ^{j-1}, which we achieve by setting `spec_opt_next = 'old_pol'` when calling `get_pol_var`, as shown in Lines 34 and 50. For the other solution methods the field `spec_opt_next'` is left empty (Line 36).

In what follows we ignore the differences in (4.5) and (4.6), and use the more general notation $\hat{C}_t = \hat{C}\left(K_t, Z_t; \theta^j\right)$ and $\hat{C}_{t+1} = \hat{C}\left(K_{t+1}, Z_{t+1}; \theta^{j-1}\right)$.

Auxiliary variables

In the model function we need to calculate the capital stock in the next period K_{t+1}, and the marginal productivity of capital $MPK_t = \alpha K_t^{\alpha-1} H_t^{1-\alpha}$, for a given state $[K_t, Z_t]$ and consumption C_t. To obtain these variables we compute the labor supply. The labor supply H is an explicit function of the state variables K and Z, and consumption C:

$$\begin{aligned} H_t &= \left[\frac{1-\alpha}{\chi} C_t^{-\nu} Z_t K_t^{\alpha}\right]^{\frac{\eta}{1+\alpha\eta}} \\ &= H(K_t, Z_t, C_t) \end{aligned} \tag{4.7}$$

After substituting out H_t the capital stock in $t+1$ is a function of K_t, Z_t and C_t as well:

$$\begin{aligned} K_{t+1} &= Z_t K_t^{\alpha} H\left(K_t, Z_t, C_t\right)^{1-\alpha} + \left(1-\delta\right) K_t - C_t \\ &= K\left(K_t, Z_t, C_t\right) \end{aligned} \tag{4.8}$$

The function `stnd_rbc_aux` computes these three variables H_t, MPK_t and K_{t+1} (in logs):

```
function [LK_n,LMPK,LH] = stnd_rbc_aux(par,LK,LZ,LC)
% Get log(K_t+1), log(MPK_t) and log(H_t)
% for standard RBC model

%Hours worked (in logs):
LH = par.eta/(1+par.alpha*par.eta) * ...
        ( -log(par.chi) -par.nu*LC + ...
        log(1-par.alpha) + LZ + par.alpha*LK );

% Marginal productivity of capital (in logs):
LMPK = log(par.alpha) + LZ + (par.alpha-1)*(LK-LH);

%Capital in next period (in logs):
LK_n = log(exp(LZ + par.alpha*LK + (1-par.alpha)*LH)...
        - exp(LC) + (1-par.delta)*exp(LK));

end
```

This function is called twice in the model function. The first time to calculate K_{t+1} in Line 24 of Listing 4.1, and a second time to calculate the marginal productivity in $t+1$ in Line 53.

Formally the approximation of the labor supply and capital stock in the next period are:

$$\hat{H}_t = H\left(K_t, Z_t, \hat{C}\left(K_t, Z_t; \theta^j\right)\right) \tag{4.9}$$

$$\hat{K}_{t+1} = K\left(K_t, Z_t, \hat{C}\left(K_t, Z_t; \theta^j\right)\right) \tag{4.10}$$

Expected value and Gauss-Hermite quadrature

Next we need to evaluate the right-hand side of the Euler equation (4.3), which consists of time $t+1$ variables. Using (4.7) we define a function G for the right-hand side of the Euler equation:

$$G\left(K_{t+1}, Z_{t+1}, C_{t+1}\right) = \beta C_{t+1}^{-\nu}\left[Z_{t+1}\alpha K_{t+1}^{\alpha-1} H\left(K_{t+1}, Z_{t+1}, C_{t+1}\right)^{1-\alpha} + 1 - \delta\right]$$

The approximation of G is:

$$\hat{\Phi}\left(K_{t+1}, Z_{t+1}; \theta^{j-1}\right) = G\left(K_{t+1}, Z_{t+1}, \hat{C}\left(K_{t+1}, Z_{t+1}; \theta^{j-1}\right)\right) \tag{4.11}$$

The right-hand side of the Euler equation (4.3) includes an expectation operator. We use Gauss-Hermite quadrature to approximate the expected value. Assume we have a function $f\left(z_{t+1}, x\right)$ and z_{t+1} is governed by (4.4) with standard normally distributed shocks ϵ_{t+1}. The Gauss-Hermite approximation is then:

$$E_t f\left(z_{t+1}, x\right) \approx \sum_{l=1}^{L} \frac{\omega_l}{\sqrt{\pi}} f\left(\rho_z z_t + \sigma_z \sqrt{2} \zeta_l, x\right) \tag{4.12}$$

where ζ_l are the Gauss-Hermite nodes, and ω_l are the Gauss-Hermite weights (see Section 12.2 for the derivation).

These nodes and weights were set in Step 1A of our code (see Section 4.3) as `par.her.xi(ll)` and `par.her.wi(ll)`, respectively.

We use the Gauss-Hermite formula (4.12), and capital in $t+1$ (4.10) to compute the approximation of the expected value of (4.11):

$$\begin{aligned}
&\Psi\left(K_t, Z_t; \theta\right) \approx E_t \left\{\hat{\Phi}\left(K\left(K_t, Z_t, \hat{C}\left(K_t, Z_t; \theta^j\right)\right), Z_{t+1}; \theta^{j-1}\right)\right\} \\
&= \sum_{l=1}^{L} \frac{\omega_l}{\sqrt{\pi}} \hat{\Phi}\left(K\left(K_t, Z_t, \hat{C}\left(K_t, Z_t; \theta^j\right)\right), \exp\left(\rho_z \log\left(Z_t\right) + \sigma_z \sqrt{2} \zeta_l\right); \theta^{j-1}\right)
\end{aligned} \tag{4.13}$$

In our code Listing 4.1 the Gauss-Hermite quadrature (4.13) is calculated as follows. We loop over the shocks and weights l in Line 42 till 60. In Line 63 we sum over the nodes and weights to obtain the right-hand-side of the Euler equation.

Euler residuals

After substituting (4.13) for the right-hand side of the Euler equation (4.3) the residuals R are a function that depends on K_t, Z_t and the parameters θ:

$$R\left(K_t, Z_t; \theta\right) = \Psi\left(K_t, Z_t; \theta\right) / \hat{C}\left(K_t, Z_t; \theta^j\right)^{-\nu} - 1 \tag{4.14}$$

We have divided both sides of the Euler equation by $\hat{C}_t^{-\nu}$ to ensure that the approximation is good over the whole interval. Without this scaling the normalized Euler residuals will be larger for high levels of consumption. This is due to the risk aversion, which makes the absolute errors in equation (4.3) smaller for high levels of consumption.

Step 3B: Create function handle

The function handle of the model file needs to be passed as the argument *fun_res* to the function `solve_proj`, which solves the model (see Step 5 in Section 4.7). In our example we create the handle with:

```
%% STEP 3: Handle for objective function
% (ie. the model file)
fun_res      = @(POL)STND_RBC_proj(par,GRID,POL);
```

It should be noted that after a function handle has been created any changes to the inputs arguments (other than the variable *POL*) will *not* change the function handle when it is called. For example in our program we created the function handle *fun_res*. When we change *par.alpha* to `par.alpha = 0.5` after the handle has been created then the function handle *fun_res* will still use the original value `par.alpha = 0.36`.

4.6 Step 4: Initial guess for the policy function

The initial guess for the policy function $Y0$ should be the policy variable on the initial grid. The values $Y0$ are an input for the solver `solve_proj`:

- $Y0$: the initial value of the policy variable on the initial grid in $GRID.xx$.

Good starting values are valuable for two reasons. The first is that most algorithms are not guaranteed to find a solution, although Time Iteration should converge to the solution when the shape of the policy function is preserved sufficiently[22]. The second reason is that good starting values will significantly reduce computation time.

There are four methods for the initialization of a policy function which usually ensure convergence:

- Perturbation methods;
- Gradually changing parameters;
- Gradually increasing the grid size;
- Increasing the order of the approximation.

Perturbation solution

For most models an initialization of the policy function using a linear perturbation solution will be sufficient for convergence. Perturbation solutions can be obtained easily as a wide range of software packages are available. In addition the computation time is relatively short, so perturbation methods are the natural choice for an initial guess. It should be noted that models

[22]See Judd (1998), page 554 and 555).

featuring an attracting limit cycle can also be solved with perturbation methods (Galizia, 2018)[23].

Gradually changing parameters

For some models we can easily obtain the solution by shutting a particular mechanism down, for example by setting a parameter to 0 or 1. A loop has to be added that gradually changes the parameter value.

Gradually increasing the size of the grid

It might be difficult to get a solution when the size of the grid is large. We could start with a small grid around the steady state such that a solution is found. The grid size can then be increased using a loop. Note that for Chebyshev polynomials (including Smolyak's algorithm) changing the boundaries affects the coefficients $POL.theta$, due to the scaling down of the variables to $[-1, 1]$.

Increasing the accuracy of the approximation

We can increase the number of nodes or the order of polynomials in a multi-step approach. This is especially useful for the algorithms that rely on Chebyshev polynomials, since the Chebyshev basis functions are orthogonal to each other. For example, Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016) use this approach.

Example Standard RBC model

In our example function `main_stnd_rbc_proj` we initialize the policy function using the first order perturbation solution. The (log) linear perturbation solution is:

$$\hat{c}\left(k_t, z_t\right) = \overline{c} + H_{y,k}\left(k_t - \overline{k}\right) + H_{y,z}\left(z_t - \overline{z}\right)$$

The coefficient $H_{y,k}$ and $H_{y,z}$ are the first and second entry in $PERT.Hy\_w$ of our code. The initial guess for the policy function is:

```
%Initial policy function (for given grid):
Y0 = log(SS.css) + ...
        PERT.Hy_w(1,1)*(GRID.xx(:,1)-log(SS.kss)) + ...
        PERT.Hy_w(1,2)*(GRID.xx(:,2)-log(SS.zss));
```

We have included two options to determine the coefficient in $PERT.Hy\_w$, which are chosen by setting $par.opt.get\_pert\_sol$ to either 0 or 1.

If `par.opt.get_pert_sol == 0` the initial guess is based on a poor estimation of the perturbation solution with $Hy\_w = [0.25, 0.25]$, while the first order perturbation solution is $Hy\_w = [0.3456, 0.3525]$. In our code:

[23] The `CSD` toolbox in the folder 'TOOLS' can solve saddle cycle models. The essential part is the code `InvSubGen` written by Dana Galizia.

```
if par.opt.get_pert_sol == 0
  %Poor estimation:
  PERT.Hy_w = [0.25,0.25];
```

If `par.opt.get_pert_sol == 1` the program obtains the coefficients $H_{y,k}$ and $H_{y,z}$ by solving the model with perturbation techniques. In our code we obtain the first order perturbation solution using the `CSD` toolbox, which is included in the folder 'TOOLS'. One could calculate these coefficients with other software packages, such as `Dynare`. The model file for the perturbation solution is the function `STND_RBC_pert` in the folder 'Examples\STND_RBC_mod'.

When this option is chosen the code executes:

```
else% Solve model with perturbation
  % Symbolic model file:
  MOD = STND_RBC_pert;

  % Vector of parameters:
  MOD.par_val = [par.alpha,par.beta,...
                 par.delta,par.eta,par.nu,par.chi];

  % Vector of steady states:
  MOD.SS_vec  = [log(SS.kss),log(SS.zss),log(SS.css)];

  % Get solution:
  PERT = pert_ana_csd(MOD,par.rho_z,1,par.sigma_z);

  clear MOD;
end
```

4.7 Step 5: Solving the policy function

The model is solved in Step 5. The function `solve_proj` minimizes the Euler residuals, and will assign the optimal policy function to the structure POL. The function requires the inputs $GRID$, POL, the function handle of the model file fun_res , and the initial guess for the policy function $Y0$. In our example Step 5 is:

```
%% STEP 5: Solve the model
POL   = solve_proj(GRID,POL,fun_res,Y0);
```

The function `solve_proj` internally assigns the policy function to POL. This policy function will be a spline or a polynomial depending on the algorithm. It uses an iterative scheme to find the policy function that minimizes the Euler residuals.

For all solution methods except Time Iteration we solve the objective function directly with either `fsolve` or `lsqnonlin` of the `Optimization Toolbox`. For Time Iteration an updating technique is used that is especially useful for recursive dynamic problems. More details on the algorithms can be found in Chapter 7.

Optional: stopping criteria for solvers

All algorithms use either `fsolve` or `lsqnonlin`. The toolbox uses the default stopping criteria of these solver, but they can be adjusted by setting the input *options* of `solve_proj` as shown below. When the solution method Time Iteration is used one can additionally adjust the tolerances res_tol and $diff\_tol$ (see Chapter 9 for more details). In our example code one can choose the default accuracy by setting $par.opt\_acc$ =, or choose a higher accuracy by setting $par.opt\_acc$ =. The latter will reproduce the results in Table 4.1 in Section 4.9.

```
%  (OPTIONAL) Set stopping criteria :
par.options = [];
par.opt_acc = 1;%0: default accuracy; 1: higher accuracy
if par.opt_acc == 1
  if strcmp(POL.algo,'spl_tmi') ||...
  strcmp(POL.algo,'cheb_tmi') || ...
  strcmp(POL.algo,'smol_tmi')
    % For Time Iteration:
    par.tmi_tol     = 1e-12;
    POL.res_tol         = par.tmi_tol;
    POL.diff_tol        = par.tmi_tol;
  else
    par.tol = 1e-12;
    par.options.OptimalityTolerance = par.tol;
    par.options.FunctionTolerance = par.tol;
    par.options.StepTolerance = par.tol;
  end
end
```

4.8 Step 6: Evaluating the policy function

The policy function can be evaluated with the function `get_pol_var`, which we also used in the model function in Step 4. The input variables are the structure $GRID$, the policy function POL, and an $m \times n$ matrix with the state variables (xx). Each of the n column represents a state variable, and is m is the number of data points. The output is the policy variable in a column vector ($m \times 1$).

In the script `main_stnd_rbc_proj` we plot the policy function using the model specific plotting function `plot_pol_stnd_rbc`. The graphs show that the approximated policy function is close to log linear, also outside the grid.

We also evaluate the policy function in a stochastic simulation. The simulation is carried out by the function `stnd_rbc_sim`. We call this function in the following block:

```
%% Step 6: Evaluate policy function (in simulation)
opt_sim.TT      = 1500; % # periods in simulation
opt_sim.T_ini   = 10;   % ini. periods at steady state
opt_sim.rws     = 2;   % number of simulated series

[SIM] = stnd_rbc_sim(par,SS,POL,GRID,opt_sim);
```

In this code *opt_sim* is a structure which sets the number of periods in the simulation (TT), the number of series to simulate (rws), and also the initial number of periods at the deterministic steady state (T_ini).

The function `stnd_rbc_sim` loops over time, and evaluates the policy function in each period:

Listing 4.2: Simulation in `main_stnd_rbc_proj`

```
% loop over time :
for it = T_ini+1:T_ini+TT

  % Calculate TFP (add shock)
  LZ(:,it) = par.rho_z * LZ(:,it-1) + par.sigma_z * epsilon(:,it);

  % Calculate policy variable:
  LC(:,it) = get_pol_var(POL,[LK(:,it),LZ(:,it)],GRID);

  % Calculate K_t+1 and H_t:
  [LK(:,it+1),~,LH(:,it)] =  stnd_rbc_aux(par,LK(:,it),LZ(:,it),LC(:,it));

end
```

The loop first calculates Total Factor Productivity (LZ), which takes normally distributed shocks *epsilon* as input. Next it evaluates the policy function for consumption (LC) given the state variables $LK(:,it)$ and $LZ(:,it)$. Finally it computes capital in the next period $LK(:,it+1)$, and the auxiliary variable hours worked $LH(:,it)$ using the function `stnd_rbc_aux`, which we discussed in Section 4.5.

4.9 Performance

In this section we review the computation time and accuracy for 5 algorithms. We review splines with Direct Computation (`'spl_dir'`) and Time Iteration (`'spl_tmi'`), Chebyshev with Galerkin's method (`'cheb_gal'`), Smolyak's algorithm with Direct Computation (`'smol_dir'`), and monomials with minimization of squared errors (`'mono_mse`).

The computation time is the time needed to solve the model, excluding the computation of the errors. The errors are the normalized Euler Equation Errors calculated in consumption equivalent unit as in Judd (1992):

$$EEE\left(K_t, Z_t; \theta\right) = \frac{\Psi\left(K_t, Z_t; \theta\right)^{-1/\nu}}{\hat{C}\left(K_t, Z_t; \theta\right)} - 1 \tag{4.15}$$

where $\Psi\left(K_t, Z_t; \theta\right)$ is defined in equation 4.13. We compute the errors on the initial grid (*on grid)*, and on a grid with 1,000 equidistant nodes in each dimension (*off grid*). When *on grid* and *off grid* errors are of similar magnitude for splines and Smolyak algorithms then the accuracy can be improved by using tighter stopping criteria (see Section 4.7). For complete polynomials it might not be possible to achieve higher accuracy by tighter stopping criteria, because complete polynomials are overidentified as the number of gridpoints is higher than the number of parameters[24].

The results are shown in Table 4.1. It should first be noted that computation times are low. For all basis functions a maximum (off grid) error of 10^{-6} can obtained in less than 0.05 seconds. In fact, projection can be faster and more accurate than perturbation when we solve a model only once.

A Spline with Direct Computation (`'spl_dir'`) is fast and accurate for a low number of gridpoints, but computation times increase rapidly with the number of nodes. The reason is that the algorithm has to numerically approximate an $m \times m$ Jacobian matrix, for m total gridpoints. Due to the high non-linearity[25] of the system no solution is found for 25 and 50 nodes in each dimension. For larger grids Time Iteration (`'spl_tmi'`) is faster. With splines the error decay up to 50 nodes[26] is $\mathcal{O}\left(q^{-4}\right)$, where q is the number of nodes in each dimension. This is an accordance with De Boor (1978).

The Chebyshev polynomial with Galerkin's method (`'cheb_gal'`) is both fast and accurate. An maximum error of -7.1 (in log10) is achieved in 0.04 seconds, and an error of -13 (in log10) in less than 0.3 seconds. Smolyak's algorithm with Direct Computation (`'smol_dir'`) is also fast and accurate. To achieve a certain accuracy level Smolyak's algorithm is slower than a Chebyshev polynomial with Galerkin's method. The main reason is that

[24] The exception are policy functions with one state variable and the number of nodes set to the order plus 1. These are exactly identified.

[25] Changing the solution at one gridpoint will change the spline. This will affect the solution at other gridpoints in a non-linear way.

[26] The error decay will be lower when the inaccuracy of the solution at the gridpoints comes into play.

Table 4.1: Performance for Standard RBC model

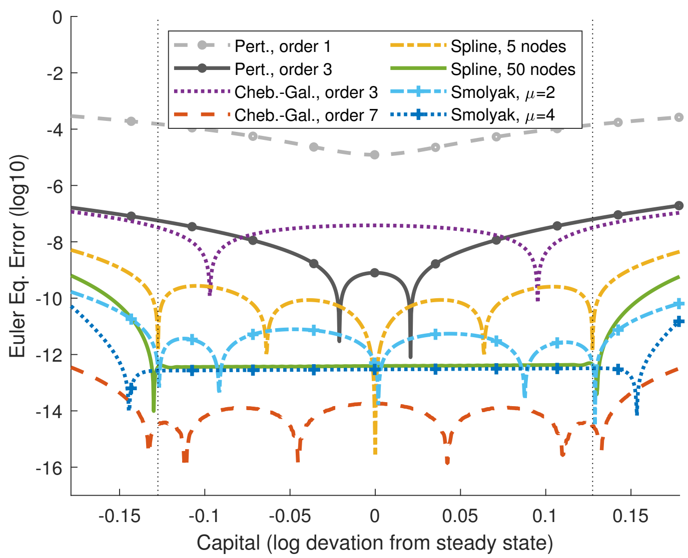
	Spline, Direct Computation						
Nodes per dim.	3	5	7	10	15		
Total Nodes	9	25	49	100	225		
Comp. time	0.05	0.06	0.12	0.27	0.73		
EEE, off grid	-6.3	-8.9	-9.5	-10.1	-10.8		
EEE, on grid	-15.4	-15.4	-14.9	-14.5	-14.1		
	Spline, Time Iteration						
Nodes per dim.	3	5	7	10	15	25	50
Total Nodes	9	25	49	100	225	625	2500
Comp. time	0.81	0.85	0.90	0.99	1.21	2.05	5.00
EEE, off grid	-6.3	-8.9	-9.5	-10.1	-10.8	-11.7	-12.3
EEE, on grid	-12.3	-12.3	-12.3	-12.3	-12.3	-12.3	-12.3
	Complete Chebyshev poly., Galerkin						
Order	1	2	3	4	5	6	7
Total Nodes	4	9	16	25	36	49	64
Comp. time	0.04	0.02	0.04	0.07	0.11	0.18	0.28
EEE, off grid	-3.4	-5.6	-7.1	-8.8	-10.8	-12.0	-13.4
EEE, on grid	-3.9	-6.0	-7.4	-9.2	-10.9	-12.3	-13.7
	Smolyak, Direct Computation						
Accuracy (μ)	1	2	3	4			
Total Nodes	5	13	29	65			
Comp. time	0.04	0.04	0.15	0.64			
EEE, off grid	-3.7	-7.5	-11.1	-12.8			
EEE, on grid	-15.6	-15.5	-15.1	-12.8			
	Monomials, min. of squared errors						
Order	1	2	3	4	4	5	5
Total Nodes	4	9	16	25	121	36	121
Comp. time	0.04	0.02	0.02	0.05	0.06	0.06	0.07
EEE, off grid	-3.6	-5.9	-7.2	-9.0	-8.7	-8.4	-8.3
EEE, on grid	-3.6	-5.9	-7.2	-9.0	-8.7	-8.4	-8.3
	Perturbation						
Order	1	2	3				
Comp. time	0.15	0.18	0.30				
EEE (off grid)	-3.32	-4.44	-6.38				

Computation times in seconds. Errors are the maximum Euler Equation Errors, in absolute values and log10. 'Off grid' refers to the equidistant grid with 1 million nodes, while 'on grid' refers to the initial grid used to solve the model.

Smolyak's polynomial is less effective in reducing errors. For example with $\mu = 4$ the polynomial is of degree 16 and there are 65 gridpoints, which achieves an error of -12.8 (in log10). For a complete Chebyshev polynomial of degree 7 we need 64 gridpoints, and the error is -13.4 in log10.

Monomial basis functions with minimization of squared errors (`'mono_mse'`) is faster and more accurate than Chebyshev polynomials up to order 4. For higher order approximations monomials are inaccurate. In fact,

Figure 4.1: Euler Equation Errors for RBC model (at $Z = 1$)



Vertical dotted lines are the bounds of the grid.

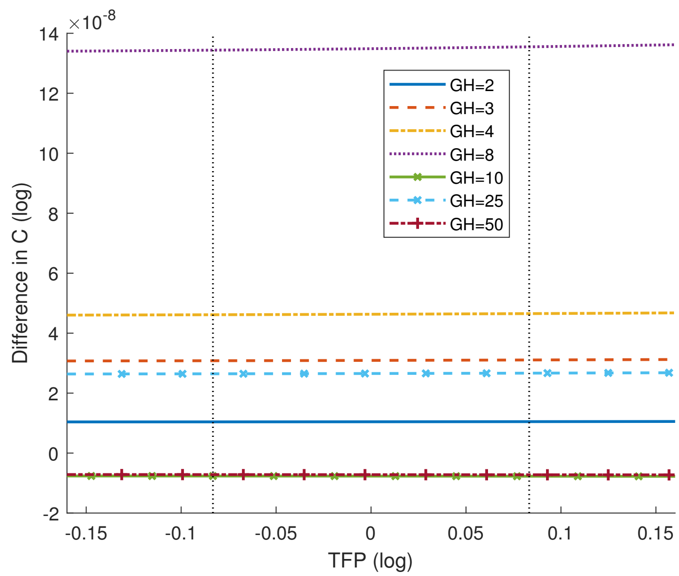
the error increases with the order of the approximation above order 4. This is due to the collinearity of monomials, and scaling issues as discussed in Section 7.3. We increased the number of nodes for the order 4 and 5 monomials, which has little effect on the accuracy. This will be the case in general, also for complete Chebyshev polynomials as Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016) confirm.

We plot the errors as a function of the capital stock in Figure 4.1. The figure shows that the errors within the boundaries are relatively constant for projection methods. With perturbation methods errors increase further from the steady state.

4.10 Sensitivity Gauss-Hermite quadrature

To investigate the effect of the number of Gauss-Hermite nodes on the policy function we plot the policy function for different number of Gauss-Hermite nodes. In Figure 4.2 the difference with the solution for 5 Gauss-Hermite nodes is plotted. The other state variable, capital, is set at its non-stochastic steady state. The differences are of the order 10^{-7} for $\sigma_z = 0.01$. We included the most extreme differences (10 and 8 nodes). For other numbers of nodes the differences are smaller, for example for 6, 7, 9, 11, 15, 20, 30 and 40 nodes (not shown). Given the small differences we may conclude that the policy function is relatively insensitive to the number of Gauss-Hermite nodes as long as more

Figure 4.2: Difference with 5 Gauss-Hermite nodes (at $K = K_{ss}$)



Vertical dotted lines are the boundaries of the grid used in the approximation.

than 1 node is chosen. With one node the differences are of the order 10^{-4}.

5. Multiple policy variables

This chapter illustrates how a model with multiple policy variables can be solved. We use a simple RBC model with an extra asset, housing. This 'Housing Model' (see Chapter 13 for more model details) is solved in the program `main_housing_proj` in the folder 'Examples'. This chapter focuses on the two main differences between solving a model with one policy variable and multiple policy variables.

The first difference is that each policy variable gets an index. The index is determined by the column of the variable in the initial guess. For example if we have two policy variables a and b, and the initial guess is $Y0 = [a_0, b_0]$ then variable a gets index $i\_pol = 1$ and variable b index $i\_pol = 2$. The index i_pol is used when evaluating the policy function with `get_pol_var`.

The second difference is that the residuals of the model function need to be grouped. Assume each policy variable can be linked to a specific residual vector, R_a and R_b, each with length m. The output of the model function needs to be the vector $R = [R_a; R_b]$, which has dimensions $2m \times 1$. Stacking the residual vectors vertically ensures that row i and row $i+m$ in R refer to the same gridpoint. In addition, when policy variable a affects R_a directly, and b affects R_b directly, then R_a and R_b should be stacked vertically in the same order as a and b were stacked horizontally in $Y0$.

5.1 Housing model

The model consists of three state variables, which are capital K_t, housing D_t and Total Factor Productivity Z_t. We use two policy variables, which are capital in the next period, K_{t+1} and current period consumption C_t[27]. The model is captured by five equations (see Section 13 for details):

$$C_t + K_{t+1} + D_{t+1} \leq Z_t K_t^{\alpha} + (1-\delta_k) K_t + (1-\delta_d) D_t \tag{5.1}$$

$$C_t^{-\nu} = \lambda_t \tag{5.2}$$

$$\lambda_t = \beta E_t \left\{ \lambda_{t+1} \left[Z_{t+1} \alpha K_{t+1}^{\alpha-1} + 1 - \delta_k \right] \right\} \tag{5.3}$$

$$\lambda_t = \beta E_t \left\{ \varrho D_{t+1}^{-\eta} + \lambda_{t+1} (1-\delta_d) \right\} \tag{5.4}$$

$$z_t = \rho_z z_{t-1} + \sigma_z \epsilon_t \tag{5.5}$$

where smaller cases indicate logs, ie. $z_t = \log(Z_t)$. Housing is D, capital is K, consumption is C, the multiplier on the budget constraint is λ, productivity is Z, the autocorrelation coefficient ρ_z , and the shocks are scaled by σ_z. The shocks are standard normally distributed, ie. $\epsilon_t \sim \mathcal{N}(0,1)$. Note that equation (5.3) is the Euler equation for capital, and (5.4) the Euler equation for housing.

5.2 Policy function

The model consists of three state variables, K_t, D_t and Z_t. We solve the model by approximating the policies for K_{t+1} and C_t. The policy function for capital in the next period is $K_{t+1} = \hat{K}\left(K_t, H_t, Z_t; \theta^1\right) = \exp\left[\hat{k}\left(k_t, h_t, z_t; \theta^1\right)\right]$. The policy function for consumption is $C_t = \hat{C}\left(K_t, H_t, Z_t; \theta^2\right) = \exp\left[\hat{c}\left(k_t, h_t, z_t; \theta^2\right)\right]$. The superscript j in θ^j is the index of the policy variable. This index is determined by the column index of the initial guess $Y0$. To ensure capital gets index $i\_pol = 1$ we put capital in the first column of $Y0$, and housing in the second column. The initial guess is based on the first order perturbation solution:

```
%pre-allocate dimensions
Y0 = NaN(GRID.mm,2);
```

[27] Using D_{t+1} as second policy variable will result in worse convergence for most algorithms, since the marginal utility of consumption λ will be affected directly by both K_{t+1} and D_{t+1} through the budget constraint. With C_t (instead of D_{t+1}) as policy variable the marginal utility of consumption in $t+1$ will only be affected indirectly by K_{t+1}. This helps solving the model more effectively.

```
%Initial guess for capital:
Y0(:,1) = log(SS.Kss) + ...
  0.9608*(GRID.xx(:,1)-log(SS.Kss)) +...
  0.0540*(GRID.xx(:,2)-log(SS.Dss)) +...
  0.0829*(GRID.xx(:,3)-log(SS.Zss));

%Initial guess for consumption:
Y0(:,2) = log(SS.Css) + ...
  0.4722*(GRID.xx(:,1)-log(SS.Kss)) +...
  0.0266*(GRID.xx(:,2)-log(SS.Dss)) +...
  0.3865*(GRID.xx(:,3)-log(SS.Zss));
```

With $Y0$ we have determined the indices *i_pol* of the policy variables. When calling `get_pol_var` the fourth argument has to be set to $i\_pol = 1$ to obtain capital K_{t+1} and $i\_pol = 2$ to obtain consumption C_t. For Time Iteration the policy variables are the columns in $POL.YY$ in the same order as in $Y0$. In our example we evaluate the policy function at the initial grid as follows:

```
LK = GRID.xx(:,1);%first state variable,    log(K_t)
LD = GRID.xx(:,2);%second state variable,   log(D_t)
LZ = GRID.xx(:,3);%third state variable,    log(Z_t)

%policy variables, log(K_t+1) and  log(C_t):
if ~(strcmp(POL.algo,'spl_tmi') || ...          strcmp
    (POL.algo,'smol_tmi') ||...
  strcmp(POL.algo,'cheb_tmi') )

  %use initial grid for polynomials
  % (or ignore for spl_dir)
  spec_opt    = 'ini_grid';
  % fourth entry is index for policy variable (i_pol)
  LK_n          = get_pol_var(POL,[LK,LD,LZ],GRID,1,
      spec_opt);
  LC    = get_pol_var(POL,[LK,LD,LZ],GRID,2,spec_opt);

else %for 'spl_tmi','cheb_tmi','smol_tmi':

  %Pol. variables in columns of POL.YY for 'tmi':
  LK_n    = POL.YY(:,1);
  LC      = POL.YY(:,2);
end
```

5.3 Residuals

The Euler residuals are calculated similarly to the standard RBC example discussed in Chapter 4. For given policies for K_{t+1} and C_t we can determine housing D_{t+1} from the budget constraint:

$$\hat{D}_{t+1} = Z_t K_t^{\alpha} + (1-\delta_k) K_t + (1-\delta_d) D_t - \hat{K}\left(K_t, H_t, Z_t; \theta^k\right) - \hat{C}\left(K_t, H_t, Z_t; \theta^c\right)$$

The multiplier λ_t is defined by (5.2).

As in the other models we need to evaluate next period's policy functions. Next period's choice determines $\hat{C}_{t+1}$, and enables us to calculate the two residual functions R^1 and R^2, corresponding to (5.3) and (5.4), respectively[28]:

$$\begin{aligned} R^1 &= \beta E_t \left\{ \hat{\lambda}_{t+1} \left[Z_{t+1} \alpha \hat{K}_{t+1}^{\alpha-1} + 1 - \delta_k \right] \right\} / \hat{\lambda}_t - 1 \\ R^2 &= \beta E_t \left\{ \varrho \hat{D}_{t+1}^{-\eta} + \hat{\lambda}_{t+1} (1-\delta_d) \right\} / \hat{\lambda}_t - 1 \end{aligned}$$

where we divided both sides of the Euler equation with λ to get more equally distributed normalized errors as explained in Section 4.5. In our model function we stack the residual vectors vertically, such that RES is a $2m \times 1$ vector:

```
% Euler residuals
RES1     = sum(rhs_j1,2)/lambda - 1;
RES2     = sum(rhs_j2,2)/lambda - 1;

% RES1 and RES2 are mm by 1 vectors
% concatenated vertically:
RES = [RES1;RES2];
```

Note that the first residual vector $RES1$ is more directly linked with the first policy variable K_{t+1}. This requires that the residual vector $RES1$ comes first when stacking the residual vectors vertically, because K_{t+1} was also the first policy variables ($i\_pol = 1$)[29].

[28]See Step 3A in Section 4.5 for details on how to approximate the expected value using Gauss-Hermite quadrature.

[29]This ordering is required for Galerkin's algorithm, because the coefficients of each policy function are set such that the corresponding residuals are orthogonal to the polynomial terms.

Part II

Theoretical description of algorithms

6. General Approach

The goal of each of the algorithms is to numerically approximate a policy function that solves a recursive dynamic optimization problem. A policy function gives the control (or policy) variable Y as a function of the state variables x. If the exact policy function is $Y(x)$ the algorithm approximates this function with $\hat{Y}(x;\theta)$ where θ is a vector of parameters of the basis function, either a spline or a polynomial.

The objective of projection methods is to find the policy function that solves the dynamic optimization problem. The algorithms require a function that computes the Euler residuals for a given approximation of the policy function. In practice the residual function $R(x;\theta)$ is a function with the model equations, which is discussed in Chapter 8. These residuals are computed on the initial grid, which consists of gridpoints of the n state variables. The construction of the grid is discussed in Section 6.1. Next we discuss Collocation in Section 6.2. With collocation we solve a model at the gridpoints using either Direct Computation or Time Iteration, which are the most intuitive solution methods. The approaches using Minimization of Squared Error (least squares) and Galerkin's method are discussed in the next Chapter 7.

The remainder of this Chapter is mostly copied from Duineveld (2021). We explain the general approach with the Deterministic Brock-Mirman model used in Chapter 3. We use a slightly more formal approach as in that chapter.

The relevant equations are the resource constraint and the First Order Condition:

$$\begin{aligned} K_{t+1} + C_t &= K_t^{\alpha} \\ C_t^{-1} &= \beta C_{t+1}^{-1} \alpha K_{t+1}^{\alpha-1} \end{aligned} \tag{6.1}$$

We choose consumption C_t as the policy variable, which is a function of the state variable capital K_t. There exists a policy function $C_t = C(K_t)$, which exactly solves this dynamic optimization problem. Instead we numerically approximate the policy function with $\hat{C}(K_t;\theta)$. The approximation $\hat{C}(K_t;\theta)$ will not exactly solve this dynamic system, and we need to compute the errors with a residual function.

Given the approximation of consumption we can compute next period's variables:

$$\hat{K}_{t+1} = \hat{K}\left(K_t;\theta^j\right) = K_t^{\alpha} - \hat{C}\left(K_t;\theta^j\right) \tag{6.2}$$

$$\hat{C}_{t+1} = \hat{C}\left(\hat{K}_{t+1};\theta^{j-1}\right) \tag{6.3}$$

where superscript j indicates the current period choices, and $j-1$ next period's choices, although this difference is only relevant for Time Iteration.

The residuals in the Euler equation (6.1) as a result of the approximation are:

$$R\left(K_t;\theta\right)=\beta\hat{C}\left(\hat{K}\left(K_t;\theta^j\right);\theta^{j-1}\right)^{-1}\alpha\hat{K}\left(K_t;\theta^j\right)^{\alpha-1}-\hat{C}\left(K_t;\theta^j\right)^{-1} \tag{6.4}$$

The objective is to find the policy function that minimizes the residuals on a specified interval of the state variables. This interval is determined by a lower and upper bound of each state variable. Within this interval discrete points, or gridpoints, are chosen, where we evaluate the Euler residuals.

6.1 Defining a grid

The policy function is set such that the residuals on the gridpoints are minimized. The grid is defined over an interval of each of the n state variables where we want the approximation to be good. The intervals are defined by the lower and upper bounds $\left[\underline{x}^i,\overline{x}^i\right]$ for $i=[1,\ldots,n]$.

For all methods except Smolyak's algorithm[30] we use a Cartesian product to construct the grid. For each state variable i a set of q^i nodes $X^i=\left\{x_1^i,x_2^i,\ldots,x_{q^i}^i\right\}$ are defined on the interval between the lower and upper bound. For splines and monomials we use equidistant nodes, and for complete Chebyshev polynomials we use the Chebyshev nodes. We call the Cartesian product of these sets the initial grid:

$$\mathcal{X}=X^1\times\ldots\times X^n \tag{6.5}$$

This set consists of $m=\prod_{i=1}^n q^i$ points where the residual function is evaluated.

The toolbox constructs the grid with the function `prepgrid`. The set of nodes X^i are assigned to the cell array $GRID.gridVecs$. The initial grid (6.5) is the field $GRID.xx$. Each column in $GRID.xx$ represents a state variable.

6.2 Collocation

With collocation we solve the model at the m gridpoints with either Direct Computation or Time Iteration. With Direct Computation we solve a system of equations with a non-linear equation solver based on a Newton-type of algorithm. This methods numerically estimates the $m\times m$ Jacobian matrix of the system equations. Time Iteration is specifically designed to solve a recursive dynamic system of equations. The algorithms chooses the period t policy variable at the gridpoints, while holding the period $t+1$ policy function constant. This ensures that the $m\times m$ Jacobian matrix of the system of

[30]See Section 7.5 for the construction of the grid with Smolyak's algorithm.

equations is sparse with only entries on the diagonal. Each iteration is therefore computationally less intensive, but also less effective than with Direct Computation.

For both Direct Computation and Time Iteration the solution at the gridpoints uniquely defines the coefficients θ of the basis function, either a spline or a polynomial. Assume the algorithms finds the solution $\tilde{Y}$ at the gridpoints $x \in \mathcal{X}$. The coefficients of the basis function θ can be determined by some function:

$$\theta = \Gamma\left(x, \tilde{Y}\right) \tag{6.6}$$

Direct Computation

For Direct Computation the residuals can be written as a function of either the coefficients θ or the choice variable $\tilde{C}$ at the gridpoints. In equation (6.4) the residuals were a function of the coefficients θ. Using (6.6) the alternative formulation of (6.4) in terms of choice variable $\tilde{C}$ is:

$$R\left(K_t, \tilde{C}_t\right) = R\left(K_t; \Gamma\left(K_t, \tilde{C}_t\right)\right) \tag{6.7}$$

The objective is to set the residual equal to 0 at all m gridpoints[31]. In the original formulation:

$$0 = R\left(K_t; \Gamma\left(K_t, \tilde{C}_t\right)\right) \tag{6.8}$$

This system of equations can be solved with a Newton-type of non-linear equation solver[32]. This type of solver will converge at least quadratically close to the solution (Judd, 1998). Note that a change in $\tilde{C}_{i,t}$ at gridpoint i affects the coefficients θ, which also affects the solution at other gridpoints $j \neq i$. This necessitates the numerical approximation of the dense $m \times m$ Jacobian. As the Jacobian has m^2 elements each iteration is computationally expensive for a large number of gridpoints m. This makes the solution method is less efficient than Time Iteration for large grids. For highly non-linear systems Direct Computation might have convergence issues[33].

Time Iteration

The Time Iteration algorithm is described by Judd (1998). Compared to Direct Computation it economizes on the iteration step. In iteration j the algorithm solves for period t choices $\tilde{C}_t^{\,j}$, while using the coefficients of the

[31] See equation (6.5).

[32] By default the toolbox uses Matlab's `fsolve` with the 'trust-region-dogleg' algorithm for Direct Computation.

[33] The algorithm will locally linearize the system of equations, which might be a poor approximation far away from the solution.

previous iteration θ^{j-1} for period $t+1$ choices. For the simple example we replace (6.2) and (6.3) with:

$$\begin{aligned}\hat{K}_{t+1} &= K_t^{\alpha} - \tilde{C}_t^j = \hat{K}\left(K_t, \tilde{C}_t^j\right) \\ \hat{C}_{t+1} &= \hat{C}\left(\hat{K}_{t+1}; \theta^{j-1}\right) \\ &= \hat{C}\left(\hat{K}\left(K_t, \tilde{C}_t^j\right); \theta^{j-1}\right)\end{aligned}$$

This gives us the residual function:

$$R\left(K_t, \tilde{C}_t^j; \theta^{j-1}\right) = \beta \hat{C}\left(\hat{K}\left(K_t, \tilde{C}_t^j\right); \theta^{j-1}\right)^{-1} \alpha \hat{K}\left(K_t, \tilde{C}_t^j\right)^{\alpha-1} - \left(\tilde{C}_t^j\right)^{-1} \tag{6.9}$$

Equation (6.9) defines a system of m non-linear equations in as many unknowns $\tilde{C}_t^j$. We solve the system of equations $R\left(K_t, \tilde{C}_t^j; \theta^{j-1}\right) = 0$ with a Newton-type of algorithm[34]. As the residual at gridpoint i only depends on $\tilde{C}_{i,t}^j$ the Jacobian matrix is sparse with entries on the diagonal only[35]. In addition, we do not have to recompute the spline or polynomial when numerically approximating the Jacobian. This makes the algorithm efficient for recursive dynamic problems. After solving the system of equations (6.9) for the policy variable $\tilde{C}_t^j$ we update the coefficients θ using (6.6), and repeat the process until convergence.

We use two stopping criteria, which both have to be satisfied. The first criterion is the maximum absolute difference in the policy variable between iterations, which has to satisfy $\max\left|\tilde{Y}^j - \tilde{Y}^{j-1}\right| \leq \epsilon^d$. The second criterion is the maximum Euler residual at the gridpoints when using the updated policy θ^j for both current and next period's policy. Formally this stopping criterion is $\max\left|R\left(x, \tilde{Y}^j; \theta^j\right)\right| \leq \epsilon^r$, where x are the state variables.

7. Algorithms

In this Chapter we discuss all the algorithms. This chapter is also mostly copied from Duineveld (2021).

7.1 Splines

Splines are determined by Matlab's `griddedInterpolant`. Spline basis functions are best used in combination with collocation. The policy variable(s)

[34] By default the toolbox uses Matlab's `fsolve` with the 'trust-region' algorithm for square problems. This algorithm allows the use of a sparse Jacobian.

[35] With multiple policy functions the Jacobian consists of repeated blocks of diagonal matrices.

$\tilde{Y}$ at each gridpoint $x \in \mathcal{X}$ is obtained with either Direct Computation or Time Iteration. The solution at the gridpoints determines the parameters θ of a piece-wise cubic polynomial, or spline as in (6.6). For simplicity we use a grid with equidistant nodes.

By default the toolbox sets the interpolation method of `griddedInterpolant` to `'spline'`[36]. This interpolation method uses a cubic spline determined with not-a-knot end conditions, which results in a twice differentiable spline. When such a cubic spline is used to approximate a four times differentiable function the convergence is $\mathcal{O}\left(q^{-4}\right)$, where q is the number of nodes per dimension (De Boor, 1978).

For the univariate case with q data points $(x_1, y_1), \ldots, (x_q, y_q)$ a cubic spline takes the piece-wise form:

$$
\begin{aligned}
S_1(x) &= y_1 + \theta_{1,1}\Delta x_1 + \theta_{1,2}\Delta x_1^2 + \theta_{1,3}\Delta x_1^3 \quad \text{for } x \in [x_1, x_2] \\
S_2(x) &= y_2 + \theta_{2,1}\Delta x_2 + \theta_{2,2}\Delta x_2^2 + \theta_{2,3}\Delta x_2^3 \quad \text{for } x \in [x_2, x_3] \\
\vdots \ &\ \vdots \\
S_{q-1}(x) &= y_{q-1} + \theta_{q-1,1}\Delta x_{q-1} + \theta_{q-1,2}\Delta x_{q-1}^2 + \theta_{q-1,3}\Delta x_{q-1}^3 \quad \text{for } x \in [x_{q-1}, x_q]
\end{aligned}
$$

with $\Delta x_i = x - x_i$.

This univariate spline has $3(q-1)$ coefficients, which can be determined with the following conditions. At the interior points the function needs to be continuous, which gives us $q-1$ conditions:

$$S_i(x_{i+1}) = y_{i+1}$$

In addition the first and second derivative have to be continuous at the interior points, which gives us two times $q-2$ conditions:

$$
\begin{aligned}
S_i^{'}(x_{i+1}) &= S_{i+1}^{'}(x_{i+1}) \\
S_i^{''}(x_{i+1}) &= S_{i+1}^{''}(x_{i+1})
\end{aligned}
$$

The not-a-knot end conditions require that the third derivative is also continuous at the gridpoints x_2 and x_{q-1}:

$$
\begin{aligned}
S_1^{'''}(x_2) &= S_2^{'''}(x_2) \\
S_{q-2}^{'''}(x_{q-1}) &= S_{q-1}^{'''}(x_{q-1})
\end{aligned}
$$

This yields a linear system of $3(q-1)$ equations in the univariate case. With multi-dimensional interpolation each dimension is treated independently, and sequential one-dimensional interpolation is carried out[37].

[36] The interpolation method can be changed with the field *POL.spl_meth*.

[37] Matlab does not specify the algorithm for multi-dimensional interpolation, but the results are equivalent to sequential one dimensional interpolation.

Spline with Direct Computation

The algorithm `'spl_dir'` uses Direct Computation as discussed in Section 6.2. The objective is equation (6.8): solve the residual R at each gridpoint by adjusting the policy variable at the gridpoints. We need to numerically approximate the full Jacobian matrix, because a change in the policy variable at one gridpoint will change the spline, and therewith affect the solution at other gridpoints. With multiple policy variables the Jacobian of the system of equations will be an $dm \times dm$ matrix, where m is the total number of gridpoints, and d the number of policy variables.

Spline with Time Iteration

The algorithm `'spl_tmi'` uses Time Iteration as discussed in Section 6.2. We solve the residual function at each gridpoint by choosing the period t policy variable, holding the period $t+1$ policy function constant. For a single policy variable the solution at each gridpoint does not affect the solution at other gridpoints, and the Jacobian matrix will only have entries on the diagonal. With multiple variables the Jacobian matrix consists of repeated blocks of diagonal matrices.

7.2 General: construction of polynomials

For `'mono'`, `'cheb'`, and `'smolyak'` the polynomials are constructed with a similar procedure. The multivariate polynomial Ω ($m \times p$ matrix) is constructed with three elements: the initial grid xx, the array Φ which consists of univariate polynomial terms, the matrix LL which compiles the polynomial.

The following dimensions are used:

- m the total number of gridpoints;
- n the number of state variables;
- p the total number of (multivariate) polynomial terms;
- k the maximum degree of a (univariate) polynomial.

Initial grid x

The initial grid x is an $m \times n$ matrix, where each column j is a state variable x_j ($m \times 1$).

Univariate polynomials terms

We have to differentiate between monomials used for the algorithm `'mono_mse'` and Chebyshev polynomials used for the algorithms with `'cheb'`, and `'smolyak'`.

For univariate monomial terms are:

$$T_i(x) = x^i$$

For Chebyshev polynomials (of the first kind) the univariate terms have the recurrent relation:

$$\begin{aligned} T_0(\tilde{x}) &= 1 \\ T_1(\tilde{x}) &= \tilde{x} \\ T_{v+1}(\tilde{x}) &= 2\tilde{x}T_v(\tilde{x}) - T_{v-1}(\tilde{x}) \end{aligned} \tag{7.1}$$

where $\tilde{x}$ is a scaled down variable to the interval $[-1, 1]$. The transformation is a function $\tilde{x}(x)$ (see equation (7.7)) and this allows us to write $T_{v+1}(x) = T_{v+1}(\tilde{x}(x))$. Using index $j = 1, \ldots, n$ for the state variables we get:

$$T_i(x_j) = \begin{cases} x_j^i & \text{for monomials} \\ T_i(\tilde{x}(x_j)) & \text{for Chebyshev polynomials} \end{cases}$$

Note that each $T_i(x_j)$ is an $m \times 1$ vector.

Matrix of univariate polynomial terms Φ

The array Φ consists of all univariate polynomial terms up to order k. The dimensions of Φ are $m \times k \times n$, where m is the number of nodes in the initial grid, k is the degree of the polynomial, and n is the number of state variables.

For state variable j the polynomial terms are:

$$\Phi^j = [T_1(x_j), \ldots, T_k(x_j)]$$

which is an $m \times k$ matrix. Note that we omit $T_0 = 1$. We concatenate each Φ^j in the third dimension.

Element index LL

The matrix L $(p \times n)$ consists of indices that refer to the elements in Φ. The element $l_{i,j}$ refers to $T_{l_{i,j}}(x_j)$, meaning the order $l_{i,j}$ polynomial for state variable j. The elements in row i are multiplied by each other to form column i of the polynomial matrix Ω, meaning $\Omega_i = \prod_{j=1}^{n} T_{l_{i,j}}(x_j)$.

Polynomial

The $m \times p$ matrix Ω is:

$$\Omega(x) = \begin{bmatrix} \Omega_1(x_1) & \cdots & \Omega_p(x_1) \\ \vdots & \ddots & \vdots \\ \Omega_1(x_m) & \cdots & \Omega_p(x_m) \end{bmatrix} \tag{7.2}$$

In our code we use the names *XX_poly*[38] for Ω, and *poly_elem*[39] for L. The matrix Ω is constructed as:

```
XX_poly = ones(mm,pp);%initiate with ones

for ii = 1:pp
  for jj = 1:nn
        if LL(ii,jj)>0%omit T_0=1 terms
        XX_poly(:,ii) = XX_poly(:,ii) .* Phi(:,
            poly_elem(ii,jj),jj);
        end
  end
end
```

Complete polynomials

For the algorithms `'cheb'` and `'mono'` we use complete polynomials. A complete polynomial of degree k in n dimensions consists of all possible combinations with $\sum_{j=1}^{n} l_j \leq k$. Using short-hand notation $T_i^j = T_i(x_j)$ a complete polynomial of degree k in n dimensions is (Judd, 1998):

$$\mathscr{P}_k^n \equiv \left\{ T_{l_1}^1 \cdots T_{l_n}^n \mid \sum_{j=1}^{n} l_j \leq k, 0 \leq l_1, \ldots, l_n \right\} \tag{7.3}$$

The toolbox allows for asymmetric polynomials where state variable j has a maximum degree k_j polynomial[40], ie. the restriction $l_j \leq k_j$.

We use complete polynomials for the algorithms `'cheb'` and `'mono'`, because they achieve almost the same accuracy as a tensor product, despite having a lower number of coefficients (Judd, 1992). For a complete polynomial the number of coefficients grows polynomially in the number of dimensions, while tensor product grow exponentially (Judd, 1998).

Example with two dimensions

To illustrate the construction of a complete polynomial we use a two dimensional example, which is also used in Chapter 10. Assume we have two state variables with bounds $[1, 3]$ for x_1 and $[10, 25]$ for x_2.

[38] Or *XX_poly_dw* for scaled down variables.
[39] For the Smolyak algorithm *smol_el_ani* is $L + 1$ and Φ also contains $T_0 = 1$.
[40] The field *ord_vec* in the input argument *meth_spec* is the $1 \times n$ vector $[k_1, \ldots, k_n]$.

We start with the monomial case, and use 3 and 4 gridpoints for x_1 and for x_2, respectively. With equidistant nodes the resulting fields are $gridVecs\{1,1\} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ for x_1 and $gridVecs\{1,2\} = \begin{bmatrix} 10 & 15 & 20 & 25 \end{bmatrix}$ for x_2. The initial grid (12×2) is:

$$xx = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\ 10 & 10 & 10 & 15 & 15 & 15 & 20 & 20 & 20 & 25 & 25 & 25 \end{bmatrix}^{\mathsf{T}}$$

We want to construct a complete polynomial of degree $k = 2$. We construct the matrix Φ^j $(m \times k)$ for $j = 1, 2$ consisting of the univariate monomials up to order 2:

$$\Phi^j = \left[x_j, x_j^2\right]$$

We construct the polynomial with the matrix LL:

$$LL = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 2 & 1 & 0 \end{bmatrix}^{\mathsf{T}}$$

Each element $l_{i,j}$ in LL refers to a polynomial term in the column vector $\Phi^j\left(:, i\right)$ with $\Phi^j\left(:, 0\right) = 1$. The polynomial terms in row i are multiplied by each other to form column i of the polynomial matrix Ω. For example, row four of LL is $[2, 0]$, meaning $x_1^2 x_2^0 = x_1^2$, which is the entry in the fourth column of Ω. The complete polynomial $\Omega\left(x\right)$ (dimension 12×6) is:

$$\Omega\left(x\right) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix} \tag{7.4}$$

7.3 Monomials

With monomials the policy function is approximated with a complete polynomial. The grid with monomial basis functions consists of equidistant nodes. Monomials are simple to use, but have three disadvantages compared to Chebyshev polynomials. The first is that monomials are collinear. For example, x^2 and $x^4$are very close to each other around 0. The second disadvantage is that monomials are not scaled. For example, the term x^4 will have a complete different magnitude than the term x. The third disadvantage is known as Runge's phenomenon, which results in oscillation at the edges of an interval for polynomials of high degree when equidistant nodes are used.

To construct complete polynomials we use the procedure described above in Section 7.2. For example, the approximation with a complete polynomial of degree two with two state variables x_1 and x_2 is:

$$\hat{Y}\left(x; \theta\right) = \theta_1 + \theta_2 x_1 + \theta_3 x_2 + \theta_4 x_1^2 + \theta_5 x_1 x_2 + \theta_6 x_2^2 \tag{7.5}$$

Using the notation $\Omega_i(x)$ to refer to multivariate polynomial term i we may also write:

$$\hat{Y}(x;\theta) = \sum_{i=1}^{p} \theta_i \Omega_i(x)$$

where p is the total number of polynomial terms. Using the matrix notation as in (7.4) this is equivalent to:

$$\hat{Y}(x;\theta) = \Omega(x)\,\theta$$

Monomials with Minimization of Squared Errors

The algorithm `'mono_mse'` minimizes the squared errors at the gridpoints by setting the coefficients θ. Assume we have a residual function $R(x;\theta)$, which is evaluated at gridpoints $i = 1, \ldots, m$. The objective is to minimize the sum of the square residuals:

$$\min_{\theta} \sum_{i=1}^{m} R(x_i;\theta)^2 \tag{7.6}$$

where m is the total number of gridpoints. To minimize (7.6) the default algorithm is Matlab's `lsqnonlin` with a 'trust-region' algorithm.

7.4 Complete Chebyshev polynomials

The most commonly used basis functions are Chebyshev polynomials. Chebyshev polynomials are superior to monomial basis functions for three reasons. The first is that Chebyshev polynomials of the first kind have the discrete orthogonality property:

$$\sum_{k=0}^{N-1} T_i(x_k)\,T_j(x_k) = 0 \quad \text{if } i \neq j$$

where N is any integer greater than $\max(i,j)$, and x_k are the Chebyshev nodes (see below), and $T_i(x)$ is given by the recursive relation in (7.1). Note that this orthogonality is defined for the one dimensional case.

The second reason for the superiority of Chebyshev polynomials is that they are scaled such that the absolute value of the extrema never exceeds 1. The third reason is that they are very effective at reducing Runge's phenomenon. Runge's phenomenon is that polynomial interpolation results in oscillation at the edges of an interval for polynomials of high degree when equidistant nodes are used. This phenomenon is avoided with Chebyshev nodes.

To make use of the favorable properties of Chebyshev polynomials it is necessary to linearly map variables from the interval $[\underline{x}, \overline{x}]$ to $[-1, 1]$. This transformation is given by:

$$\tilde{x}(x) = 2\frac{x - \underline{x}}{\overline{x} - \underline{x}} - 1 \tag{7.7}$$

which we call the scaling down of variables. The inverse of this map, which we call scaling up, is:

$$x(\tilde{x}) = \frac{(\tilde{x} + 1)(\overline{x} - \underline{x})}{2} + \underline{x} \tag{7.8}$$

where subscripts are the degree of the polynomial.

To construct complete polynomials we use the procedure described above in Section 7.2. For example, the complete Chebyshev polynomial of degree two with two variables x_1 and x_2 is:

$$\hat{Y}(x;\theta) = \theta_1 + \theta_2\tilde{x}(x_1) + \theta_3\tilde{x}(x_2) + \theta_4\left(2\tilde{x}(x_1)^2 - 1\right) + \theta_5\tilde{x}(x_1)\tilde{x}(x_2) + \theta_6\left(2\tilde{x}(x_2)^2 - 1\right) \tag{7.9}$$

where θ are the coefficients on the polynomial terms $i = 1, \ldots, p$. Alternatively we can write:

$$\hat{Y}(x;\theta) = \sum_{i=1}^{p}\theta_i\Omega_i(x) \tag{7.10}$$

where $\Omega_i(x)$ refers to the multivariate polynomial term in column i of Ω, and p is the total number of polynomial terms. In matrix notation this becomes:

$$\hat{Y}(x;\theta) = \Omega(x)\theta \tag{7.11}$$

To construct the initial grid we use the Chebyshev nodes, which are in the interval $[-1, 1]$. These nodes are determined by the formula:

$$\tilde{x}_i = \cos\left(\frac{2i-1}{2q}\pi\right)$$

for $i = 1, \ldots, q$. The q nodes are the roots of the polynomial $T_q(x)$. For example for $q = 2$ the nodes are the roots of $T_2(x) = 2\tilde{x}^2 - 1$, which are $\pm\frac{1}{2}\sqrt{2}$. Note that these roots never include the bounds $[-1, 1]$.

Chebyshev with Galerkin's method

With the algorithm 'cheb_gal' we use Galerkin's method to obtain the coefficients θ in (7.10). We calculate the product of the residual function $R(x;\theta)$ and each polynomial term $\Omega_j(x)$ at all gridpoints $i = 1, \ldots, m$. The objective for each coefficient θ_j is to set the sum of these products to zero:

$$0 = \sum_{i=1}^{m} R(x_i;\theta)\,\Omega_j(x_i) \tag{7.12}$$

As there are $j = 1, \ldots, p$ coefficients this is a system of p equations in p unknowns. This system is solved using a non-linear equation solver, based on a Newton-type of algorithm. By default we use Matlab's `fsolve` with the 'trust-region-dogleg' algorithm. If multiple policy variables need to be solved each policy variable has its own residual function. With d policy variables this is a system of dp equations. For a large number of coefficients it is more efficient to use Time Iteration.

Chebyshev with Time Iteration

The algorithm 'cheb_tmi' solves the residual function at the gridpoints with Time Iteration as explained in Section 6.2. Given the solution at the gridpoints the coefficients θ are obtained by solving a least square problem. The solution at the gridpoints is a $m \times 1$ vector $\hat{Y}(x)$, and the $m \times p$ matrix Ω is defined in (7.11). The coefficients θ are determined using Matlab's `mldivide`, which gives the least-squares solution of a linear system of equations $\hat{Y} = \Omega\theta$ when it is overidentified.

Chebyshev with Minimization of Squared Error

The algorithm 'cheb_mse' minimizes the squared errors at the gridpoints by setting the coefficients as with 'mono_mse'. The objective is the same and given by equation (7.6). The differences are the scaling down of variables, the polynomial itself, and the nodes. To solve the objective we use Matlab's `lsqnonlin` with a 'trust-region' algorithm by default.

7.5 Smolyak's algorithm

Smolyak's algorithm can be implemented in various ways. We use the method described by Judd et al. (2014)[41]. The algorithm constructs a sparse grid consisting of Chebyshev extrema. The solution at the gridpoints determines the coefficients of a sparse Chebyshev polynomial. The (sparse) gridpoints are

[41] We implemented the provided Matlab code: Rafa Valero (2021), Smolyak Anisotropic Grid (https://www.mathworks.com/matlabcentral/fileexchange/50963-smolyak-anisotropic-grid), MATLAB Central File Exchange. Retrieved November, 2021.

concentrated on the axis and the corners of the grid[42]. The solution at the gridpoints is computed with Direct Computation or Time Iteration. With Direct Computation we use the coefficients θ as choice variables. With Time Iteration we solve the policy variable at the gridpoints, and obtain the coefficients by solving a linear system of equations.

For the construction of the isotropic sparse grid[43] we largely follow the exposition by Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016). First we choose the accuracy parameter μ[44]. The degree of the Chebyshev polynomial will be 2^{μ}. For accuracy $\mu > 0$ there are $q_{\mu} = 2^{\mu} + 1$ number of nodes in each dimension, and for $\mu = 0$ there is one node $q_0 = 1$.

The extrema of a univariate Chebyshev polynomial (also called Gauss-Lobotto nodes) for given μ with $j = 1, \ldots, q_{\mu}$ are (Judd et al., 2014):

$$\zeta_j^{\mu} = \begin{cases} 0 & \text{for } \mu = 0 \\ -\cos\left(\frac{j-1}{q_{\mu}-1}\pi\right) & \text{for } \mu > 0 \end{cases}$$

We define the nested sets:

$$\mathcal{G}^{\mu} = \left\{\zeta_1^{\mu}, \ldots, \zeta_{q_{\mu}}^{\mu}\right\}$$

where $\mathcal{G}^{\mu} \subset \mathcal{G}^{\mu+1}$. The first three sets are: $\mathcal{G}^0 = \{0\}$, $\mathcal{G}^1 = \{-1, 0, 1\}$, and $\mathcal{G}^2 = \left\{-1, -\frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2}, 1\right\}$. We introduce the notation $T_k^i = T_k(\tilde{x}_i)$, which is the univariate Chebyshev basis function of degree k in dimension i as defined in (7.1). The nodes $\mathcal{G}^{\mu}$ correspond to the extrema of the basis functions $T_0, \ldots, T_{2^{\mu}}$, with the extremum of T_0 set to 0.

The multivariate sparse grid is a union of the Cartesian products[45]:

$$\mathbb{G}(\mu, n) = \bigcup_{\sum \mu_n = \mu} \left(\mathcal{G}^{\mu_1} \times \ldots \times \mathcal{G}^{\mu_n}\right) \tag{7.13}$$

For example with $n = 2$ dimensions and $\mu = 1$ (meaning a degree $2^{\mu} = 2$ polynomial) we get:

[42] See for example Figure 11 in Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016).

[43] The toolbox allows for the construction of an anisotropic grid as described by Judd et al. (2014). The procedure is very similar, where the accuracy parameter μ is specified for each dimension.

[44] In the notation of Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016): $\mu = q - n$.

[45] Fernández-Villaverde, Rubio-Ramírez, and Schorfheide (2016) include Cartesian products with $\sum \mu_n < \mu$, but since $\mathcal{G}^{\mu} \subset \mathcal{G}^{\mu+1}$ these lower ranked Cartesian products are redundant. For example $\mathcal{G}^1 \times \mathcal{G}^0 \subset \mathcal{G}^2 \times \mathcal{G}^0$

$$
\begin{aligned}
\mathbb{G}(1,2) &= \bigcup_{\sum \mu_n = 1} \left(\mathcal{G}^{\mu_1} \times \mathcal{G}^{\mu_2}\right) \\
&= \left(\mathcal{G}^1 \times \mathcal{G}^0\right) \cup \left(\mathcal{G}^0 \times \mathcal{G}^1\right) \\
&= \{(-1,0),(0,0),(1,0)\} \cup \{(0,-1),(0,0),(0,1)\} \\
&= \{(0,0),(-1,0),(1,0),(0,-1),(0,1)\}
\end{aligned}
$$

Similarly with $n = 2$ dimensions and $\mu = 2$ (meaning a degree $2^\mu = 4$ polynomial) we get:

$$
\begin{aligned}
\mathbb{G}(2,2) &= \bigcup_{\sum \mu_n = 2} \left(\mathcal{G}^{\mu_1} \times \mathcal{G}^{\mu_2}\right) \\
&= \left(\mathcal{G}^2 \times \mathcal{G}^0\right) \cup \left(\mathcal{G}^0 \times \mathcal{G}^2\right) \cup \left(\mathcal{G}^1 \times \mathcal{G}^1\right) \cup \left(\mathcal{G}^1 \times \mathcal{G}^1\right)
\end{aligned}
$$

Note that $\mathcal{G}^1 \times \mathcal{G}^0 \subset \mathcal{G}^2 \times \mathcal{G}^0$, so $\mathbb{G}(1,2) \subset \mathbb{G}(2,2)$.

The sparse grid exactly identifies the coefficients of a polynomial, and we can infer from the grid, which polynomial terms are included. For example, for a two dimensional grid ($n = 2$) and an accuracy $\mu = 1$ we get a degree $2^\mu = 2$ polynomial. The grid consists of the sets $\left(\mathcal{G}^1 \times \mathcal{G}^0\right) \cup \left(\mathcal{G}^0 \times \mathcal{G}^1\right)$. This corresponds to a polynomial consisting of only univariate terms T_0, T_1^i, and T_2^i for $i = 1, 2$. The degree 2 bivariate terms $T_1^i T_1^j$ for $i \neq j$ are omitted.

To generalize this we define the set of univariate Chebyshev polynomials up to order k as $\mathcal{T}_i^k = \left\{T_0^i, T_1^i, \ldots, T_k^i\right\}$. Note that for accuracy μ the degree of the polynomial is 2^μ. For example, in two dimensions with $\mu_1 = 2$ and $\mu_2 = 1$ the Cartesian product $\mathcal{G}^2 \times \mathcal{G}^1$ defines 15 gridpoints. The corresponding tensor product $\mathcal{T}_1^{2^{\mu_1}} \otimes \mathcal{T}_2^{2^{\mu_2}}$ is a set of 15 bivariate polynomials:

$$
\mathcal{T}_1^4 \otimes \mathcal{T}_2^2 = \left\{ \begin{array}{ccc} T_0^1 T_0^2, & \ldots & , T_4^1 T_0^2 \\ T_0^1 T_1^2, & \ldots & , T_4^1 T_1^2 \\ T_0^1 T_2^2, & \ldots & , T_4^1 T_2^2 \end{array} \right\} \tag{7.14}
$$

This sparse grid and sparse set of polynomials is very effective at tackling the curse of the dimensionality. A standard Cartesian product with q nodes in n dimensions consists of a total of q^n nodes. We would need at least 5 gridpoints in each dimension to estimate a degree 4 complete polynomial. For 8 dimensions this would result in $5^8 = 390,625$ gridpoints to estimate a total of 495 coefficients.

The Smolyak algorithm with a sparse degree 4 polynomial (ie. $\mu = 2$) in 8 dimensions results in only 145 nodes and coefficients. The resulting polynomial consists of the univariate terms T_0, T_1^i, T_2^i, T_3^i, and T_4^i for $i = 1, \ldots, 8$, and all possible combinations of the bivariate terms $T_1^i T_1^j$, $T_2^i T_1^j$, and $T_2^i T_2^j$ for $i \neq j$. Compared to a complete polynomial 350 terms are omitted by the Smolyak algorithm: the degree 3 terms $T_1^i T_1^j T_1^l$, and the degree 4 terms $T_1^i T_1^j T_1^l T_1^m$,

$T_2^i T_1^j T_1^l$ and $T_3^i T_1^j$ for all possible combination with $i \neq j \neq l \neq m$. In general, a degree 4 Smolyak polynomial does not contain polynomial terms consisting of more than 2 variables.

Following the explanation by Judd et al. (2014) the polynomial results in the approximation of the policy variable:

$$\hat{Y}(x;\theta) = \sum_{i=1}^{p} \theta_i \Omega_i(x)$$

where p is the total number of polynomial terms, equal to the number of gridpoints m, and $\Omega_i(x)$ are the multivariate polynomial terms as explained in Section 7.2. We can write in matrix notation:

$$\hat{Y}(x;\theta) = \Omega(x)\,\theta$$

Smolyak with Direct Computation

The algorithm `'smol_dir'` directly solves the residuals at the gridpoints by setting the coefficients θ[46]. The objective is to solve:

$$0 = R(x;\theta)$$

which is a system of m equations in m unknowns as the number of coefficients is equal to the number of gridpoints. The Jacobian matrix of this system is $m \times m$ because each coefficient affects the solution at all gridpoints. For more details on Direct Computation see Section 6.2.

Smolyak with Time Iteration

The algorithm `'smol_tmi'` solves the residual function by choosing the period t policy variable with Time Iteration as explained in Section 6.2. Given the period t policy variable at the gridpoints $\hat{Y}(x)$, and the matrix with polynomial terms Ω (as in equation (7.2) in Section 7.2) we determine the coefficients θ by solving a linear system of equations:

$$\theta = \Omega^{-1}\hat{Y}$$

In practice we solve this linear system with Matlab's `mldivide`.

[46] This contrasts with `'spl_dir'`, where we solve for the policy variable, and then determine the coefficients using equation (6.6).

Part III

Technical descriptions of functions

8. Model file

The model should be a function that takes the grid and the policy functions as inputs and gives the Euler residuals as output. One has to create a handle to this model function, which only takes the structure with the policy function POL as input. To evaluate the policy function we use `get_pol_var` (described in Section 8.2). The function `get_pol_var` takes the state variables xx as input, which is an $m \times n$ matrix where each column vector represents a state variable, and each row a datapoint. The output is an $m \times d$ matrix, where each column represents a policy variable. One has to use an algorithm specific format to evaluate the policy function as explained in Section 8.1. For examples of model functions see Listing 3.3 and Listing 4.1.

The model function itself can include any amount of input fields, but needs to include at least:

- $GRID$: a structure with all necessary properties of the grid as assigned by the function `prepgrid` (see Section 10.1);
- POL: a structure that needs to contain the algorithm $algo$, and the policy function (see Section 9.1). Note that the policy function will be assigned to POL by the function `solve_proj`, based on the algorithm and the initial guess.

The output of the function needs to be:

- RES: residuals in a $dm \times 1$ vector, where m is the total number of gridpoints, and d the number of policy functions. When multiple policy variables are used ($d > 1$) the residuals vectors should be stacked vertically as explained below.

Vertical concatenation of residuals

The residuals vectors need to be stacked vertically. An example of this ordering can be found in Chapter 5. To explain the ordering we assume there are two policy variables ($d = 2$), and the total number of gridpoints is m. As $d = 2$ there are two residuals functions R_1 and R_2, which are both column vectors with length m. These vectors need to be stacked vertically such that the model function returns a $2m \times 1$ residual vector $R = [R_1; R_2]$. Grouping the residuals this way ensures that row j and row $j + m$ in R refer to the same gridpoint. This format is necessary, because the sparse Jacobian for Time Iteration is constructed based on this format.

In addition the order of the residual vectors should correspond to the order of the policy variables. In the example of Chapter 5 we had an Euler equation for capital and an Euler equation for consumption. The complete residual vector is constructed as $R = [R_1; R_2]$. As capital is the first policy variable the Euler residuals for capital should be in R_1 and the other Euler residuals in R_2.

8.1 Formats for policy function evaluation

To evaluate the policy function in the model file we need to differentiate between two formats. There is a standard format and a format specific to Time Iteration algorithms (`'tmi'`). Examples of these formats are shown in Listing 3.3 and Listing 4.1. In this section we assume there are $i = 1, \ldots, d$ policy variables y^i. Each policy variable is a function of the two state variables $X1$ and $X2$.

Standard Format

With the standard format the same policy function is used for current period's choices and next period's choices. To evaluate policy variable i (index *i_pol*) in period t use:

```
y_t = get_pol_var(POL,[X1_t,X2_t],GRID,i_pol);
```

We can use the same format to evaluate the policy function in period $t + 1$.

For algorithms with polynomials not using Time Iteration[47] we can save computation time by evaluating the policy function at the initial grid, which is stored in the structure $GRID$. We have to set the input argument `spec_opt='ini_grid'` of `get_pol_var` to achieve this:

```
spec_opt = 'ini_grid';
y_t = get_pol_var(POL,[X1_t,X2_t],GRID,i_pol,spec_opt);
```

For the polynomial algorithms this option will ignore the inputs arguments $[X1_t, X2_t]$, and the policy function will be evaluated using the initial grid $GRID.xx$. When the option `spec_opt='ini_grid'` is used in combination with a spline `get_pol_var` ignores the option, and inputs $[X1_t, X2_t]$ are used to evaluate the policy function.

Time Iteration

With Time Iteration we solve for the period t policy variable, given the policy function in period $t + 1$ as explained in Section 6.2. With Time Iteration the solver `solve_proj` assigns the period t policy variable at the gridpoints to the field $POL.YY$, which is an $m \times d$ matrix where d is the number of policy variables, and m the number of gridpoints. To evaluate the period t policy variable with index *i_pol* we call:

[47]These are the algorithms `'cheb_gal'`, `'cheb_mse'`, `'mono_mse'`, and `'smol_dir'`.

```
y_t = POL.YY(:,i_pol);
```

For next period's choices we use the policy function of the previous iteration[48]. To evaluate this policy function we set `spec_opt='old_pol'` as input argument for `get_pol_var`:

```
spec_opt_next = 'old_pol';
y_n = get_pol_var(POL,[X1_n,X2_n],GRID,i_pol,
    spec_opt_next);
```

8.2 Function `get_pol_var`

The function `get_pol_var` takes the policy function in POL, the state variables in matrix xx, and the grid structure $GRID$ as input and gives the policy variable as output. When multiple policy variables are used one needs to specify the index *i_pol* of the policy variable.

The inputs are:

- POL: a structure which contains the field *algo* (see Chapter 7), and a field containing the appropriate policy function. For splines this is the field *pp_y* and for polynomials this is the field *theta*. For solution method `'tmi'` there are two special cases of the policy function. The period t policy function evaluated at the initial grid is the field YY, and the policy function of the previous iteration is the field *pp_y_old* for splines and *theta_old* for polynomials;

- xx: a matrix with the state variables in column vectors stacked behind each other ($m \times n$ matrix, where m is the number of points to be evaluated, and n is the number of state variables) as shown in Listings 3.3 and 4.1 for a model file, and in Listing 4.2 for a simulation. When `spec_opt='ini_grid'` the input xx is ignored for polynomials[49];

- $GRID$: structure with the necessary grid properties, which are assigned by `prepgrid` (see Chapter 10);

- *i_pol* (optional if $d = 1$): the index of the policy variable to be evaluated. The total number of policy variables is d. The index *i_pol* is determined by the column index in the initial guess $Y0$ (see Chapter 5 for an example);

- *spec_opt* (optional): there are two options for this field, either `spec_opt='ini_grid'` or `spec_opt='old_pol'`. Other values are ignored. If `spec_opt='ini_grid'` and the algorithm is `'cheb_gal'`,

[48] The old policy function is assigned to *POL.pp_y_old* for splines, and to *POL.theta_old* for polynomials.

[49] For polynomials in combination with Time Iteration the option `spec_opt='ini_grid'` will throw an error.

'cheb_mse', 'smol_dir', or 'mono_mse' the initial grid (either XX_poly or XX_poly_dw) is used to evaluate the policy function[50]. Note that the input argument xx will be ignored in this case. The option spec_opt='ini_grid' is ignored when used in combination with 'spl_dir', and will result in an error when used in combination with Time Iteration[51]. If spec_opt='old_pol' the old policy function is used. For splines this is the field *pp_y_old*, and for polynomials *theta_old*. These fields will be assigned to POL when solving the model with Time Iteration ('tmi').

9. Solving the model

The model is solved using the function `solve_proj`, which is explained in the following section.

9.1 Function `solve_proj`

The function `solve_proj` solves for the policy function that minimizes the Euler residuals. To solve the model one calls the function `solve_proj` with inputs:

- $GRID$: the structure with the grid properties assigned by `prepgrid` as described in Chapter 10;
- POL: a structure with the required field *algo*, that defines the algorithm. There are several optional fields, which are listed in the Subsection Optional Fields below;
- fun_res: the function handle to the model function as described in Section 4.5, and the examples Listing 3.3 and 4.1;
- $Y0$: the initial guess of the policy functions evaluated at the initial grid $GRID.xx$. $Y0$ should be a $m \times d$ matrix with m the total number of gridpoints, and d the number of policy variables. The column index of a variable in $Y0$ determines the index *i_pol*, which is used to evaluate the policy function with `get_pol_var` (see Section 8.2 and for an example Chapter 5);
- *options* (optional): a structure which can replace fields of the *optimoptions* structure of the solvers (either `lsqnonlin` or `fsolve`). If 'options.override_all = 1' then *options.optimoptions* is used as the *options* input argument of the solver. If *options.Algorithm* is set than

[50] This will save computation time as the polynomial of the initial grid does not have to be reconstructed.

[51] With Time Iteration the policy function at the initial grid is assigned to $POL.YY$.

the *optimoptions* structure will be created using the specified algorithm. Otherwise only the fields specified in *options* will be used to replace their standard values. See below for some details[52].

The output is:

- *POL*: a structure to which the policy function is added. For polynomial algorithms the policy functions are assigned to the field *theta*. This is a $p \times d$ matrix with the polynomial coefficients, where d is the number of policy variables, and p the number of polynomial terms. For spline algorithms the policy function is *pp_y*, which is $1 \times d$ cell array. Each cell contains a spline for a policy variable. The splines are created using Matlab's `griddedInterpolant`. The interpolation method is specified in *POL.spl_meth* for which `'spline'` is the default option.

The solver uses the following functions from Matlab's `Optimization Toolbox`:

- `lsqnonlin` for algorithms `'cheb_mse'` and `'mono_mse'`;
- `fsolve` for all other algorithms. Note that for Time Iteration this solver is only used to solve for the period t policy variable.

We use default options for `fsolve` and `lsqnonlin` with two exceptions. The first exception is for Time Iteration. For this solution method the algorithm of `fsolve` is set to `'trust-region'`, which allows for a sparse Jacobian matrix. The `'JacobPattern'` is set to sparse identity matrix[53]. The second exception is the `'Display'` option, which is set to `'off'` by default.

Algorithms

We repeat the overview of the algorithms in Table 9.1. For recommendations we refer to Section 1.4. The details of each algorithm are discussed in Chapter 7.

Table 9.1: Overview of algorithms

Algorithm	Basis function	Proj. Cond.	Solution Meth.
'spl_dir'	Spline	Collocation	Direct Comp.
'spl_tmi'	Spline	Collocation	Time Iteration
'cheb_gal'	Compl. Chebyshev polyn.	Galerkin	Newton type
'cheb_tmi'	Compl. Chebyshev polyn.	Collocation	Time Iteration
'cheb_mse'	Compl. Chebyshev polyn.	Min. Sq. Err.	Trust-Region
'mono_mse'	Monomials (compl. polyn.)	Min. Sq. Err.	Trust-Region
'smol_dir'	Smolyak-Chebyshev polyn.	Collocation	Direct Comp.
'smol_tmi'	Smolyak-Chebyshev polyn.	Collocation	Time Iteration

[52] Or see the subfunction `set_default_opt_solver` inside `solve_proj`.

[53] In case of multiple policy variables the pattern consists of repeated blocks of identity matrices.

Optional fields

There are several optional fields in POL.

Spline algorithms

- spl_meth: for the algorithms `'spl_tmi'` and `'spl_dir'` the interpolation method of the spline can be set in the optional field $POL.spl\_meth$. The Matlab documentation for the function `griddedInterpolant` describes the choices under 'Method'. The default of the toolbox is `'spline'`.

Time Iteration

For the solution method `'tmi'` several options can be set in fields of the structure POL:

- $diff\_tol$: the tolerance for the maximum absolute change in the the policy function between two iterations: $\max \left|\hat{y}^j - \hat{y}^{j-1}\right|$where $\hat{y}^j$ is the policy variable of iteration j. The default is 1e-8;
- res_tol: the tolerance $\epsilon^r_{\max}$, which is the acceptance level for the maximum absolute value of the (Euler) residuals: $\max \left| R\left(\hat{y}^j_t; \theta^j\right)\right|$. The default is 1e-8;
- max_iter: the maximum number of iterations in the `'while'` loop. The default is 500;
- $step\_acc$: all tolerances are scaled with $step\_acc$ when the solver stalls[54]. The default value is 0.1;
- mem_Y: memory parameter when updating the policy function Y, ie. $Y = (1 - mem_Y)Y_{new} + mem_Y Y_{old}$, where Y_{new} is the solution at the gridpoints found in the current iteration. The default is 0.

10. Construction of grid

The grid parameters, and also the grid itself are stored in the structure $GRID$. It is created with the function `prepgrid`, which takes the grid parameters, and the algorithm as inputs. The output is the structure $GRID$ which includes all the required fields.

The `prepgrid` function is demonstrated in Section 10.4 with the example script `grid_example` in the folder 'PROMES_v05.0.0/Examples', which shows all the output variables (and intermediate variables). All functions of this chapter except `prepgrid` are found in the subfolders

[54] When $output.iterations == 0$ indicating that the solver got stuck at the initial point.

'PROMES_v05.0.0/grid_subfun' and 'PROMES_v05.0.0/smolyak_subfun'. The latter folder contains the code to construct the Smolyak grid. This code is provided by Rafa Valero[55], and the underlying algorithm is described in Judd et al. (2014). The relevant subfolders need to be on the searchpath to construct the grid.

10.1 Function `prepgrid`

The function `prepgrid` constructs structure $GRID$ with all the necessary fields. It mainly prepares the call to *gridstruct* or *gridstruct_smolyak*. The function `prepgrid` has five input arguments:

- *nn*: the number of state variables;
- *lb*: vector of lower bounds in each dimension (1 x *nn* vector);
- *ub*: vector of upper bounds in each dimension (1 x *nn* vector);
- *algo*: the algorithm, which should be assigned to $POL.algo$. See Subsection 9.1 for a list of options;
- *algo_spec* (optional): structure with algorithm specific fields. If this input is not specified the default values are used. For the `'spline'` algorithms this is the field *qq*, which is set to $qq = 5 * ones(1, nn)$ by default. For the algorithms with `'cheb'` and `'mono'` the specific fields are *qq* and *ord_vec*. For `'cheb'` the default values are $qq = 6 * ones(1, nn)$, and $ord\_vec = 5 * ones(1, nn)$. For `'mono'` the default values are $qq = 4 * ones(1, nn)$, and $ord\_vec = 3 * ones(1, nn)$. For the `'smol'` algorithms only the field *mu* has to be assigned, which is set to $mu = 2 * ones(1, nn)$ by default.

The function `prepgrid` assigns the *grid_type*, which is either `'spline'`, `'cheb'`, `'mono'` or `'smolyak'`. For the types `'spline'`, `'cheb'`, and `'mono'` we call the function `gridstruct`. For the type `'smolyak'` we call the function `gridstruct_smolyak`. These two functions construct the structure $GRID$ and are explained in next two sections.

10.2 Function `gridstruct`

The function `gridstruct` is used to create a structure that contains all the necessary properties of the grid when the *grid_type* is `'spline'`, `'cheb'`, or `'mono'`. The necessary properties include the gridvectors, the full grid, and if required the complete polynomial of the grid.

The input arguments are:

[55] Rafa Valero (2021) Smolyak Anisotropic Grid, MATLAB Central File Exchange, Retrieved November 5, 2021 (https://www.mathworks.com/matlabcentral/fileexchange/50963-smolyak-anisotropic-grid).

- nn: the number of state variables
- qq: vector of number of gridpoints in each dimension ($1 \times n$ vector);
- lb: vector of lower bounds in each dimension ($1 \times n$ vector);
- ub: vector of upper bounds in each dimension ($1 \times n$ vector);
- $grid\_type$: either `'cheb'` for a complete Chebyshev polynomial with Chebyshev nodes, `'mono'` for a complete polynomial based on monomials with equidistant nodes, or `'spline'` for an equidistant grid;
- ord_vec: the maximum order of the polynomial in each dimension ($1 \times n$ vector). This input is only required for $grid\_type$ `'cheb'`, and `'mono'`

The function `gridstruct` adds the input arguments nn, qq, lb, ub, and $grid\_type$ as fields to $GRID$. Several additional fields are added, which are best explained with the example in Section 10.4. The additional fields are:

- mm which is the total number of gridpoints;
- $gridVecs$, and xx which contain the initial gridvectors (see Sections 10.5), and the initial grid (see Section 10.6), respectively;
- $poly\_elem$ which is a $p \times n$ matrix which is used to construct a multivariate polynomial as in Section 7.2. The element $l_{i,j}$ refers to the univariate polynomial $T_{l_{i,j}}(x_j)$. The multiplication of the polynomial terms in a row construct a multivariate polynomial (either XX_poly or XX_poly_dw, see the example in Section 10.4);
- XX_poly if the $grid\_type$ is `'mono'`, which is the complete polynomial of the full grid xx (see Section10.9);
- $gridVecs\_dw$, xx_dw, and XX_poly_dw if the $grid\_type$ is `'cheb'` . These are the scaled down versions of $gridVecs$, xx, and XX_poly, respectively. The scaling down maps the variables linearly from $[lb, ub]$ to $[-1, 1]$. The nodes are the Chebyshev nodes. For more details see Sections 10.5, 10.6 and 10.9, respectively.

10.3 Function `gridstruct_smolyak`

The function `gridstruct_smolyak` is used to create a structure that contains all the necessary properties of the grid when the $grid\_type$ is `'smolyak'`. The necessary properties include the gridvectors, the full grid, the Smolyak polynomial, and the matrix inverse of the polynomial. This function makes calls to the code provided by Rafa Valero[56]. The algorithm underlying this code is originally described in Judd et al. (2014), and also found in Section 7.5.

[56]See Footnote 55.

This manual does not further explain the code by Rafa Valero, which can be found in the subfolder 'smolyak_subfun'.

The input arguments are:

- *nn*: the number of state variables
- *lb*: vector of lower bounds in each dimension ($1 \times n$ vector);
- *ub*: vector of upper bounds in each dimension ($1 \times n$ vector);
- *mu_vec*: vector of the accuracy *mu* in each dimension ($1 \times n$ vector).

The function `gridstruct_smolyak` adds the input arguments *nn*, *lb*, *ub*, and *grid_type* as fields to *GRID*. Several additional fields are added, which are best explained with the example in Section 10.4. The additional fields are:

- *mm* which is the total number of gridpoints;
- *xx* which is the initial grid (see Section 7.5);
- *smol_elem_ani* which is a $p \times n$ matrix used to construct a multivariate polynomial as in Section 7.2. The element $l_{i,j}$ refers to the univariate polynomial $T_{l_{i,j}-1}(x_j)$. The multiplication of the polynomial terms in a single row give a multivariate polynomial *XX_pol_dw*, see the example in Section 10.4;
- *xx_dw*, and *XX_pol_dw*, which are the scaled down initial grid, and the Smolyak polynomial of this grid. The scaling down maps the variables linearly from $[lb, ub]$ to $[-1, 1]$. The nodes are the Chebyshev extrema. For more details see Sections 7.5.

10.4 Example `grid_example`

We demonstrate the fields of the structure *GRID* with the file `grid_example`, which prints some of the properties of the grid on screen. The function `grid_example` can be found in the folder 'PROMES_v05.0.0/Examples'. That example prints the various grid variables on screen, and plots the initial grid. The code (excluding the printing commands) is:

```
% Add relevant folders of Promes toolbox:
addpath ('..');
addpath ('..\grid_subfun');
addpath ('..\smolyak_subfun');

%% Initializtion of grid parameters:
gin.nn = 2;%number of state variables
gin.lb = [1,10];%lower bounds for [x1,x2]
gin.ub = [3,25];%upper bounds for [x1,x2]

```

```
% Set solution method
% 'cheb_gal','cheb_tmi','cheb_mse',
% 'spl_tmi','spl_tmi',
% 'smol_tmi','smol_tmi',
% 'mono_mse';
algo = 'cheb_gal';

if strncmp(algo,'cheb',4) || strncmp(algo,'mono',4)
  meth_spec.ord_vec = 2*ones(1,gin.nn);%order in each dim.
  meth_spec.qq = [3,4];%number of nodes in each dim.

elseif strncmp(algo,'spl',3)
  meth_spec.qq       = [3,4];%number of nodes in each dim.

elseif strncmp(algo,'smol',4)
  meth_spec.mu_vec = [2,2];%accuracy in each dim.
end

% Construct structure with grid:
[GRID] = prepgrid(gin.nn,gin.lb,gin.ub,algo,meth_spec);
```

All algorithms except Smolyak

The grid example is discussed for all algorithms except the Smolyak algorithm. The other algorithms use an initial grid based on a Cartesian product. The grid for these algorithms is constructed with the function `gridstruct`. The Smolyak grid is constructed differently.

Gridvectors

The function `gridstruct` first constructs the gridvectors *gridVecs*, plus the scaled down versions in *gridVecs_dw* in case the *grid_type* is `'cheb'`. These gridvectors consist of either equidistant or Chebyshev nodes with the specified amount of gridpoints, and on the interval determined by the lower and upper bound. The gridvectors are constructed using the function `constr_vecs` (see Section 10.5). The gridvectors will be added as fields to the structure *GRID*.

In the example we have specified that the state variables have 3 and 4 gridpoints, with bounds $[1,3]$ for x_1 and $[10,25]$ for x_2. If the *grid_type* is `'mono'` or `'spline'` the function `gridstruct` will use equidistant nodes for the gridvectors. With equidistant nodes the resulting fields are $gridVecs\{1,1\} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ for x_1 and

$gridVecs\{1,2\} = \begin{bmatrix} 10 & 15 & 20 & 25 \end{bmatrix}$ for x_2, which are linearly spaced vectors from lower to upper bound with the specified amount of gridpoints.

For the algorithms using Chebyshev polynomials `prepgrid` sets the $grid\_type$ to `'cheb'`, which means Chebyshev nodes are used. These nodes in the interval $[-1, 1]$ are added as $gridVecs\_dw$. Scaled down variables are computed using the linear mapping from the lower and upper bounds $[lb, ub]$ to $[-1, 1]$. The linear map is:

$$\tilde{x} = \frac{2x}{ub - lb} - \frac{lb + ub}{ub - lb} \tag{10.1}$$

where $\tilde{x}$ denotes the scaled down variable x (see Section 10.13).

The scaled down nodes correspond to the roots of the Chebyshev polynomials. If there are q nodes, then these nodes are the roots of the order q polynomial. For example, for $q = 2$ the nodes are the roots of the second order polynomial $2\tilde{x}_1^2 - 1$. These roots are $\pm\frac{1}{2}\sqrt{2}$. The minimum number of nodes in each dimension is therefore the order of the polynomial plus 1[57].

In our case we have $q = 3$ and $q = 4$, which results in:

$$gridVecs\_dw\{1,1\} = \begin{bmatrix} -0.866 & 0 & 0.866 \end{bmatrix}$$
$$gridVecs\_dw\{1,2\} = \begin{bmatrix} -0.924 & -0.383 & 0.383 & 0.924 \end{bmatrix}$$

for $\tilde{x}_1$ and for $\tilde{x}_2$, respectively. To scale up these vectors into the interval $[lb, ub]$ we use the inverse of (10.1) (see Section 10.11). These scaled up vectors are stored in the field $gridVecs$.

Grid

After the gridvectors are constructed the function `gridstruct` will construct the initial grid xx using the function `constr_grid` (see Section 10.6), which takes the gridvectors as input. The function `constr_grid` constructs the initial grid using Matlab's `ndgrid`, which computes the Cartesian product of the gridvectors. For each state variable the output of `ndgrid` is transformed into a $m \times 1$ column vector with m being the total number of nodes. These column vectors are stacked next to each other to form the $m \times n$ matrix xx, where n is the number of state variables. In our code:

```
[x1,x2] = ndgrid(gridvecs{1,1},gridvecs{1,2});

xx      = NaN(mm,nn);
xx(:,1) = reshape(x1,[],1);
xx(:,2) = reshape(x2,[],1);
```

With the gridvectors $gridVecs\{1,1\} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ and $gridVecs\{1,2\} = \begin{bmatrix} 10 & 15 & 20 & 25 \end{bmatrix}$ the result is:

[57]Otherwise the complete polynomial will contain a column vector with zeros.

$$xx = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\ 10 & 10 & 10 & 15 & 15 & 15 & 20 & 20 & 20 & 25 & 25 & 25 \end{bmatrix}^{\mathsf{T}} \tag{10.2}$$

where each column (note the transpose) in x represents a state variable, and each row a unique gridpoint. For the Chebyshev polynomials we construct a full grid for both the scaled up (xx) and scaled down (xx_dw) variables.

Complete polynomials

For the grid types using polynomials (`grid_type='cheb'` and `grid_type='mono'`) the function `gridstruct` will construct complete polynomials as explained in Section 7.2. It first constructs the field *poly_elem* with the function `poly_elem_ani`, and then calls the function `get_poly_ani`. The complete polynomial with monomial basis functions of order two in both dimensions is:

$$XX\_poly = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix}$$

where $x_1 = xx\,(:,1)$ and $x_2 = xx\,(:,2)$ are the state variables in $m \times 1$ column vectors. The resulting complete polynomial XX_poly is printed on screen when the algorithm is set to `'mono_mse'`.

For Chebyshev polynomials only the scaled down grid $\tilde{x}$ ($GRID.xx\_dw$) is used. A second order complete Chebyshev polynomial with two variables consists of the terms:

$$XX\_poly\_dw = \begin{bmatrix} 1 & \tilde{x}_1 & \tilde{x}_2 & 2\tilde{x}_1^2 - 1 & \tilde{x}_1\tilde{x}_2 & 2\tilde{x}_2^2 - 1 \end{bmatrix} \tag{10.3}$$

The scaled down complete polynomial with Chebyshev nodes will be printed on screen when algorithm is set to either `'cheb_gal'`, `'cheb_tmi'` or `'mse'`.

10.5 Function `constr_vecs`

This function constructs a cell array $gridVecs$, where each cell contains a vector of gridpoints (for state variable i this is a $1 \times q\,(i)$ vector). This function allows for either equidistant nodes (`nod_type='equi'`) or Chebyshev nodes (`nod_type='cheb'`). In addition, one can choose for scaled up or scaled down variables, where scaled up variables are in the interval $[lb\,(i)\,, ub\,(i)]$ and scaled down variables in the interval $[-1, 1]$.

This function is called by `gridstruct`. When `grid_type='equi'` scaled up vectors with equidistant nodes are constructed, and when `grid_type='cheb'` both scaled down and scaled up vectors are constructed.

The inputs of the function are:

- qq: vector of number of gridpoints in each dimension ($1 \times n$ vector);
- nod_type: a string set to either `'equi'` for equidistant nodes, or `'cheb'` for Chebyshev nodes;
- $scale\_type$ (optional): a string which is either `'up'` (default for `nod_type='equi'`) or `'dw'` (default for `nod_type='cheb'`), referring to scaled up variables (taking values between lb and ub) or scaled down variables (taking values between -1 and 1), respectively;
- lb (required for `scale_type='up'`): vector of lower bounds in each dimension ($1 \times n$ vector);
- ub (required for `scale_type='up'`): vector of upper bounds in each dimension ($1 \times n$ vector).

The output of the function is:

- $gridVecs$: $1 \times n$ cell array containing the grid vector (either scaled up or scale down, depending on the $scale\_type$) in each dimension (the i-th cell contains a row vector of length $q(i)$).

Note that when `gridstruct` calls this function with `scale_type='dw'` then the output will assigned to $GRID.gridVecs\_dw$.

The function `constr_vecs` uses the functions:

- `chebnodes` (for `grid_type='cheb'`), which returns the Chebyshev nodes (see Section 10.14);
- `sc_cheb_dw` (for `scale_type='dw'`): see Section 10.13;
- `sc_cheb_up` (for `grid_type='cheb'` in combination with `scale_type='up'`): see Section 10.11.

10.6 Function `constr_grid`

This function constructs a grid, based on the Cartesian product of the n gridvectors in the cell array $gridVecs$. The output is an $m \times n$ matrix xx, where each column vector is a state variable, and each row represents a unique gridpoint. The function constructs the grid with Matlab's `ndgrid`, where each grid vector is expanded into a n dimensional array. These arrays are reshaped into column vectors, which are stacked next to each other.

The inputs of the function is:

- $gridVecs$: a $1 \times n$ cell array, as described in Section 10.5. It should be noted that $gridVecs$ can contain either scaled up or scaled down variables.

The output of the function is:

- xx: an $m \times n$ matrix where each column represents a state variable, and each row represents a unique gridpoint. Note that xx can be either scaled up or down [58].

10.7 Function `poly_elem_ani`

The function `poly_elem_ani` constructs the matrix *poly_elem* as explained in Section 7.2[59]. Each element $l_{i,j}$ in *poly_elem* refers to the univariate polynomial term $T_i(x_j)$ of order i for state variable j. These univariate polynomial terms $T_i(x_j)$ are found in the matrix Φ^j (see Section 10.8). The matrix *poly_elem* and the array Φ are used to construct the complete in the function `get_poly_ani`.

The inputs are:

- nn: the number of state variables;
- *ord_vec*: the maximum order of the polynomial in each dimension ($1 \times n$ vector).

The output is:

- *poly_elem*: an $p \times n$ matrix which is used to construct a complete polynomial. Each element $l_{i,j}$ refers to the univariate polynomial term $T_i(x_j)$ of order i for state variable j.

10.8 Function `constr_univar_basis`

The function `constr_univar_basis` constructs the univariate polynomial in matrix Φ^j up to order k, taking a vector x_j as input, as explained in Section 7.2.

The inputs are:

- xx: an $m \times 1$ vector of datapoints;
- *order*: the order of the univariate polynomial;
- *poly_type*: either `'cheb'` for Chebyshev polynomials of the first kind, or `'mono'` for monomials.

The output is:

- PHI: an $m \times order$ matrix, where each column i is the polynomial term $T_i(x)$. Note that $T_0(x)$ is not included.

[58] The labeling as either xx or xx_dw is done in the function `gridstruct`.

[59] The matrix *poly_elem* is called LL in that section

10.9 Function get_poly_ani

The function get_poly_ani constructs a complete polynomial of the grid, using either Chebyshev polynomials (poly_type='cheb') or monomials (poly_type ='mono'). The procedure is explained in Section 7.2, where Ω is the output XX_poly of the function get_poly_ani.

The inputs are:

- xx: an $m \times n$ matrix of gridpoints (either scaled up or scaled down);
- ord_vec: the maximum order of the polynomial in each dimension ($1 \times n$ vector);
- $poly\_type$: a string either 'cheb' or 'mono', which is an input for the function constr_univar_basis. For type='cheb' the Chebyshev polynomials of the first kind are used, for type='mono' monomials are used;
- $poly\_elem$: a $p \times n$ matrix to construct an anisotropic polynomial (see Section 10.7).

The output is:

- XX_poly: $m \times p$ matrix of the complete polynomial of $poly\_type$ for the grid xx.

Complete polynomials

An example of complete polynomials with two variables, x_1 and x_2, is as follows. For monomials the third order polynomial is:

$$XX\_poly = \left[\begin{array}{cccccccccc} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 & x_1^3 & x_1^2 x_2 & x_1 x_2^2 & x_2^3 \end{array}\right]$$

where x_1 and x_2 are column vectors (see function constr_grid in Section 10.6). The complete Chebyshev polynomial (of the first type) of order two is:

$$XX\_poly = \left[\begin{array}{cccccc} 1 & \tilde{x}_1 & \tilde{x}_2 & 2\tilde{x}_1^2 - 1 & \tilde{x}_1 \tilde{x}_2 & 2\tilde{x}_2^2 - 1 \end{array}\right] \tag{10.4}$$

where $\tilde{x}_1$ and $\tilde{x}_2$ are column vectors of the scaled down variables (see function constr_grid in Section 10.6).

10.10 Function scal_mat_up

The function takes a scaled down grid xx_dw and scales it up to output xx, using sc_cheb_up. The inputs are:

- xx_dw: scaled down grid ($m \times n$ matrix) (see Section 10.6)

- lb: vector of lower bounds in each dimension ($1 \times n$ vector);
- ub: vector of upper bounds in each dimension ($1 \times n$ vector);

The output is:

- xx: a is scaled up grid ($m \times n$ matrix), with each column i in xx_dw linearly transformed using `sc_cheb_up`, using bounds $lb\,(i)$ and $ub\,(i)$.

The function uses:

- `sc_cheb_up`, which is explained in Section 10.11.

10.11 Function `sc_cheb_up`

This function uses a linear transformation of a variable of the form $xx = (xx\_dw + 1)(ub - lb)/2 + lb$. This means a variable xx_dw with the basis interval $[-1, 1]$ is linearly mapped to the interval $[lb, ub]$[60]. This is the inverse transformation of the function `sc_cheb_dw`. The inputs are:

- lb and ub: the lower and upper bound (both scalars) of variable xx;
- xx_dw: an array of gridpoints for one variable on the interval $[-1, 1]$.

The output is:

- xx: an array of one variable with the same dimensions as xx_dw and scaled up to the interval $[lb, ub]$.

10.12 Function `scal_mat_dw`

The function takes a scaled up grid (xx) and scales it down (output xx_dw) using `sc_cheb_dw`. The inputs are:

- xx: scaled up grid ($m \times n$ matrix) (see Section10.13)
- lb: vector of lower bounds in each dimension ($1 \times n$ vector);
- ub: vector of upper bounds in each dimension ($1 \times n$ vector);

The output is:

- xx_dw: a is scaled down grid ($m \times n$ matrix) with each column in $xx\,(i)$ linearly transformed using `sc_cheb_dw`, using bounds $lb\,(i)$ and $ub\,(i)$.

The function uses:

- `sc_cheb_dw`, which is explained in Section 10.13.

[60] Values can be outside the interval $[-1, 1]$, which results in xx also being outside $[lb, ub]$.

10.13 Function `sc_cheb_dw`

This function linearly transforms a variable with the formula $xx\_dw = 2xx/(ub - lb) - (lb + ub)/(ub - lb)$. A variable xx based on the interval $[lb, ub]$ is linearly mapped to interval $[-1, 1]$[61]. This is the inverse transformation of the function `sc_cheb_up`. The inputs are:

- lb and ub: the lower and upper bound (both scalars) of variable xx;
- xx: an array (of any dimension) of gridpoints for one variable based on the interval $[lb, ub]$.

The output is:

- xx_dw: an array of one variable with the same dimensions as xx and scaled down to the interval $[-1, 1]$.

10.14 Function `chebnodes`

This function constructs a column vector of the Chebyshev nodes in the range $[-1, 1]$. The input is:

- dd: the number of nodes.

The output is:

- x: a column vector (dd x 1) of the Chebyshev nodes in the range $[-1, 1]$.

[61] Values can be outside the interval $[lb, ub]$, which results in xx also being outside $[-1, 1]$.

Part IV

Example models

11. Deterministic Brock-Mirman model

The Brock-Mirman model was used in Chapter 3 as a simple example. In this chapter we describe the derivation of the equations used there. The Brock-Mirman model is interesting, because the optimal solution can be derived analytically, even for the stochastic version. We used the deterministic version for simplicity reasons.

The agent in the Brock-Mirman model maximizes his discounted utility :

$$\max \sum_{t=1}^{\infty} \beta^{t-1} \log\left(C_t\right)$$

subject to:

$$K_{t+1} + C_t = K_t^{\alpha} \tag{11.1}$$

where C_t is consumption in period t, β is the discount factor, K_t is the capital stock *at the beginning* of the period, and K_t^{α} is the production function.

We rewrite the maximization problem in a infinite horizon Lagrangian:

$$\mathcal{L} = \sum_{t=1}^{\infty} \beta^{t-1} \left\{\log\left(C_t\right) + \lambda_t \left[K_t^{\alpha} - K_{t+1} - C_t\right]\right\}$$

where λ_t is the Lagrangian multiplier on the resource constraint. The solution is an infinite series for C_t, K_{t+1}, and λ_t. The sufficient First Order Conditions with respect to C_t, and K_{t+1} are:

$$\begin{aligned} \frac{1}{C_t} &= \lambda_t \\ \lambda_t &= \beta \lambda_{t+1} \alpha K_{t+1}^{\alpha-1} \end{aligned} \tag{11.2}$$

The second equation is referred to as the Euler equation, and characterizes the dynamic solution. We can substitute out λ using (11.2) and obtain:

$$\frac{1}{C_t} = \beta \frac{1}{C_{t+1}} \alpha K_{t+1}^{\alpha-1} \tag{11.3}$$

Analytical solution

The model has an analytical solution, which is:

$$C_t = (1 - \alpha\beta)\, K_t^{\alpha}$$

With this policy function next period's capital stock is:

$$\begin{aligned} K_{t+1} &= K_t^{\alpha} - (1 - \alpha\beta)\, K_t^{\alpha} \\ &= (\alpha\beta)\, K_t^{\alpha} \end{aligned}$$

Substituting this into the The Euler equation yields:

$$\frac{1}{(1 - \alpha\beta)\, K_t^{\alpha}} = \beta \frac{1}{(1 - \alpha\beta)\, [(\alpha\beta)\, K_t^{\alpha}]^{\alpha}} \alpha\, [(\alpha\beta)\, K_t^{\alpha}]^{\alpha - 1}$$

which proofs that both equations are satisfied for the given solution.

Steady state

From the Euler equation (11.3) we derive steady state capital:

$$\overline{K} = [\alpha\beta]^{\frac{1}{1-\alpha}}$$

and from the resource constraint (11.1) we derive steady state consumption:

$$\overline{C} = \overline{K}^{\alpha} - \delta \overline{K}$$

12. Standard RBC model

In Chapter 4 we used a standard Real Business Cycle (RBC) as an example. In this chapter we derive the equations used in that chapter. This includes the computation of the expected value using Gauss-Hermite quadrature.

12.1 Model

A standard Real Business Cycle (RBC) model with a representative agent is a dynamic model where the agent has to determine how much to work, consume and invest. Hours worked gives disutility, consumption gives instant positive utility, while investment increases future capital income.

The objective function of the agent is:

$$\max E_1 \sum_{t=1}^{\infty} \beta^{t-1} \left\{ \frac{C_t^{1-\nu}}{1-\nu} - \chi \frac{H_t^{1+\frac{1}{\eta}}}{1+\frac{1}{\eta}} \right\}$$

where C_t is period t consumption, and H_t is period t labor supply. The real budget constraint is:

$$C_t + K_{t+1} = Z_t K_t^{\alpha} H_t^{1-\alpha} + (1-\delta) K_t \tag{12.1}$$

where K_t is the capital stock at the beginning of period t, and δ is the depreciation rate of capital. Total Factor Productivity (TFP) Z_t evolves by an exogenous process:

$$z_t = \rho_z z_{t-1} + \sigma_z \epsilon_t \tag{12.2}$$

where $z_t = \log(Z_t)$, ρ_z is the autocorrelation coefficient, and σ_z is the standard deviation of the shocks. The shocks ϵ_t are standard normally distributed ($\epsilon_t \sim \mathcal{N}(0,1)$).

The optimization problem can be written with an infinite Lagrangian:

$$\mathcal{L} = E \sum_{t=1}^{\infty} \beta^t \left\{ \frac{C_t^{1-\nu}}{1-\nu} - \chi \frac{H_t^{1+\frac{1}{\eta}}}{1+\frac{1}{\eta}} + \lambda_t \left[Z_t K_t^{\alpha} H_t^{1-\alpha} + (1-\delta) K_t - C_t - K_{t+1} \right] \right\}$$

where λ_t is the shadow price of the budget constraint, and K_1 is given. Maximization of this Lagrangian with respect to hours H_t, consumption C_t and capital in next period K_{t+1} yields the following First Order Conditions:

$$\begin{aligned}
C_t^{-\nu} &= \lambda_t \\
\chi H_t^{\frac{1}{\eta}} &= \lambda_t Z_t (1-\alpha) K_t^{\alpha} H_t^{-\alpha} \\
\lambda_t &= \beta \lambda_{t+1} \left[F_k (K_t, H_t) + 1 - \delta \right]
\end{aligned}$$

Substituting out λ gives:

$$\chi H_t^{\frac{1}{\eta}} = C_t^{-\nu} Z_t (1-\alpha) K_t^{\alpha} H_t^{-\alpha} \tag{12.3}$$

$$C_t^{-\nu} = \beta E_t \left\{ C_{t+1}^{-\nu} \left[Z_{t+1} \alpha K_{t+1}^{\alpha-1} H_{t+1}^{1-\alpha} + 1 - \delta \right] \right\} \tag{12.4}$$

We can derive an analytical expression for labor supply, given capital, TFP and consumption using (12.3):

$$H_t = \left[\frac{1-\alpha}{\chi} C_t^{-\nu} Z_t K_t^{\alpha} \right]^{\frac{\eta}{1+\alpha\eta}} \tag{12.5}$$

12.2 Gauss-Hermite quadrature

The general rule for Gaussian-Hermite approximation is:

$$\int_{-\infty}^{\infty} \exp\left(-z^2\right) g(z)\, dz \approx \sum_{j=1}^{J} \omega_j g(\zeta_j) \tag{12.6}$$

with Gauss-Hermite nodes $j = 1, ..., J$, roots ζ_j and weights ω_j (see Judd, 1998).

Assume we have a function $f(z_{t+1}, x)$ with exogenous variable z_{t+1}. This variable evolves according to (12.2) with standard normally distributed shocks $\epsilon_{t+1} \sim \mathcal{N}(0, 1)$. The expected value of this function is:

$$E_t f(z_{t+1}, x) = \int_{-\infty}^{\infty} f(\rho_z z_t + \sigma_z \epsilon_{t+1}, x) \frac{1}{\sqrt{2\pi}} \exp\left(-\epsilon_{t+1}^2/2\right) d\epsilon_{t+1} \tag{12.7}$$

To write (12.7) in the same form as (12.6) we need a change of variable $\phi = \frac{\epsilon_{t+1}}{\sqrt{2}}$, such that $\exp\left(-\epsilon_{t+1}^2/2\right) = \exp\left(-\phi^2\right)$. The approximation of the integral is:

$$\int_{-\infty}^{\infty} f\left(\rho_z z_t + \sigma_z \sqrt{2}\phi, x\right) \frac{1}{\sqrt{2\pi}} \exp(\phi) \sqrt{2} d\phi$$

$$\approx \sum_{j=1}^{J} \frac{\omega_j}{\sqrt{\pi}} f\left(\rho_z z_t + \sigma_z \sqrt{2}\zeta_j, x\right)$$

where the extra term $\sqrt{2}$ (before $d\phi$) follows from integration by substitution.

12.3 Steady state

To derive the analytical steady state we start with the Euler equation (12.4):

$$\begin{aligned}\overline{C}^{-\nu} &= \beta \left\{ \overline{C}^{-\nu} \left[\overline{Z} \alpha \overline{K}^{\alpha-1} H^{1-\alpha} + 1 - \delta \right] \right\} \\ \overline{H} &= \left[\frac{1 - \beta(1-\delta)}{\overline{Z}\alpha\beta} \right]^{\frac{1}{1-\alpha}} \overline{K} = \Omega^{\frac{1}{1-\alpha}} \overline{K}\end{aligned}$$

with $\Omega = \frac{1-\beta(1-\delta)}{\alpha\beta\overline{Z}}$.

Substituting this into the resource constraint (12.1) yields:

$$
\begin{aligned}
\overline{C}+\overline{K} &= \overline{Z}\overline{K}^{\alpha}\overline{H}^{1-\alpha}+(1-\delta)\,\overline{K} \\
\overline{C} &= \overline{Z}\overline{K}^{\alpha}\left[\Omega^{\frac{1}{1-\alpha}}\overline{K}\right]^{1-\alpha}-\delta\overline{K} \\
&= \left(\overline{Z}\Omega-\delta\right)\overline{K}
\end{aligned}
$$

Substituting the expressions for $\overline{H}$ and $\overline{C}$ into the labor supply function (12.3) and solving for $\overline{K}$ yields:

$$
\overline{K}=\left[\left(\frac{1-\alpha}{\chi}\overline{Z}\left[\overline{Z}\Omega-\delta\right]^{-\nu}\right)^{\eta}\Omega^{\frac{\alpha\eta+1}{\alpha-1}}\right]^{\frac{1}{1+\eta\nu}}
$$

13. Housing model

In Chapter 5 we used an RBC model with housing to demonstrate how to solve a model with two policy variables. In this chapter we describe the model and the derivations of the equations.

Model

The objective of the agent is:

$$
\max E_1 \sum_{t=1}^{\infty}\beta^{t-1}\left[U\left(C_t\right)+V\left(D_t\right)\right]
$$

where C is consumption, and D is housing. The agent maximizes the objective subject to the budget constraint:

$$
C_t+K_{t+1}+D_{t+1}\leq Z_tK_t^{\alpha}+\left(1-\delta_k\right)K_t+\left(1-\delta_d\right)D_t \tag{13.1}
$$

The First Order Conditions for C_t, K_{t+1}, and D_{t+1} are:

$$
\begin{aligned}
U'\left(C_t\right) &= \lambda_t \\
\lambda_t &= \beta E_t\left\{\lambda_{t+1}\left[Z_{t+1}\alpha K_{t+1}^{\alpha-1}+1-\delta_k\right]\right\} && (13.2)\\
\lambda_t &= \beta E_t\left\{V'\left(D_{t+1}\right)+\lambda_{t+1}\left(1-\delta_d\right)\right\} && (13.3)
\end{aligned}
$$

We use the functional form:

$$
\begin{aligned}
U\left(C_t\right) &= \frac{C_t^{1-\nu}-1}{1-\nu} \\
V\left(D_t\right) &= \varrho\frac{D_t^{1-\eta}-1}{1-\eta}
\end{aligned}
$$

Steady state

The Euler equation for capital (13.2) is standard and yields:

$$
\begin{aligned}
1 &= \beta\left(\overline{Z}\alpha\overline{K}^{\alpha-1} + 1 - \delta_k\right) \\
\overline{K} &= \left(\frac{\overline{Z}\alpha\beta}{1-\beta\left(1-\delta_k\right)}\right)^{\frac{1}{1-\alpha}}
\end{aligned}
$$

From the Euler equation for housing (13.3) we derive:

$$
\begin{aligned}
V'\left(\overline{D}\right) &= \overline{\lambda}\frac{1-\beta\left(1-\delta_d\right)}{\beta} \\
\varrho\overline{D}^{-\eta} &= \overline{C}^{-\nu}\frac{1-\beta\left(1-\delta_d\right)}{\beta} \\
\overline{D} &= \left(\frac{1-\beta\left(1-\delta_d\right)}{\varrho\beta}\right)^{\frac{1}{-\eta}}\overline{C}^{\frac{\nu}{\eta}}
\end{aligned}
$$

And finally from the budget constraint:

$$
\begin{aligned}
\overline{C} + \delta_k\overline{K} + \delta_d\overline{D} &= \overline{ZK}^{\alpha} \\
\overline{C} + \delta_d\left(\frac{1-\beta\left(1-\delta_k\right)}{\varrho\beta}\right)^{\frac{1}{-\eta}}\overline{C}^{\frac{\nu}{\eta}} &= \overline{ZK}^{\alpha} - \delta_k\overline{K}
\end{aligned}
$$

We solve for $\overline{C}$ and $\overline{D}$ numerically using a non-linear equation solver.

Bibliography

De Boor, Carl (1978). *A practical guide to splines*. Vol. 27. springer-verlag New York.

Duineveld, Sijmen (2021). “Standardized projection algorithms to solve dynamic economic models”. Manuscript submitted for publication.

Fernández-Villaverde, Jesús, Juan Francisco Rubio-Ramírez, and Frank Schorfheide (2016). “Solution and estimation methods for DSGE models”. In: *Handbook of macroeconomics*. Vol. 2. Elsevier, pp. 527–724.

Galizia, Dana (2018). *Saddle Cycles: Solving Rational Expectations Models Featuring Limit Cycles (or Chaos) Using Perturbation Methods*. Tech. rep. Carleton University, Department of Economics.

Gaspar, Jess and Kenneth L Judd (1997). “Solving large-scale rational-expectations models”. In: *Macroeconomic Dynamics* 1.1, pp. 45–75.

Heer, Burkhard and Alfred Maussner (2009). *Dynamic general equilibrium modeling: computational methods and applications*. Springer Science & Business Media.

Judd, Kenneth L (1992). “Projection methods for solving aggregate growth models”. In: *Journal of Economic theory* 58.2, pp. 410–452.

— (1998). *Numerical methods in economics*. MIT press.

Judd, Kenneth L et al. (2014). “Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain”. In: *Journal of Economic Dynamics and Control* 44, pp. 92–123.

Miranda, Mario J and Peter G Helmberger (1988). “The effects of commodity price stabilization programs”. In: *The American Economic Review*, pp. 46–58.