Centre for International Macroeconomic Studies
School of Economics, University of Surrey

An Advanced Course On The Science and Art of DSGE Modelling*

Hyungseok Joo
Ricardo Nunes
Donghyun Park

September 2021

*This note is based on Antonio Mele's lecture notes. The authors would like to thank Antonio Mele who kindly allowed us to use his lecture note.

CONTENTS

LISTINGS

1 INTRODUCTION

1.1 INTRODUCTION

This advanced course is aimed at researchers who are already fluent in Dynare and MATLAB, but are finding that their ambition currently exceeds what they are able to do with these tools. This course will be useful to anyone who is engaged in practical macroeconomic modelling work, especially if they are interested in working with models that are either computationally expensive to solve and simulate, highly nonlinear, or infinite dimensional due to heterogeneous agents.

Taken over the four days the participants will be guided through a series of sessions covering a wide range of advanced topics:

1. Applied Dynamic Programming and Global Solution Algorithms

2. Classical Heterogeneous Agents Models and Occasionally Binding Constraints

3. Central Bank Communication, Imperfect Credibility, and Optimal Monetary Policy Application

4. Modern Topics on Heterogeneous Agents and Wealth Accumulation

These notes refer to topics 1. Please refer to the OneDrive for more details.

1.2 READINGS

For the lectures on global solution methods, we suggest you look at Ljungqvist and Sargent (2012) and Stokey et al. (1989) for the theory. One very good book about numerical methods is Judd (1998). Some codes make use of the CompEcon Toolbox which accompanies Miranda and Fackler (2004), hence the reader should refer to it for details.

1.3 CODE

All of the course's code is available from OneDrive. If anything goes wrong, please e-mail m.alvaro-taus@surrey.ac.uk. Where blocks of code are reasonably short, they will be provided in the main body of this text, but for longer pieces of code you are referred to this website. Where possible, the PDF version of this document will contain links to the mentioned files.

1.4 INSTRUCTORS

The sessions will be taught by researchers working at The Centre for International Macroeconomic Studies (CIMS) in the School of Economics, University of Surrey.

Szabolcs Deák is a Lecturer at the University of Exeter since 2019. He was previously a full-time Research Fellow in the School of Economics at the University of Surrey. He joined the School in September 2013 to support the research activities of the ESRC funded project "Agent-Based and DSGE Macroeconomic Models: A Comparative Study". He received his Masters degree in Economics from the University of Szeged (Hungary) in 1999 and worked there as a full-time Lecturer from 1999 to 2005. He went on to study Economics at post-graduate level from 2006 at Bocconi University (Milan, Italy), receiving his PhD in 2011. He previously worked at the Monetary Policy Research Division of the European Central Bank (Frankfurt am Main, Germany) and held a Jean Monnet Postgraduate Fellowship at the European University Institute (Florence, Italy). For further details of publications see IDEAS-RePEc.

Hyungseok Joo is a Lecturer in the School of Economics at the University of Surrey. He graduated from the Yonsei University (Seoul, Republic of Korea). He then completed his MS in Economics at University of Wisconsin, Madison, then he went to Boston University obtaining a Ph.D. in Economics in 2015. He joined the University of Surrey as Lecturer in 2019, after having spent years of research and teaching at Wayne State University, USA.

Ricardo Nunes is a Professor in the School of Economics at the University of Surrey. He graduated from Universitat Pompeu Fabra (Barcelona, Spain) obtaining a MSc in Economics in 2003 and a PhD in Economics in 2007. After graduating he spent 10 years in the Federal Reserve System under various roles. In 2007 he joined the Board of Governors of the Federal Reserve System, where he worked as an economist and senior economist. In 2014 he moved to

the Federal Reserve Bank of Boston working as a senior economist and policy advisor. He was also a visiting researcher at the Bank of Portugal and the IMF and has given talks at various central banks. His main research is on monetary and fiscal policy. He has published extensively in these areas including the Quarterly Journal of Economics, Journal of Monetary Economics, Journal of Economic Theory, Journal of European Economic Association, among others.

1.5 COURSE OUTLINE

The lectures over the four days will cover topics including:

- **Day 1: Applied Dynamic Programming: Global Solution Algorithms**
  - The basic theory of dynamic programming.
  - Numerical integration.
  - Function approximation.
  - Value function iteration.
  - Policy function iteration.
  - Projection methods.
  - Exercises in Lab.

2 DYNAMIC PROGRAMMING AND PROJECTION METHODS

2.1 INTRODUCTION

Most macroeconomic models have a **recursive** structure, and they can be represented as a **functional equation** in some functional space. In this part of the course we will look at the basic dynamic programming theory, and its practical implementation. This is the more general approach for solving dynamic models in macroeconomics. In particular, it **always** works: given enough time, the algorithms that we will see (value function iteration and policy function iteration) converge to the solution of the model at hand. The crucial words are "given enough time": these algorithms can be very slow also for relatively simple models.

We will then look at techniques that are designed to solve functional equations in general, called **projection methods**. This approach is much faster than standard algorithms based on basic dynamic programming results. However, these algorithms do not have clear convergence properties, and many times the researcher has to "help" the code to converge by using some tricks. In some sense, working with projection methods is more an art than an exact science.

We will start with a review of the basics of dynamic programming. We will then study MATLAB implementations of the value and policy function iteration algorithms in the simplest dynamic macroeconomic model you can think of: stochastic growth model with 2-state, iid productivity shocks. We will look at numerical techniques that are generally used to improve the performance of the simplest dynamic programming algorithm: numerical integration, solution of non-linear equations and optimization. We will then present projection methods, and apply these techniques to a slightly more complicated model (continuous support for productivity shocks, allowing for persistence), and compare the performance with the basic algorithms.

2.2 DYNAMIC PROGRAMMING

We will approach the dynamic programming techniques as the basic way of solving a macroeconomic model. In fact, every other technique (including log-linearization, perturbation, etc.) is taking the recursivity of the problem as given, and using it to solve it in a simpler way.

However, some frameworks can be particularly "hostile" to the researcher, and simplified algorithms that use some specific properties of the model cannot be implemented. In these cases, it is **always** possible to apply basic implementations of the value function iteration (VFI) algorithm and complemented it with other approaches. You will see examples of this mixed approach in section **??**.

To give you a few examples in which dynamic programming can be the only feasible strategy, here are two ideas:

1. Strong presence of non-linearities can induce a very different behaviour of the system when far from the steady state. This is true for example in models with ZLB, as shown in the work of Braun et al. (2012).

2. Non-convexities imply that first order conditions are not sufficient for a solution. This is true for example in heterogeneous agents models with discrete choices in labour supply (work/not work decision) as in Chang and Kim (2007).

For these problems, global techniques are more suited, and VFI is a very powerful algorithm, since it can potentially solve any problem that can be stated as a Bellman equation. However, it rapidly becomes computationally burdensome as the number of state variables increases. There are many tricks to deal with this problem, and VFI can be combined with other numerical procedures that would help speed up the algorithm (interpolation, projections methods, etc.).

VFI is appropriate for models in which time inconsistency is not present. However, many authors have worked on extending the same idea of dynamic programming to time inconsistent problems (see for example Marcet and Marimon (2011)). With some modifications, the same algorithms can be used also for optimal policy problems and New Keynesian models, bearing in mind that the number of endogenous states couls be a source of troubles. For the purpose of this course we will present the technique with a very simple RBC model, but the reader should bear in mind that the application of this method is much broader.

2.2.1 The Basics of Dynamic Programming

Dynamic programming became the standard method in macroeconomics during 70s. Currently, all macroeconomists use models that can be solved with the tools of dynamic programming. In order to explain the main ideas of it, we are going to use a stochastic, basic RBC model and solve it with the simplest implementation of the value function iteration algorithm. To fix ideas, let us shut down uncertainty for the moment, since everything we say will carry on in the presence of uncertainty under the assumption of Markovian exogenous shocks.

As it is well known, a basic RBC model can be solved by looking at the competitive equilibrium of the economy. However, since there are no market failures, the competitive equilbrium is Pareto efficient, and therefore allocations can be more easily found by solving a planner problem. This is the strategy that we will follow here. Another strategy would be to solve both the firms' and the agents' problems as dynamic programs and then find the equilibrium of the economy. In many cases in which the economy presents distortions, this is the way to go, however in the simplest RBC model they are equivalent.

Assume then that the production function is

$$Y_t = F(A_t, L_t, K_{t-1}) \tag{2.1}$$

where A_t is the productivity shock, L_t is hours supplied, and K_{t-1} is physical capital inherited from the previous period. The utility function of the representative household is

$$U_t = U(C_t, L_t) \tag{2.2}$$

In order to solve it, we will make the following simplifying assumptions:

Assumption 1 (Utility function). *$U(C_t, L_t) = U(C_t)$, $U : R_+ \to R$ is bounded, continuously differentiable, strictly increasing, strictly concave and $\lim_{C\to 0} U'(C) = \infty$.*

This assumption guarantees that there is no disutility of work, and therefore labor supply is equal to the labor endowment (normalized to 1). The rest of assumption 1 guarantees interiority and uniqueness of the solution.

Assumption 2 (Production function). *The production function $F : R_+^3 \to R_+$ is continuously differentiable, strictly increasing, homogeneous of degree 1 and strictly quasi-concave, with*

$$F_K > 0; F_{KK} < 0; F_L > 0; F(A, L, 0) = 0 \qquad \forall K, L > 0$$

$$\lim_{K\to 0} F_K(A, 1, K) = \infty, \lim_{K\to\infty} F_K(A, 1, K) = 0 \qquad \text{(Inada conditions)}$$

Assumption 2 is useful in making the problem "well-behaved", i.e. the constraint set does not have any pathology. We also assume for the moment that the productivity A_t is constant: $A_t = A$.

Finally, the resource constraint of the economy is

$$C_t + K_t - (1-\delta) K_{t-1} \leq F(A, 1, K_{t-1}) \tag{2.3}$$

where investment is represented by $I_t = K_t - (1-\delta) K_{t-1}$. For simplicity, there is no government spending in this baseline model.

We can therefore consider the problem of a benevolent social planner with the objective of maximizing the expected discounted utility of the representative household by choosing sequences $\{C_t, K_t\}_{t=0}^{\infty}$, subject to the resource constraint (2.3), and taking K_{-1} as given. Notice that it can never be optimal to waste output, therefore equation (2.3) holds with equality. We can therefore use it to eliminate consumption from the problem. Define

$$f(K) \equiv F(A, 1, K) + (1-\delta) K$$

and rewrite the problem as[1]:

$$\max_{\{K_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t U\left(f(K_{t-1}) - K_t\right) \tag{2.4}$$

$$s.t. \quad 0 \leq K_t \leq f(K_{t-1}), \quad t = 0, 1, ... \tag{2.5}$$

$$K_{-1} \text{ given}$$

Assume for a moment that we want to solve this problem over a finite horizon T. In this case, we have to find a sequence $\{K_t\}_{t=0}^{T}$ that solves problem 2.4. This is a standard concave program, since the set of sequences $\{K_t\}_{t=0}^{T}$ that satisfy (2.5) is a closed, bounded and convex subset of $\mathbb{R}^{T+1}$, and the objective in (2.4) is continuous and strictly concave, therefore there is a unique solution and we can characterize it with Kuhn-Tucker conditions. Since $f(0) = 0$ and $U'(0) = \infty$, constraint (2.5) only binds for K_T, and of course $K_T = 0$. Therefore we can get the following first-order conditions:

$$\beta f'(K_t) U'\left(f(K_t) - K_{t+1}\right) = U'\left(f(K_{t-1}) - K_t\right) \tag{2.6}$$

which is a standard Euler equation. Taking into account the boundary conditions

$$K_t = 0, \quad K_{-1} > 0 \text{ given}$$

[1]The constraint comes from the fact that we want consumption to be non-negative.

we can solve for the optimal sequence.

Now, let's go back to infinite horizon. How do we solve for the optimal (infinite!) sequence? A natural way is to solve for the finite horizon case as we just did, and then take the limit of the solution as T goes to infinity. However, this approach will work only under very specific circumstances. In particular, first order conditions must be sufficient for characterizing the solution. We thus follow a different approach due to Richard Bellman, which is more general and works for most problems in macroeconomics.

One reasonable guess is that a solution must take the form $K_t = g(K_{t-1})$ with $g : R_+ \to R_+$. What is the intuition? In each period, the planning problem is always the same, and only the capital that we have at the beginning of each period changes. Hence, the choice of future capital stock and consumption must be a function of current capital stock. How do we get the function g?

Imagine for a second that we already solved the program (2.4) for all possible values of initial capital K_{-1}. We can define a function $V : R_+ \to R$ by saying that, for any possible $K_{-1} > 0$ given,

$$V(K_{-1}) \equiv \max_{\{K_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t U\left(f(K_{t-1}) - K_t\right)$$
$$s.t. \quad 0 \leq K_t \leq f(K_{t-1}), \quad t = 0, 1, ...$$

The function V is called the *value function* of the problem 2.4. If instead of starting with initial capital K_{-1}, we start with capital equal to K_0, then $V(K_0)$ is the value of the discounted flow of utility under optimal choices, obtained with an initial capital K_0. Given that the problem is identical in period 0 and in period 1, we can rewrite the planner problem at time 0 as:

$$\max_{K_0} \left[U\left(f(K_{-1}) - K_0\right) + \beta V(K_0)\right] \tag{2.7}$$
$$s.t. \quad 0 \leq K_0 \leq f(K_{-1})$$
$$K_0 \geq 0, \quad K_{-1} > 0 \quad \text{given}$$

If we knew V, we could get g from (2.7), by defining $g : R_+ \to R_+$ as follows: let $K_0 = g(K_{-1})$ be the maximizer in (2.7), and consequently $C_0 = f(K_{-1}) - g(K_{-1})$. Given that the problem is identical in every period t, the function $K_t = g(K_{t-1})$ completely describes the dynamics for $t = 0, 1, ...$ for our problem. However, we don't know V! What can we do?

Remember that V is the maximized objective function for program (2.4). Therefore, if program (2.7) also solves the same problem, it must be that

$$V(K_{-1}) = \max_{0 \leq K_0 \leq f(K_{-1})} \left\{U\left(f(K_{-1}) - K_0\right) + \beta V(K_0)\right\}$$

Under this formulation we don't need time subscripts, since this is a static problem! We can therefore write it as

$$V(K) = \max_{0 \leq y \leq f(K)} \left\{ U\left(f(K) - y\right) + \beta V\left(y\right) \right\} \tag{2.8}$$

This is an equation in which the unknown is a function (the value function V), and this is the reason why it is called *functional equation*[2]. The solution therefore is a function. It may seem that transforming the initial sequential problem (in which the solution is a infinite sequence) into a functional equation complicates the analysis. However, this is not the case: it turns out that it is much easier to solve the functional equation. Once we solve the equation, we get also a maximizer function $y = g(K)$. We can use this function to calculate the optimal sequence of capital accumulation starting from K_{-1}.

Notice the important properties of the Bellman equation. First of all, what is telling you is that, in each period, all the information about the past is contained in the current level of capital. This is why we can eliminate time subscripts. We call this property **stationarity**. The variables that summarize the information about the past are called **state variable(s)**.

Equivalence of Sequential and Functional Equation Solutions

We have established that we can analyze problems like (2.4) by solving a functional equation like (2.8). However, we need to be sure that solutions of the sequential problems are also solutions of the functional equation. We need to establish if a solution exists and if it is unique. Moreover, we need a method to solve for the optimal V (and g of course). This is a complicated mathematical issue, and we are not going to dive into the details, but the interested reader can check Stokey et al. (1989) and Miao (2014).

The equivalence can be established by using the Contraction Mapping Theorem (CMP). This theorem shows that, under general assumption that our model satisfies, there is a unique continuous, increasing and concave value function V that solves the Bellman equation. Moreover, the policy function g is also continuous.

This result can be easily generalized to cases with more than one endogenous state variable. The reader is advised to read Stokey et al. (1989) and Miao (2014) for a detailed treatment of this case.

[2]This equation is often referred to as the *Bellman equation*, in honour of Richard Bellman who formulated it in 1957.

Stochastic Models

Everything we said applies to stochastic models where the shocks A_t follows a Markov process. The mathematics behind this extension is quite complicated and involves measure theory, but as before we are going to skip all tedious technical details. We can easily apply the same approach for the stochastic case, by making a few assumptions about the probability distribution we use.

For our purposes, we will just state the extended problem and claim that we can use functional equations to solve it. Again, the reader interested in the details can refer to Stokey et al. (1989) and Miao (2014). For simplicity, we can assume that the possible realizations of the shock A_t are a finite number: $A_t \in \left\{\widehat{A_1}, ..., \widehat{A_S}\right\}$. This is very restrictive and absolutely not needed: we can have countable, or continuous shocks. What is needed is instead a crucial assumption about the probability distribution of A_t:

Definition 1. *A stochastic process $\{A_t\}$ is said to have the **Markov property** if for all $j \geq 1$ and all t ,*

$$Prob(A_{t+1}|A_t, A_{t-1}, ..., A_{t-j)} = Prob(A_{t+1}|A_t).$$

We assume the vector of shocks $\{A_t\}$ satisfies the Markov property. Indicate the conditional probabilities of state A' tomorrow, given state A today, with $\pi\left(A'|A\right)$. We can basically repeat the whole analysis in the previous section and prove that we can solve the sequential problem by analyzing a slightly different version of the Bellman equation:

$$V\left(K, A\right) = \max_{0 \leq y \leq f(K,A)} U\left(K, y, A\right) + \beta \sum_{A'} \pi\left(A'|A\right) V\left(y, A'\right) \qquad \forall \left(K, A\right) \quad (2.9)$$

2.2.2 The Value Function Iteration (VFI) Algorithm

For our purpose, the most important result related to CMP is an algorithm to solve for the value function. In particular, it is possible to show that, starting from an arbitrarily chosen value function, and applying iteratively the Bellman equation to get a new value function, we converge to the true value function. The convergence is pointwise, i.e. for every point in the state space we converge to the true value function for that point. This procedure is called **value function iteration** and it is the basic numerical method that economists use to solve a Bellman equation.

In this section we show how we can write a simple MATLAB code to solve for the value function by using VFI. In the following sections we describe how we can deal with more complicated models and the limits of this technique.

VFI on a Computer

The CMP guarantees that the VFI converges to the unique value function that solves the problem. In particular, it is possible to show that, given a guess for the value function, we always converge to the true value function by iterating on the Bellman equation. Notice that the initial guess can be arbitrary, but of course if we start with a good guess (i.e., a guess close to the solution), we can converge faster to the true value function. Therefore in many situations it might be useful to find a good initial guess by using some of the local techniques studied in this course.

First of all, we cannot work in a PC with continuous variables: we need to discretize the number of possible choices we have. We therefore define a grid $G \equiv \{x_1, ..., x_m\}$ (where each point x_i is a combination of K and A in our simple RBC model), and this will be our state space. We can therefore proceed in the following way:

1. We form an initial guess for the value function for each point on the grid: $V_i^0 = V^0(x_i)$
2. For any $n \geq 0$, get a new guess for the value function by computing $V^{(n+1)}(\cdot) = TV^{(n)}(\cdot)$ for any point in the grid, where the operator T is the so called **Bellman operator**:

$$(T)(V(K,A)) = \max_{0 \leq K' \leq f(K,A)} U(f(K,A) - K') + \sum_{A'} \pi\left(A'|A\right) V\left(K', A'\right) \tag{2.10}$$

3. Stop if $\left\| V^{(n+1)} - V^{(n)} \right\| < \epsilon$. Otherwise, go back to 1.

The algorithm looks pretty simple, but you will soon realize that bringing an algorithm to the code is not so easy sometimes. First of all, we need to make assumptions on the various functional forms that we want to use. In the basic RBC model, for example, we must specify the utility function and the production function. We also need to choose the parameters of the model: in the RBC model, we must choose a value for the discount factor β, the parameters of the utility function, and the depreciation rate of capital. Another crucial parameter is the criterion for convergence (the ϵ above).

After these steps, we also need to define the grid on which we look for the solution. This can also be a complicated task: in many cases with more than one state variable, restricting yourself to look for a solution on a very small grid is useful as a start. Typically, a good idea is to generate a grid that includes the deterministic steady state among the grid points. However, in some case you might be interested on what happens when you are far from the steady state[3] and therefore you may need a very wide grid. This method is quite robust and usually wide grids do not impose much of a burden.

Since we are going to use MATLAB, we need to develop a code that exploits the main advantages of the software while avoiding its drawbacks. MATLAB is not very good in loops, because it works with a structure of matrices and vectors, therefore it is very important to try to exploit this feature of the software. I will provide a way to write the code in matrix terms, which closely follows Ljungqvist and Sargent (2012), chapter 4.

Assume for simplicity that we have only two possible values for the productivity shock $[A_1, A_2]$. We store the transition probabilities of these two realizations in a matrix $\mathcal{P}$, where for example $\mathcal{P}_{12} \equiv \pi(A_2|A_1)$. Let there be n grid points for capital $[k_1, k_2, ..., k_n]$. Define two matrices U_j such that:

$$U_j(i,h) = U\left(f\left(k_i, A_j\right) - k_h\right), \quad i = 1, ..., n, \quad h = 1, ..., n$$

These matrices store the value of utility of consumption for any possible combination of capital today and tomorrow.

Define also two $n \times 1$ vectors $V_j, j = 1, 2$ such that $V_j(i) = V_j\left(k_i, A_j\right), i = 1, ..., n$. Let $\mathbf{1}$ be a $n \times 1$ vector of ones. We can define an operator $T\left([V_1, V_2]\right)$ that maps couples of vectors $[V_1, V_2]$ into a couple of vectors:

$$TV_1 = \max\{U_1 + \beta\mathcal{P}_{11}\mathbf{1}V_1^{'} + \beta\mathcal{P}_{12}\mathbf{1}V_2^{'}\}$$
$$TV_2 = \max\{U_2 + \beta\mathcal{P}_{21}\mathbf{1}V_1^{'} + \beta\mathcal{P}_{22}\mathbf{1}V_2^{'}\}$$

We can write these equations in compact form:

$$\begin{bmatrix} TV_1 \\ TV_2 \end{bmatrix} = \max\left\{ \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} + \beta\left(\mathcal{P} \otimes \mathbf{1}\right) \begin{bmatrix} V_1^{'} \\ V_2^{'} \end{bmatrix} \right\} \tag{2.11}$$

where $\otimes$ is the Kronecker product. We can solve by iterating on the operator T until convergence.

[3] For example, when you want to analyze what happens after big shocks that bring the economy far from the steady state. This is important when analyzing deep recessions or sovereign defaults episodes where GDP might fall by two-digit figures.

The Code

In this section we go step by step through the code. It is assumed the reader has a basic knowledge of MATLAB fundamentals. The code is named `vfi_AM.m`. We use log utility of consumption, Cobb-Douglas production function, and we assign standard values to parameters. We choose a wide grid in order to show that the method does not pose particular problems in this case. Also, notice that all the matrices in the code are transposed with respect to the previous section (this is faster due to the way in which MATLAB operates).

The file `parameters.m` sets the parameters values, and the grid for capital (stored in the column vector `kgrid`). Before executing the main file `vfi_AM.m` we ALWAYS need to call this script, otherwise we get an error message (given that some parameters would not be defined). We store parameters and the grid in this file so that later on it's easier to do comparative statics.

The file `vfi_AM.m` generates the big matrices U_j. We first calculate consumption allocation implied by all the possible combinations of K and K', for each value of the realization of the shock. Every column is therefore consumption implied by any possible choice for K', given a particular value for K. Notice that some values for consumption will be negative, therefore we can set them up to `NaN` to make sure that when we take the logs we don't get an error message. We then calculate utility of consumption by taking logs and we set utility for those consumption values that were negative at minus infinity (in this way, the max operation will never consider them to be optimal). We also clear variables that are not needed and take lots of memory space (this is not true in this model, however it might be important in more complicated setups with many state variables).

The following lines initialize variables in order to make space in the memory for them. The initial guess for the value function is zero for every gridpoint.

The main loop (a `while` loop) iterates over the Bellman equation. It uses the max operator to find the K' in the grid that maximizes the value for the agent, taking the value function from the previous iteration as given. The max operator delivers two numbers for each realization of A_t: the first (for example, `tv1`) is the value of the maximized object, the second (`tdecis1`) is the position of the maximizer in the grid (i.e., if the value of `tdecis1` is 100, it means that the maximizer is the 100th value in `kgrid`). Notice that we don't need to modify the matrices `util1` and `util2` at each iteration. We calculate `metric` as the *sup norm*, which is our concept of distance for determining convergence. We keep iterating until `metric` becomes small enough. We also save the value of `metric` in a vector for generating a graph of the convergence process. Finally, after

Listing 2.1: vfi_AM.m, Part 1

```
cons1 = bsxfun(@minus, A_high*kgrid'.^alpha  + delta*kgrid' ,kgrid);
cons2 = bsxfun(@minus, A_low*kgrid'.^alpha  + delta*kgrid' ,kgrid);

cons1(cons1<=0) = NaN;
cons2(cons2<=0) = NaN;

util1 =  log(cons1);
util2 =  log(cons2);

util1(isnan(util1)) = -inf;
util2(isnan(util2)) = -inf;
```

Listing 2.2: vfi_AM.m, Part 2

```
v       = zeros(nk,2);   % initial guess for value function:
                  % set to zero for simplicity
decis   = zeros(nk,2);   % initial value for policy function
metric  = 10;               % initial value for the convergence
   metric
iter = 0;
tme = cputime;
[rs,cs] = size(util1);
```

Listing 2.3: vfi_AM.m, Part 3

```
while metric > convcrit;

  contv= beta*v*prob'; % continuation value
    [tv1,tdecis1]=max(bsxfun(@plus,util1,contv(:,1)) );
  [tv2,tdecis2]=max(bsxfun(@plus,util2,contv(:,2)) );

  tdecis=[tdecis1' tdecis2'];
  tv=[tv1' tv2'];

  metric=max(max(abs((tv-v)./tv)));
  v=   tv;
  decis= tdecis;%
  iter = iter+1;
  metric_vector(iter) = metric;
  disp(sprintf('iter = %g ; metric = %e', iter,metric));
end;
disp('  ');
disp(sprintf('computation time = %f', cputime-tme));

% transform the decision index in capital choice
decis=(decis-1)*ink + mink;
```

convergence is achieved, we transform the variable `decis` into actual capital values over the grid.

The last part of the code does some simulations, generating a series for consumption and investment. In order to generate a series for A_t, it uses a subroutine called `markov.m`, which generates a Markov chain realization for the productivity shock (we are not going to give details about it). The file `figures.m` plots both the value and the policy function over the grid, and the series. The last graph is a log_{10} scale plot of `metric` for each iteration (i.e., on the vertical axis we get the order in terms of powers of 10 for the *sup norm*. Each graph is saved as a PostScript file.

2.2.3 Exercises

Exercise 2.1 Playing with the code

Part I — Running the code

Run the code `do_vfi_AM.m`. Familiarize with the output and the graphs. Now, after the line which loads parameter values, change the value for the discount factor to 0.995, by adding the following line:

```
betta = 0.995
```

This line changes the value set in the file `parameters.m` to a new value (this is a general way to do comparative statics by using a baseline set of parameters.) Save this file as `do_vfi_AM_beta.m` and run it.

1. What do you notice in the convergence process? What happens to computational time? (Why?)

Part II — Accuracy

We now want to see what happens if we change the number of gridpoints. Change the code so that the grid has 100 gridpoints. (**Hint**: notice that after you set `nk=100`, then you also have to modify the variables `kgrid` and `ink`, therefore you need to add those lines too!!!) Save the file as `do_vfi_AM_smallgrid.m` and run it.

1. What can you notice? Now try with 2000 gridpoints, save the file as `do_vfi_AM_largegrid.m`, and run it. Do you see any change?

Part III — Comparative statics

We want to do a series of comparative statics exercises. In order to do that, we can modify the file `do_compstat`, which is a basic structure for this task. Open the file `do_compstat.m`. You should see the following lines (I have excluded the lines that plot the simulated results for brevity):

```
%% Generate solution and simulation for case A
% load parameters and grid
parameters;
% solve the model via VFI
vfi_AM;
% store results in few new variables
v_A = v;
decis_A = decis;
```

```
controls_A = controls;

%% Generate solution and simulation for case B
% load parameters and grid
parameters;
% modify parameters here

% solve the model via VFI
vfi_AM;
% store results in few new variables
v_B = v;
decis_B = decis;
controls_B = controls;
```

This file compares a case A and a case B, where the case A is the benchmark (i.e. the solution with default parameters), and case B is the one with the updated parameter value. Notice the line that says:

```
% modify parameters here
```

We can just put the new value we want to consider there. Then by running the file we should get the graphs of case A and B and compare them.

1. As a first try, set `delta = 0.8` (notice that this implies a larger depreciation rate for capital, given the definition of δ in the code). Save the file as `do_compstat_delta.m`. Run the do file, what changes with respect to the benchmark case?

1. Now let's change the values for the shock realizations `A_high` and `A_low`. In particular, let's have a mean-spread transformation such that the the mean is the same given the iid hypothesis for the transition matrix. Set `A_high=1.25` and `A_low=0.75`, save the file as `do_compstat_shock.m` and run the code. What can you notice?

1. Let's now relax the assumption of iid shocks. In order to do that, we need to change the transition matrix `prob` by inducing some persistence in the stochastic process. Modify the matrix in such a way that each realizations has a probability of repeating itself in the following period of 95%, i.e. the probability of a high (low) realization tomorrow given that the realization today is high (low) is 0.95. Save the file as `do_compstat_persistent.m` and run the code. What conclusions can you draw from the graphs?

Exercise 2.2 Irreversible investment

VFI makes easy to incorporate inequality constraints. In order to see this, modify the code by introducing an irreversibility constraint for investment, i.e. $K_t \geq (1-\delta)K_{t-1}$.

1. Repeat the previous exercise for the model with irreversible investment. The relevant codes are `do_vfi_AM_irrinv.m` and `do_compstat_irrinv.m`.

Exercise 2.3 Convex adjustment costs

It is also quite straightforward to introduce convex adjustment costs. Modify the code by adding the adjustment costs in the form $AC(K_t, K_{t-1}) \equiv \zeta(K_t - K_{t-1})^2$, where ζ is a parameter. Choose $\zeta = .25$ to begin with, then play with it and see what happens.

Exercise 2.4 Reversible vs irreversible investmen

Now let's compare the reversible investment model with the one with irreversible investment. In order to do that, we use the file `do_compare.m`. (Notice that in this case we will have to change parameters in two points of the code!)

1. Run the code as it is, and look at the graphs. What are the main differences between the reversible and irreversible investment models? Check in particular the simulated series of consumption and investment.

1. Now let's how the depreciation rate is crucial. First change the depreciation rate of capital to `delta=0.99`. You can do this by inserting `delta=0.99` in the appropriate spaces (remember: you have two do it twice for this code!!!). Save the new file as `do_compare_delta.m` and run it. What do you observe? Can you explain it intuitively? Now set `delta=0.5` (in both the appropriate spaces!!!) and run the code again. What now?

1. Let's see how the model with irreversible investment reacts when the variance of the shocks is reduced. Set `A_high=1.25` and `A_low=0.75`, save the file as `do_compare_shock.m` and run the code. This must be surprising for you! (Is it?)

1. Finally, what about persistence? Set the transition matrix `prob` such that each realization has a probability of repeating itself in the following period of 95%, i.e. the probability of a high (low) realization tomorrow given that the realization today is high (low) is 0.95. Save the file as `do_compare_persistent.m` and run the code.

Exercise 2.5 Many possible realizations of the technology shock

We can adapt the code to deal with more than two realizations of the shocks. This is a slightly more involved change since we need to modify the code in several points:

1. we need a new variable that stores the number of realizations of the shocks (call it `num_realiz`) in the parameters' file
2. we need to provide a transition matrix `prob` which adapts to `num_realiz` (for the moment we stick to the iid case for simplicity) in the parameters' file
3. we need a vector where we store the actual values for technology shock realizations in the parameters' file
4. we need to adapt the way in which we create the matrices U_j (hint: use the Kronecker product, use Matlab help for the `kron` command)
5. we need to adjust the way in which we compute the Bellman operator, in particular for the expectations' part
6. finally, we must adapt the simulations to the generic case with many possible realizations of the Markov chain

We can of course do the same for the model with irreversible investment. The two solution codes are respectively `vfi_AM_general.m` and `vfi_AM_irrinv_general.m`.

2.3 POLICY FUNCTION ITERATION

The value function iteration algorithm is quite slow, and for many economic problems computational speed is important. We therefore would like to have a similar approach that has the same convergence properties, but it is faster. We can have a much faster computation of the solution by using the policy function iteration algorithm, also known as Howard's improvement algorithm.

Remember that our problem is

$$V(K_{-1}) \equiv \max_{\{K_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t U\left(f(K_{t-1}) - K_t\right)$$
$$s.t. \quad 0 \leq K_t \leq f(K_{t-1}), \quad t = 0, 1, ...$$

The policy function iteration (PFI) guesses a policy function $g : X \to \mathbb{R}$ such that $K_t \approx g^{(0)}(K_{t-1})$. Given this guess, it is possible to calculate the value function implied by it. Then we can perform a step of VFI, getting a new policy function guess, and iterate until convergence. The algorithm therefore has the following steps:

1. Start with a guess for the policy function $K' = g^{(0)}(K)$
2. Calculate the value associated with this policy function:

$$V^{(0)}(K) = \sum_{t=0}^{\infty} \beta^t U\left(f(K_{t-1}) - g^{(0)}(K_{t-1})\right)$$

3. For any $n \geq 0$, get a new policy function $K' = g^{(n+1)}(K)$ by solving

$$\max_{K'} \left\{ U\left(f(K) - K'\right) + \beta V^{(n)}\left(K'\right) \right\}$$

4. Calculate the value associated with this policy function:

$$V^{(n+1)}(x) = \sum_{t=0}^{\infty} \beta^t U\left(f(K_{t-1}) - g^{(n+1)}(K_{t-1})\right)$$

5. Iterate over 3-4. Stop if $\left\| V^{(n+1)} - V^{(n)} \right\| < \epsilon$

To simplify notation, we can think of the system as residing in one of N predetermined positions, call them x_i for $i = 1, 2, ..., N$. There is a class of predetermined stochastic matrices $\mathcal{M}$ of size $N \times N$, with elements denoted as P. Notice that $P_{ij} = Prob\{x_{t+1} = x_j \mid x_t = x_i\}$. We can write the Bellman equation as:

$$v(x_i) = \max_{P \in \mathcal{M}} \left\{ u(x_i) + \beta \sum_{j=1}^{N} P_{ij} v\left(x_j\right) \right\}$$

or in a more compact form:

$$v = \max_{P \in \mathcal{M}} \left\{ u + \beta P v \right\} \tag{2.12}$$

We can rewrite it as as

$$v = Tv$$

where T is the operator corresponding to the RHS of (2.12). Define another operator $B \equiv T - I$ such that

$$Bv = \max_{P \in \mathcal{M}} \{u + \beta P v\} - v$$

and therefore we can see the policy function iteration as solving $Bv = 0$. The algorithm we can use is the following:

1. Given $P^{(n)}$, get $v^{(n)}$ from

$$\left(I - \beta P^{(n)}\right) v^{(n)} = u^{(n)} \tag{2.13}$$

2. Find $P^{(n+1)}$ such that

$$u^{(n+1)} + \left(\beta P^{(n+1)} - I\right) v^{(n)} = Bv^{(n)} \tag{2.14}$$

The first step is a linear algebra problem that can be solved by posing:

$$v^{(n)} = \left(I - \beta P^{(n)}\right)^{-1} u^{(n)}$$

You can interpret the policy iteration algorithm as a form of the Newton's method to find the zeroes of $Bv = 0$. Notice that, by using (2.13) you can rewrite equation (2.14) as:

$$\left(I - \beta P^{(n+1)}\right) v^{(n+1)} + \left(\beta P^{(n+1)} - I\right) v^{(n)} = Bv^{(n)}$$

or in other terms

$$v^{(n+1)} = v^{(n)} + \left(I - \beta P^{(n+1)}\right)^{-1} Bv^{(n)} \tag{2.15}$$

Since from equation (2.13) we can interpret $\left(I - \beta P^{(n+1)}\right)$ as the gradient of $Bv^{(n)}$, therefore Howard's improvement algorithm is a version of the Newton's method[4] to find the roots of $Bv = 0$.

How do we implement it numerically? Say we have only two possible values for the shock as in the previous Section, and define the two $n \times n$ matrices:

$$J_h\left(k_i, k_j\right) = \left\{ \begin{array}{ll} 1 & if \quad g\left(k_i, A_h\right) = k_j \\ 0 & o/w \end{array} \right\}$$

[4]More on Newton's method in the next section

For a given policy function, $k' = g(k, A)$, define two $n \times 1$ vectors U_h such that:

$$U_h(k_i) = u\left(A_h f(k_i) + (1-\delta)k_i - g(k_i, A_h)\right)$$

Assume the policy function is used forever. We can associate the two vectors $V_h(k_i)$ as the values associated with starting from state (k_i, A_h). Therefore:

$$\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} + \beta \begin{bmatrix} \mathcal{P}_{11}J_1 & \mathcal{P}_{12}J_1 \\ \mathcal{P}_{21}J_2 & \mathcal{P}_{22}J_2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$$

We can therefore solve for V_h by means of elementary linear algebra and have:

$$\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \left[I - \beta \begin{pmatrix} \mathcal{P}_{11}J_1 & \mathcal{P}_{12}J_1 \\ \mathcal{P}_{21}J_2 & \mathcal{P}_{22}J_2 \end{pmatrix} \right]^{-1} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \tag{2.16}$$

Now, we can do the following:

1. Given an initial feasible policy function, calculate equation (2.3) and use them to solve for the value functions with equation (2.16)
2. Do one iteration on the Bellman equation (2.11) by using value functions found in step 1
3. Iterate until convergence

It turns out that this algorithm is much faster than value function iteration, it converges in fewer iterations. We can speed up even more by slightly changing step 2: instead of doing just one iteration on the Bellman equation, do m iteration, where m is a reasonably small integer (I would say up to 50).

2.3.1 The Code

The PFI code is very similar to the VFI, of course. In fact, the code for VFI must be changed in a few critical spots. The file `pfi_AM.m` is substantially identical to `vfi_AM.m` (hence initial conditions, grid and parameters are set in the same way), except in the main loop.

Listing 2.4: pfi_AM.m

```
1 while metric > convcrit;
2
3   contv= beta*v*prob'; % continuation value
```

Listing 2.4 (continued): pfi_AM.m

```
4
5   [tv1,tdecis1]=max(bsxfun(@plus,util1,contv(:,1)) );
6   [tv2,tdecis2]=max(bsxfun(@plus,util2,contv(:,2)) );
7
8   tdecis=[tdecis1' tdecis2'];
9
10  % Build return vectors
11  r1 = zeros(cs,1);
12  r2 = zeros(cs,1);
13  for i=1:cs
14    r1(i) = util1(tdecis1(i),i);
15    r2(i) = util2(tdecis2(i),i);
16  end
17
18  % create matrices Js (see lecture notes)
19  g2=sparse(cs,cs);
20  g1=sparse(cs,cs);
21  for i=1:cs
22    g1(i,tdecis1(i))=1;
23    g2(i,tdecis2(i))=1;
24  end
25  % This is the marix P (see lecture notes)
26  trans=[ prob(1,1)*g1 prob(1,2)*g1; prob(2,1)*g2 prob(2,2)*g2];
27
28  % Linear algebra step to get the value function associated with
        P
29  tv(:) = ((speye(2*cs) - beta.*trans))\[ r1; r2 ];
30
31  metric=max(max(abs((tv-v)./tv)));
32  v=   tv; % .15*tv+.85*v; %
33  decis= tdecis;%
34  iter = iter+1;
35  metric_vector(iter) = metric;
36  disp(sprintf('iter = %g ; metric = %e', iter,metric));
37 end;
```

Listing 2.4 (continued): pfi_AM.m

```
38 disp('  ');
39 disp(sprintf('computation time = %f', cputime-tme));
40
41 % transform the decision index in capital choice
42 decis=(decis-1)*ink + mink;
```

We need to calculate the return matrices implied by the new policy, and then the matrices J that contain the policy function in matrix form. We then perform the linear algebra step to find the new value function. Notice that the code can be written in a more efficient way, however it is already more than 20 times faster than the VFI code, and converges in substantially fewer iterations. This is a general property of the Newton's method, since it has a quadratic convergence, while VFI has linear convergence[5].

2.3.2 Exercise: Irreversible investment vs. adjustment costs, the revenge

Modify the PFI code to analyse the model with irreversible investment and the model with investment adjustment costs seen in the section about value function iteration. (Hint: are the modifications done for the VFI code enough?)

2.4 NUMERICAL METHODS

In this section, we discuss some extension and drawbacks of the VFI and PFI codes used in the previous sections. We then illustrate a few numerical basic tools, included in the CompEcon Toolbox which we will use in the projection methods section.

2.4.1 Integrals

The codes used in the dynamic programming chapters assumed discrete shocks. However, they can be easily adapted to continuous shocks. In order to do this,

[5]The interested reader can consult Judd (1998) for more details.

we need to find a way to calculate expectations over a continuous supports, which is equivalent to calculating an integral. There are different ways to do it, and all methods approximate the integral with a finite sum (this is called *numerical quadrature*). For example, if we want to calculate the integral over a set I of a function $f(x)$ weighted by a weight function $w(x)$, we can use

$$\int_I f(x)w(x)dx \approx \sum_{i=1}^{n} f(x_i)w_i \tag{2.17}$$

and different methods usually differ in the way we choose the nodes x_i and the weights w_i. Newton-Cotes methods approximate the f between nodes using low order polynomials, then sum the integrals of those polynomials (which are easy to calculate). There are two widely used versions: the trapezoid rule, which approximates the function with piecewise linear interpolants, and the Simpson's rule which uses piecewise quadratic interpolants.

Gaussian quadrature chooses nodes and weights by matching some moments of the distribution. Imagine you have a weighting function $w : I \to R$, where $I \in R$ is an interval. Then the Gaussian quadrature of order n chooses n quadrature nodes that satisfy the $2n$ moment matching conditions:

$$\int_I x^k w(x)dx = \sum_{i=1}^{n} w_i x_i^k,\ k = 0, ..., 2n-1 \tag{2.18}$$

When the weighting function is $w(x) = 1$, we call this method Gauss-Legendre quadrature, and it turns out to be well suited for calculating integrals of smooth (i.e. with continuous derivatives) functions.

Monte Carlo methods randomly choose nodes. However, there are a lot of issues with those randomly chosen nodes, the biggest being that every random number generator is quite imprecise. There are therefore so called pseudo-Monte Carlo methods that try to improve the reliability of the integral.

Depending on the problem, there might be one method that outperforms others in terms of accuracy and/or computational time. However, it's worth mentioning that there are various ways of replicating a continuous process with one with discrete support, for example Tauchen's method is one of them[6].

All these methods are included in the CompEcon Toolbox. These routines allow to compute integrals of real-valued function over bounded intervals. Trapezoid rule is implemented by using

[6]By looking for Tauchen in Google, you may find a few different MATLAB routines that implement this method, if you are interested.

```
[x,w] = qnwtrap(n,a,b);
```

where n is the number of nodes, and a and b are the extrema of the interval. Notice that all these parameters can be vectors, in case you want to calculate multidimensional integrals. Simpson's rule and Gauss-Legendre integration have the same syntax with commands respectively `qnwsimp` and `qnwlege`. For example, we can calculate the definite integral of $\log(x)$ on $[3,7]$ with 20 nodes trapezoid rule by using

```
[x,w] = qnwtrap(20,3,7);
integral = w'*log(x);
```

Exercises

Exercise 2.6 Integrals

1. Calculate the integral of e^{-x} on the interval $[-1,1]$ using the trapezoid rule.
2. Calculate the integral of $|x|^{\frac{1}{2}}$ on the interval $[-1,1]$ using Simpson's rule.
3. Calculate the integral of $(1+25x^2)^{-1}$ on the interval $[-1,1]$ using Gauss-Legendre method.

Exercise 2.7 Compare the methods

Write a short script that compares the three methods. Calculate the integral of e^{-x}, $|x|^{\frac{1}{2}}$ and $(1+25x^2)^{-1}$ on the interval $[-1,1]$ by hand (these are pretty easy to calculate). Then use CompEcon commands and calculate the integrals with 10,20,30,100 nodes for each method. Save the results in a matrix and then compare the accuracy, i.e. the difference between the numerical integral and the correct integral calculated by hand. What can you notice?

CompEcon contains routines for computing Gaussian nodes and weights of the most common distributions. For our purpose, the most important command is `qnwnorm` which calculates the nodes and weights for multidimensional normal distributions with the syntax

```
[x,w] = qnwnorm(n,mu,var);
```

where `n` is a vector containing the number of nodes in each dimension, `mu` is the mean vector, and `var` is the variance-covariance matrix. For example, if we want to calculate the expectations of e^{-x}, with $x \sim \mathbf{N}(0,1)$, and 30 nodes, we can use

```
[x,w] = qnwnorm(30,0,1);
expectations = w'*exp(-x);
```

Exercises

Exercise 2.8 Univariate normal

Calculate the expected value of $x^{-\sigma}$, for $\sigma = \{0.5, 1, 2, 10\}$ and $x \sim \mathbf{N}(0,1)$. Use 30 nodes.

Exercise 2.9 Multivariate normal

Calculate the expected value of $e^{x_1+x_2}$, with x_1 and x_3 jointly normal with $Ex_1 = 3$, $Ex_2 = 4$, $Var(x_1) = 2$, $Var(x_2) = 4$, $Cov(x_1, x_2) = -1$. Use 10 nodes in the x_1 direction and 15 nodes in the x_2 direction.

Similar functions exist for Gamma, lognormal, Beta distributions, for example. There are more sophisticated ways of choosing nodes and weights, and CompEcon has more routines dedicated to it. The interested reader can refer to Miranda and Fackler (2004) for more details.

2.4.2 Nonlinear equations

In many cases, the optimization step in the VFI or PFI algorithm can be substituted with solving first-order conditions. This of course saves time, and it often makes the problem easier to solve. Moreover, nonlinear equations are essential in projection methods, which are a method for solving functional equations. In the following, we illustrate a few methods that are helpful for this task. All the following methods are implemented in Matlab "functions of functions", i.e. one of the input is a function that contains the equation we want to solve.

Bisection method

Bisection is the simplest method for finding the solution of a nonlinear equations of the form

$$f(x) = 0$$

where $f(x)$ is a continuous real-valued function on a real interval $[a, b]$. The algorithm is very simple:

1. Start with two points a_1, b_1 such that $f(a_1) < 0$, $f(b_1) > 0$
2. Choose a new point $x_2 \in [a_1, b_1]$ and calculate the sign of $f(x_2)$. If positive, now restrict your search to $[a_1, x_2]$, if negative to $[x_2, b_1]$.

Coding the algorithm is very easy. This method works well in case there is a unique solution, however it might create some problems when the equation at hand has more than one root. The function `bisect` in CompEcon implements the bisection method:

```
f = inline('x^3 - 5');
x = bisect(f,1,2)
```

The function can be defined with a function file, especially when one needs to find the roots of a parameterized version of the objective function. `bisect` works with vectorial or matricial functions.

Exercise 2.10 Bisection

1. Find the roots of $e^{-x^2} - \cos(x)$ over the interval $[-4, 6]$.
2. Plot the function over the interval. What can you observe?

Function iteration

This method is useful for equations of the type

$$f(x) = 0$$

where $f : R^n \to R^n$. The procedure consists of first write the equation as $x = g(x)$ for some function $g : R^n \to R^n$, and then given a guess $x^{(0)}$

1. Calculate $x^{(1)} = g(x^{(0)})$
2. Iterate over the recursion $x^{(n+1)} = g(x^{(n)})$ until convergence

Notice that rewriting the equation in the form $x = g(x)$ is always possible since $x = x - f(x)$. This method is widely used to solve Euler equations. Function iteration is guaranteed to converge to a fixed-point of g if g is differentiable and if the initial guess is close enough to the solution where $g'(x) < 1$. However, convergence happens quite often even if this condition is not satisfied. The routine `fixpoint` implements this algorithm in CompEcon. The syntax for `fixpoint` is:

```
g = inline('x^0.5');
x = fixpoint(g,0.4)
```

where the second argument of the command is an initial guess for the solution.

Exercise 2.11 Function iteration

Find the roots of $e^{-x^2} - \cos(x)$ over the interval $[-4, 6]$. (Hint: first you have to transform the equation from the form $f(x) = 0$ into $x = x - f(x)$).

Newton methods

Newton methods are the most widely used for nonlinear equations. In practice, this amounts to solve a sequence of linear problems that converge to the nonlinear problem's solution. Given the nonlinear equation $f(x) = 0$, take the first order Taylor approximation around $x^{(k)}$:

$$f(x) \approx f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)}) = 0$$

Newton methods solve the linear equation above, finding a new guess $x^{(k+1)}$ as $x^{(k+1)} = x^{(k)} - [f'(x^{(k)})]^{-1} f(x^{(k)})$, where $[f'(x^{(k)})]^{-1}$ is the Jacobian of the function f, and iterate over this until convergence. Newton's method converges if f is continuously differentiable and if the initial guess is "sufficiently" close to a root of f at which f' is invertible. However, what "sufficiently" close means is subject to trial and error. If the method does not converge, one needs to choose a better guess or change technique. Another issue is the robustness. If f is well behaved, there are usually no problems in the choice of the initial guess. However, for strange functions the solution might be highly dependent on the initial guess. Moreover, if f' is invertible but ill-conditioned, then the procedure can deliver a badly approximated solution. The CompEcon Toolbox includes a routine `newton` that computes the root of a function using the Newton's method. The syntax is slightly more complicated. We need to create a function file in the form:

```
[fval,fjac]=f(x,optional additional parameters)
```

where `fjac` is the Jacobian of the function. Then the command is

```
x = newton(f,x0)
```

where `x0` is an initial guess.

Exercise 2.12 Newton method

Find the roots of $e^{-x^2} - \cos(x)$ using the Newton method.

Quasi-Newton methods

For many problems, computing the Jacobian for implementing the Newton's method is computationally very expensive. Quasi-Newton methods are modifications of the Newton method in which the Jacobian is computed in an efficient way. A very popular one is Broyden's method. This amounts to guessing an initial value $x^{(0)}$, and a guess for the Jacobian $A^{(0)}$, and then calculate $x^{(1)} = x^{(0)} - [A^{(0)}]^{-1} f(x^{(0)})$. Then we need to update the Jacobian with the smallest possible change that satisfies the *secant condition*: $f(x^{(n+1)}) - f(x^{(n)}) = A^{(n+1)}(x^{(n+1)} - x^{(n)})$ (for speed, most of the implementation of the method update the inverse of the Jacobian). Then we iterate over the recursion $x^{(n+1)} = x^{(n)} - [A^{(n)}]^{-1} f(x^{(n)})$ until convergence. Many time the first guess $A^{(0)}$ is the numerical Jacobian at $x^{(0)}$. This method is again subject to all caveats for Newton's method, plus the fact that Jacobian's guesses must be "close" to the actual values. In fact, there is no guarantee that $A^{(n)}$ will converge to the true value of the Jacobian at the solution, and it usually does not. The library LIBM contains the routine `broydn` that implements Broyden's method in an efficient way[7]. The syntax is

```
[x, check] = broydn(f,x0,tol, iadmat,iprint, additional arguments)
```

where `tol` is the tolerance level for convergence, `iadmat` is 1 if the library `iadmat` is available, zero otherwise, `iprint` is 1 if you want to see information about each iteration, 0 otherwise. The output `check` is 0 if convergence has been achieved, 1 if there were problems.

Exercise 2.13 Broyden method

[7]This routine has been coded by Michael Reiter (IHS Wien).

1. Find the roots of $e^{-x^2} - \cos(x)$ using the Broyden's method. Play with the initial guess to see how the solution changes.
2. Imagine you have a two-periods economy with an initial amount of savings s_0. Therefore, the equations that describe the intertemporal decision of the agent are given by:

$$c_1^{-\sigma} = \beta(1+r)c_2^{-\sigma}$$
$$c_1 + \frac{c_2}{1+r} = y_1 + \frac{y_2}{1+r} + s_0$$

where $r = 0.05$, $\sigma = 2$, $\beta = .99$, and $y_1 = y_2 = 1$. Solve for the optimal allocation for different values of the initial savings. (Hint: write a function that takes as second input a vector of initial savings, and solve these equations in a vectorized way.

A WORD ON `fsolve` The function `fsolve` is included in MATLAB and it is a complicated routine that uses combinations of the methods described above and more sophisticated algorithms. It is quite general in his applicability, but most of the time it could be an overkill. Moreover, since it is quite sophisticated, it may be slow for some problems.

2.4.3 Optimization

The most intensive computational task of the VFI is the maximization step. In our code we keep it as simple as possible given that we only have to choose over one decision variable, and hence we can easily use the discretized state space for our purpose. However, this task might be more involved and sometimes troublesome if we have more than one decision variable. We can still discretize all the possible choices, however this procedure becomes more and more burdensome and memory intensive the larger the number of choice variables we have. Therefore, we have to rely on other numerical optimization techniques. In some cases, it is possible to use a first order condition approach and therefore we can treat the maximization step as finding the solution of a system on non-linear equations, using one of the techniques illustrated above. However many complicated problems are non-convex by assumption, and we might need to rely on global numerical optimization techniques, like line search, pattern search or more sophisticated procedures as genetic algorithms, simulated annealing and swarm particle search. Many times it is possible to use more sophisticate techniques just to get a good initial guess for the value function

(and policy function) and then simpler (faster) methods can deliver a solution which is accurate enough. There is always a trade-off between approximation and speed that must be taken into consideration.

Derivative-free methods

These methods are very similar in spirit to bisection for non-linear equations. They in fact look for the highest point in a series of smaller and smaller intervals. The most widely used is golden search:

1. Start with two points x_1, x_2 in the interval, such that $x_1 < x_2$, and evaluate $f(x_i)$
2. If $f(x_1) \geq f(x_2)$, then the new interval is $[a, x_2]$, otherwise is $[x_1, b]$
3. Iterate until convergence

CompEcon contains the routine `golden` that implements golden search for univariate functions. The syntax is

```
x = golden(f,a,b)
```

However, since we many times want to work pointwise with functions from R^n to R^n, the library LIBM contains the routine `goldsvec` which can be used in a vectorized problem. This routine works in the same way as `golden`:

```
x = goldsvec(f,a,b)
```

Exercise 2.14 Golden Search

1. Use the `golden` command to mazimize the MATLAB function `humps` on the interval $[-10, 10]$
2. Use the `golden` command to mazimize the MATLAB function `humps` on the interval $0.2, 2]$. Comment.
3. *(Difficult)* Imagine you have a two-periods economy with an initial amount of savings s_0, per-period CRRA utility function $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$, and discount factor β. The intertemporal budget constraint of the agent is:

$$c_1 + \frac{c_2}{1+r} = y_1 + \frac{y_2}{1+r} + s_0$$

where $r = 0.05$, $\sigma = 2$, $\beta = .99$, and $y_1 = y_2 = 1$. Solve for the optimal allocation for different values of the initial savings using the `goldsvec` routine.

Newton-Raphson method

These routines are similar to Newton's methods for nonlinear equations. The idea is to start from the second order Taylor expansion of the maximand:

$$f(x) \approx f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)}) + \frac{1}{2}(x - x^{(k)})^T f''(x^{(k)})(x - x^{(k)})$$

From here we can take the first order conditions:

$$f'(x^{(k)}) + f''(x^{(k)})(x - x^{(k)}) = 0$$

Hence a solution can be computed by iterating over

$$x^{(n+1)} = x^{(n)} - [f''(x^{(n)})]^{-1} f'(x^{(n)})$$

until convergence. This algorithm converges if f is twice continuously differentiable and if the initial guess is close to a local maximum of f at which the Hessian f'' is negative definite. Again, a good guess is crucial for this method, and ill-conditioning of the Hessian might deliver bad approximation. Analogously to non-linear equations techniques, there are of course Quasi-Newton methods that use a computable Hessian. CompEcon includes the routine `qnewton` for this purpose. Many of the routines in the MATLAB Optimization Toolbox (e.g. those in `fmincon`) also are from this family, although more sophisticated.

"I-am-in-deep-shit" problems

Once in a while, you will face a very irregular optimization problem. Either the problem is highly non-convex, or presents multiple local maxima, or it is in general highly irregular. In this cases, MATLAB has a Global Optimization Toolbox which contains global optimization routines. There is also a lot of software available online (either free or commercial) which can be used for these problems. A non-exhaustive list of possible algorithms is the following:

- Line search
- Pattern search
- Genetic algorithms
- Simulated annealing
- Swarm search optimization

Most of the time, you need to use one of these methodologies just to find an initial good guess, and then you can revert to quasi-Newton methods.

Parallelization, curse of dimensionality and interpolation

A general point is that both global and quasi-Newton techniques are usually non-vectorized, at least in their basic MATLAB version. Hence, we have to perform one maximization for each gridpoint, which is clearly highly inefficient. However VFI is an embarrassingly parallelizable algorithm, i.e. the maximization step can be performed gridpoint by gridpoint, hence each of those maximization problems can be send to a different processor. In problem with high-dimensional state spaces this helps in speeding up the convergence.

Moreover, VFI relies on setting up a grid for the state variables. This grid needs to have many points to get a good approximation, but in a unidimensional problems this is not a big issue. The problem can arise when we have more than one state variable. Say N is the number of state variables, and you want to discretize each state in m grid points. Therefore, your value function needs to be evaluated in m^N points. This is a big problem for a computer, especially it is a memory-intensive task and you might easily run out of memory. This feature of the VFI algorithm is called the *curse of dimensionality*. For this reasons, economists have developed several methods that can help in dealing with dimensionality issues. One way to reduce the computational burden is to use interpolation of the value function and the choice variables between grid points. We will talk about interpolation and function approximation in the next section when discussing projection methods. Another way is a smart choice of gridpoints, putting more in regions where the problem is highly non-linear, and less elsewhere where linearity is a good approximation.

Finally, combining VFI with other procedures helps in speeding up the convergence. Projections methods are widely used in economics, and they can be easily applied to Bellman equations. Projections methods are a general way to solve functional equations (i.e. equations where the unknown is a function), and in principle they can be applied also to the non-linear system of functional equations that is the set of first order conditions. Projections methods typically need substantially fewer grid points to get the same approximation (for example, in our RBC model you would get more or less the same approximation error with VFI over 1000 gridpoints and collocation over 10 gridpoints). In models with many state variables this property makes a big difference on computational speed. However, projections techniques involve a set of numerical techniques (integration, approximation, optimization and/or non linear equations solvers, grid choice techniques, etc.) that have big learning costs and longer coding/debugging time. In the next section we will try to cut the learning costs.

2.5 PROJECTION METHODS

Projection methods are widely used in engineering, physics and mathematics to solve ordinary or partial differential equations and other functional equations. Since in economics we also deal with functional equations, we can apply similar techniques to economic models. We will look at our basic stochastic growth model, and solve the first order conditions with projection methods.

In many cases we don't need to look at the Bellman equation to find the solution of a model, since under some regularity conditions we are able to get the solution from first order conditions. This is true if the value function is differentiable. Let us write our dynamic optimization problem in a more general way

$$\begin{aligned} \max_{\{u_t\}_{t=0}^{\infty}} \quad & \sum_{t=0}^{\infty} \beta^t r\left(x_t, u_t\right) \\ s.t. \quad & x_{t+1} = h\left(x_t, u_t\right), \quad t = 0, 1, ... \\ & x_0 \in X \text{ given} \end{aligned}$$

where r is a return function, x is a vector of states, u is a vector of controls, and h describes the law of motion for the state vector. Notice that, when we know the optimal policy, we can always write by definition:

$$V(x) = r\left(x, g(x)\right) + \beta V\left[h\left(x, g(x)\right)\right]$$

Therefore, assuming differentiability of V, we can get the following first-order conditions:

$$/u: \quad r_u\left(x, g(x)\right) + \beta V'\left(h\left(x, g(x)\right)\right) h_u\left(x, g(x)\right) = 0 \tag{2.19}$$

Moreover, we can get the envelope condition:

$$\begin{aligned} /x \quad : \quad & V'(x) = r_x\left(x, g(x)\right) + r_u\left(x, g(x)\right) g'(x) + \\ & + \beta V'\left[h\left(x, g(x)\right)\right] \cdot \left\{h_x\left(x, g(x)\right) + h_u\left(x, g(x)\right) g'(x)\right\} \end{aligned} \tag{2.20}$$

If we can write the law of motion of the state variables only as a function of the controls (which happens in many economic models), i.e. $x' = h(u)$, then using (2.19) in (2.20) we get

$$V'(x) = r_x\left(x, g(x)\right) \tag{2.21}$$

Equation (2.21) is a version of the Benveniste and Scheinkman (1979) formula. Of course we need to make assumption on the primitives of the problem to be able to get a differentiable value function. It is possible to show that, together with assumptions that deliver the uniqueness of the value function and its concavity (see for example Stokey et al. (1989) or Miao (2014)), the following assumption guarantees differentiability of the value function in the interior of X:

Assumption 3. *The return function r is continuously differentiable with respect to the first argument in the interior of A.*

Now, if we can write the law of motion of the state variables as $x' = h(u)$, I can use equations (2.19) and (2.21) to get the following expression:

$$r_u(x_t, u_t) + \beta r_x(x_{t+1}, u_{t+1}) h'(u_t) = 0, \quad x_{t+1} = h(u_t) \tag{2.22}$$

Equation (2.22) is called **Euler equation**. We can solve for the optimal policy function by using the Euler equation. Notice that the same kind of analysis can be done for the stochastic case, with the obvious adaptations.

2.5.1 The basics of projection methods

To solve Euler equations, and in general functional equations related to economic models, it is useful to introduce projection methods. Their application is nowadays widespread in macroeconomics. Judd (1998) has a very good treatment of the method, and I refer the interested reader to him for an exhaustive analysis of the technique and its applications. The paper that introduced the technique to economists is Judd (1992), and it reports many examples that can be used as exercises. Miranda and Fackler (2004) illustrate the approach and describe in details the CompEcon tools that help working with these techniques. Again, I refer the reader interested in more technical details to check these references.

The basic idea of projection methods is very simple. We want to solve a functional equation of the type:

$$\mathcal{N}(g(x)) = 0 \tag{2.23}$$

where $\mathcal{N}$ is an operator (for example, the Bellman operator, or the FOCs operator) and $g(x)$ is a function of some space X. If we solve this functional equation numerically, then we can get only an approximation of our solution $g(x)$.

Judd (1992) suggests to follow a sequence of steps. The **first step** is to choose how we approximate the solution, and an appropriate concept of distance in order to measure the accuracy of our calculated solution. The easiest way to proceed is to assume that our function can be approximated as a linear combination of some simple functions (we call them *basis functions*, or simply basis), like polynomials for example.

The **second step** is to choose a *degree of approximation* n, i.e. how many basis functions we want to use. We also need to choose a computable approximation $\widehat{\mathcal{N}}$ for $\mathcal{N}$, if the exact operator is not computable directly. For example, if the operator is an integral, we need to use quadrature methods to compute an approximated version of the operator. We also must select some functions $p_i, i = 1, ..., n$ that we will use later to calculate the projections. We have now our approximated guess/solution $\widehat{g}(x) = \sum_{i=1}^{n} a_i \phi_i(x)$ for any x[8]. Therefore, any solution can be summarized by a vector of coefficients $\mathbf{a}$.

In the **third step**, we compute numerically the approximated policy function $\widehat{g}(x) = \sum_{i=1}^{n} a_i \phi_i(x)$ for a particular guess of $\mathbf{a}$, and we also compute the so called *residual function*

$$R(x; \mathbf{a}) \equiv \left(\widehat{\mathcal{N}}\left(\widehat{g}\right)\right)(x)$$

The first guess for $\mathbf{a}$ can be important. Depending on the application, it is crucial to start with a good guess to achieve convergence.

The **fourth step** is the calculation of the projections

$$P_i(\cdot) \equiv \langle R(\cdot; \mathbf{a}), p_i(\cdot) \rangle, i = 1, ..., n \tag{2.24}$$

where $\langle \cdot \rangle$ is the inner product.

Finally, in the **fifth step**, we iterate over step 3 and 4 to get a vector of coefficients $\mathbf{a}$ that sets the projections (2.24) to zero.

[8]There are many so called approximation theorems that postulate the existence of these approximated solutions for different choices of basis functions. The most well known is probably Weierstrass approximation theorem that postulate the existence of an approximation with natural polynomials. Chebychev approximation theorem is the analogous for Chebychev polynomials. However, sometimes there is not a proper approximation theorem, and more practical considerations should be taken into account when choosing basis functions. For more details, see Judd (1992) and Judd (1998).

Choice of basis functions

There are few details to keep in mind. First, the typical choice for the inner product used in the calculation of the projections is, given a weighting function $w(x)$:

$$\langle f(x), h(x) \rangle \equiv \int f(x)h(x)w(x)dx$$

Second, there are various possible choices for basis functions. We may use ordinary polynomials $1, x, x^2, x^3, ...$ However, it turns out that they are problematic in many situations. In general, we can divide the type of basis we choose in two broad categories: spectral methods and finite element methods. Spectral methods have basis functions that are almost everywhere nonzero, and are continuously differentiable as many time as needed, therefore imposing smoothness on the approximated function (which sometimes is not a desirable feature). A very popular choice in spectral methods is Chebychev polynomials. They are defined as

$$T_n(x) \equiv \cos(n \arccos x), \quad x \in [-1, 1]$$

and it is possible to generate them with the following recursive law:

$$\begin{aligned} T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \\ T_0(x) &= 1, \quad T_1(x) = x \end{aligned}$$

It is possible to show that those polynomials satisfy an orthogonality condition for a particular inner product:

$$\int_{-1}^{1} T_i(x) T_j(x) \left(1 - x^2\right)^{-\frac{1}{2}} dx = 0, \quad i \neq j$$

Now, let $z_l^n \equiv \cos\left(\frac{(2l-1)\pi}{2n}\right), l = 1, ..., n$ be the zeroes of T_n. We can show that

$$\Sigma_{l=1}^{n} T_i\left(z_l^n\right) T_j\left(z_l^n\right) = 0, \quad i \neq j$$

therefore we want to use the roots of the Chebychev polynomials for interpolation. Moreover, there are theorems that show that, using zeroes as evaluation points, convergence is achieved faster. In general,, Chebychev polynomials are very useful if the function we want to approximate is smooth, and therefore can be used when we have the proof or the reasonable suspicion that our solutions are differentiable and regular.

Finite element techniques use instead basis functions that are zero except for a small support. A very popular choice is tent functions, aka piecewise linear basis, Take an approximation with support in $[a,b]$ and be $h=(a-b)/n$. Then, for $i=0,1,...,n$:

$$\phi_i(x)=\begin{cases}0 & a\leq x\leq a+(i-1)h\\ (x-(a+(i-1)h))/h & a+(i-1)h\leq x\leq a+ih\\ 1-(x-(a+(i-1)h))/h & a+ih\leq x\leq a+(i+1)h\\ 0 & a+(i+1)h\leq x\leq b\end{cases}$$

A generalization of those bases is piecewise degree k polynomials, like Hermite polynomials and cubic splines. Cubic splines are indeed very popular, and you will use them often for problems that are irregular.

In general, if the function we want to approximate is smooth, we use Chebychev polynomials, otherwise we can use piecewise polynomial interpolation of different orders, usually perfecting the order by trial and error, trying to use the lowest possible order that delivers a good approximation.

If our state space is multidimensional, we can use tensor products to build multidimensional approximation from unidimensional basis: if $\{\phi_i(x)\}_{i=1}^{\infty}$ is the basis for a function in one variable, we can use the tensor product basis $\{\phi_i(x)\phi_j(y)\}_{i,j=1}^{\infty}$ for functions of two variables, and so on. The main problem is that the number of elements increases exponentially with the dimension. There are various ways to overcome this problem: one is to use complete polynomials of order k. They are those polynomials built as:

$$\mathcal{P}_k\equiv\left\{x_1^{i_1}\cdot...\cdot x_n^{i_n}\left|\sum_{l=1}^{n}i_l\leq k,0\leq i_1,...,i_n\right.\right\}$$

While the previous choices are coded in some routines in CompEcon, the complete polynomials are not included.

Choice of projections

There are various possibilities for the choice of which particular projections we want to use. The difference will depend generally on the functions p_i that we use in the calculation of the projection. One possibility is the least-squares approach, which minimizes the weighted sum of squared residuals:

$$\min_{\mathbf{a}}\langle R(x;\mathbf{a}),R(x;\mathbf{a})\rangle$$

The Galerkin method uses the basis functions for projections, and solves the following equations:

$$P_i(\mathbf{a}) \equiv \langle R(x;\mathbf{a}), \phi_i(x) \rangle = 0, \quad i = 1, ..., n$$

The method of moments uses the first n polynomials:

$$P_i(\mathbf{a}) \equiv \langle R(x;\mathbf{a}), x^{i-1} \rangle = 0, \quad i = 1, ..., n$$

The subdomain method solves

$$P_i(\mathbf{a}) \equiv \langle R(x;\mathbf{a}), \mathbf{I}_{D_i} \rangle = 0, \quad i = 1, ..., n$$

where $\{D_i\}$ is a sequence of intervals covering the entire domain of the function, and $\mathbf{I}_{D_i}$ is the indicator function for D_i.

A very popular method is collocation. It is popular because it is easy to implement and easily computed with some tricks (see Miranda and Fackler (2004) for an explanation of how the method is implemented with matrix algebra). The collocation method chooses n points $\{x_i\}_{i=1}^n$ in the domain and solves

$$R(x_i;\mathbf{a}) = 0, \quad i = 1, ..., n$$

and notice that this is equivalent to solve

$$P_i(\mathbf{a}) \equiv \langle R(x;\mathbf{a}), \delta(x - x_i) \rangle = 0, \quad i = 1, ..., n$$

where $\delta(\cdot)$ is the Dirac delta function that is equal to zero everywhere but in zero, where it takes value 1. Orthogonal collocation chooses collocation nodes as the zeroes of the basis function, making the computations even faster thanks to orthogonality conditions.

Notice that the main computational burden of projection methods is the calculation of the projections, therefore speed is a requirement that we want to keep in mind when choosing which method to adopt. Collocation is very fast, since other methods would require the computation of an integral.

2.5.2 CompEcon routines for projection methods

CompEcon has a series of functions that make using projection methods easier for the beginners. Moreover, many of these routine are coded in C and then MEX-ified for MATLAB use, making them very fast. It is therefore a good idea to learn how to use them for day-to-day research. Those routines can easily

be modified to include more options once one is skilled enough, although I do not recommend to mess with them unless totally sure of what one is doing.

The routines are quite a black box, however for the beginner this is helpful, since many transformations and matrix manipulations are automated. Again, these things can be modified once one has some knowledge of the basics.

GENERATING FUNCTIONAL SPACES WITH `fundefn` The function `fundefn` creates a MATLAB structured variable that characterizes the functional space of the basis functions chosen by the programmer. The syntax is:

```
fspace = fundefn(bastype,n,a,b,order);
```

The inputs of this function are related to information about the basis functions. `bastype` is a string that identifies which type of basis function we want to use. The possible options are given by Chebichev polynomials (`'cheb'`), splines (`spli'`), or linear spline basis with finite difference derivatives (`'lin'`). The input `n` is a vector indicating the degree of approximation along each dimension (this routines automatically uses tensor product for multidimensional interpolation). The values of `a` and `b` identify the left and right endpoints for interpolation intervals for each dimension. Finally, `order` is used to specify the spline order (and it is therefore optional in all other basis type's choice), where the default is cubic splines.

An example of how to use it is the following. We want to create a functional basis space for 5th degree Chebychev polynomials for a univariate function in the interval $[-5, 6]$. We use the command:

```
fspace = fundefn('cheb',5,-5,6);
```

If instead we are interested in generating a cubic spline space in two dimensions, with 10 basis functions in the first dimension and 8 in the second on the interval $\{(x_1, x_2) : -5 \leq x_1 \leq 6, 2 \leq x_2 \leq 9\}$, we write:

```
fspace = fundefn('spli',[10 8],[-5 2], [6 9]);
```

If we prefer a 5th order spline, then we use:

```
fspace = fundefn('spli',[10 8],[-5 2], [6 9],5);
```

EVALUATING FUNCTIONS WITH `funeval` One crucial point of the projection methods is to evaluate the approximated function $\widehat{g}(x) = \sum_{i=1}^{n} c_i \phi_i(x)$, given a set of coefficients **c** and basis functions $\{\phi_i(x)\}_{i=1}^{n}$. The routine `funeval` performs this task for us. Define a set of points `x` over which we want to evaluate our approximated function, and a vector of coefficients `c`, we can then write

```
y = funeval(c,fspace,x);
```

The set of points `x` needs to be defined as a $m \times k$ matrix, where m is the number of points over which we want to evaluate the function, and k is the dimensionality of the space. For example, if we want to evaluate our approximated function over 100 points in a 3-dimensional basis space, then `x` must be a 100×3 matrix.

OTHER USEFUL ROUTINES: `funbas` AND `funnode` In our code for solving the stochastic growth model with collocation, we will make use of two more commands. The first is `funbas`, which calculates the matrix of values of the basis functions in a particular set of points `x`:

```
Basis = funbas(fspace,x);
```

This function is useful when we want to evaluate a function at the same points but with different values of the coefficients (for example, because of our iterative procedure to find the coefficients vector **c**). We can then use

```
Basis = funbas(fspace,x);
y = Basis*c;
```

which is equivalent to using `funeval`.

Finally, `funnode` computes standard nodes for the basis functions included in CompEcon. For example, if the basis functions are Chebychev polynomials, then the command

```
x = funnode(fspace);
```

returns the $1 \times k$ cell array of Chebychev zeros.

2.5.3 How to solve the stochastic growth model with collocation

In order to solve the stochastic growth model (SGM) with collocation, we need to make a few choices. First choice is, what will be our operator $\mathcal{N}$? Second choice, what function will we approximate, i.e. which one is our function $g(\cdot)$?

We will give an example working with first order conditions, hence our operator $\mathcal{N}$ will be the set of FOCS. In the example, we will use K' as our function $g(K, A)$. Another possible choice is to use consumption C, however it does not make a big difference in this simple model. In general, the choice of which operator and which functions we approximate can make a *big* difference in terms of speed, accuracy and number of "tricks" needed to get the code converging.

A general principle is to reduce the number of equations and functions to include in $\mathcal{N}(g(x)) = 0$ to the minimum. If you can substitute some variables in some equations and reduce the number of equations/unknown, then do it. Moreover, equations can be highly non-linear in some variables, while linear or quasi-linear in others: this is the typical case in Kuhn-Tucker conditions, where the equations are linear in the Lagrange multipliers. It is then sometimes helpful to exploit this feature.

For the SGM, the social planner problem is

$$\begin{aligned} &\max_{K_t, C_t} E_0 \sum_{t=0}^{\infty} \beta^t U(C_t) \\ &\quad s.t. \quad C_t + K_t - (1-\delta)K_{t-1} \leq A_t K_{t-1}^{\alpha} \end{aligned}$$

Substituting the resource constraint into the objective function, we have

$$\max_{K_t} E_0 \sum_{t=0}^{\infty} \beta^t U(A_t K_{t-1}^{\alpha} - K_t + (1-\delta)K_{t-1})$$

The first order conditions for the social planner problem are:

$$U'(C_t) = \beta E_t \left[U'(C_{t+1}) \left(\alpha A_{t+1} K_t^{\alpha-1} + 1 - \delta \right) \right] \tag{2.25}$$

We can calculate the steady state in the deterministic case, normalizing $A^{ss} = 1$:

$$1 = \beta \left[\left(\alpha K^{\alpha-1} + 1 - \delta \right) \right]$$

from which we get

$$K^{ss} = \left[\frac{1}{\alpha\beta} - \frac{1-\delta}{\alpha} \right]^{\frac{1}{\alpha-1}}$$

We will create a grid around the steady state capital K^{ss}. However, let's focus for one moment on equation (2.25). We can rewrite it highlighting the dependencies on the state variables as

$$U'(C(K,A)) = \beta E\left[U'(C(K',A'))\left(\alpha A'(K')^{\alpha-1}+1-\delta\right)\right] \tag{2.26}$$

and given our decision to approximate next period's capital, we can rewrite it by using the function $g(\cdot)$ as

$$U'(\max(0, AK^{\alpha}+(1-\delta)K - g(K,A)) = \beta E\left[U'(\max(0, A'(g(K,A))^{\alpha} \right.$$
$$\left. +(1-\delta)g(K,A) - g(g(K,A),A'))\right)\left(\alpha A'(g(K,A))^{\alpha-1}+1-\delta\right)\Big]$$

This is our operator $\mathcal{N}(g(x))$, where $x = (K,A)$ is a point in the state space.

2.5.4 The code

The code is composed of various files. The main file to run is `solveSGM.m`, which is reported in Listing 2.5. The general scheme of the code is the following:

1. Set parameters values and grid for states
2. Generates the functional space for the basis functions, and a "good" initial guess for the coefficients
3. Find coefficients that put the residual function as close to zero as possible (this is in fact done by solving nonlinear equations with Broyden's method)
4. Test solution accuracy and then simulate the model

This file calls two other files: `mainSGM.m` contains the proper collocation algorithm, and calls the file in which we stored the residual function `focsSGM.m`. In the following, we illustrate each file in details.

solveSGM.m

This file assigns values to parameters, then creates lower and upper bounds for the capital grid around the deterministic steady state. It then sets the parameters for calculating expectations with quadrature techniques, using the function

`qnwnorm` from CompEcon, and sets the range for shocks by setting lower and upper bound for the productivity level A.

Then we set the parameters for the collocation algorithm. First we choose how may rounds of approximation we want to do. This is a common practice when solving models with projection methods. Typically, it is quite easy to find a solution on a coarse grid with a low degree of accuracy. However, it is more complicated to get a good accurate solution when working with finer grids. Therefore the practice is to first run collocation over a coarse grid, find a rough solution and then use this solution as an initial guess over a finer grid. It may be necessary to do two or more rounds of approximation to get accurate solutions if the problem is highly non-linear or presents irregularity of sort.

The next line sets the order of the approximation for each round. In this example, we use order 5 in both dimensions (capital and productivity shock) for the first round, and 10 for the second round.

The following section sets the type of basis function we want to use (notice that the code has commented out all the possible options, if the reader wants to experiment with them and compare results). Then the code calls the file `mainSGM.m`, which contains the collocation algorithm. The last part of the code is then devoted to simulations (calling the file `simul_SGM.m`) and figures.

Listing 2.5: solveSGM.m

```
global alpha betta rho sig sigma delta sigeps ;
global nQuadr QuadrWeights QuadrPoints;
global  RoundAppr rounds_approx  ;

%% PARAMETERS:
alpha =.4;                      % production function coefficient
delta = .1;                     % depreciation rate for capital
betta =  .95;                    % discount factor
sig = 1; % .5;%        % CRRA utility parameter (c^(1-sig))/(1-
   sig)
sigma = .05;                     % S.D. of productivity shock
rho = 0;%;  .9; %             % persistence of productivity shock

%% create a grid for capital
kstar = (1/(alpha*betta) - ...
  (1-delta)/alpha )^(1/(alpha-1)); % det. steady state
```

Listing 2.5 (continued): pfi_AM.m

```
k_min = .5*kstar;
k_max = 2*kstar;

%% parameters for quadrature
nQuadr = 50; % 100;% %number of quadrature points;
% we choose nQuadr high to get smoothness;
[QuadrPoints,QuadrWeights] = qnwnorm(nQuadr,0,sigma^2);

%% Range for shock
sigeps = sigma/sqrt(1-rho^2);
% Range for shock:
A_max = 3*sigeps;
A_min =  -3*sigeps;

%% Parameters for the collocation algorithm
rounds_approx = 2; % number of rounds of approximation
Order_vector = [5 10; 5 10];% n. of grid points for each round
ntest = 100; % n. of grid points for testing (for each dimension)

%% Approximation type for CompEcon
  % approxtype = 'lin';  % piecewise linear
    approxtype = 'cheb'; % chebychev polynomials
%     approxtype = 'spli'; % splines
  splineorder = []; % splines' order, default are cubic splines

%% parameters for simulations
number_series =1;    % number of series
periods_simulation =  100; % number of periods for the simulation
k0 = k_min.*ones(number_series,1);

%% Run main file
mainSGM; % solves the model

% deliver an accuracy statistic (the smaller the better)
max_test
```

Listing 2.5 (continued): pfi_AM.m

```
51
52 % save the solution
53 save solutionSGM.mat park fspace ;
54
55 %% SIMULATIONS AND FIGURES
56
57 % simulate the series
58 [A, k,  c, y ,epsilon] = simul_SGM(number_series, ...
59   periods_simulation, k0, park, fspace);
60
61 figure(1);
62 subplot(2,1,1);
63 plot(1:periods_simulation, k);
64 xlabel('time'); ylabel('k'); axis([1 periods_simulation 0 7]);
65 title('Simulation: capital'); legend('k_t');
66
67 subplot(2,1,2);
68 plot(1:periods_simulation-1,c(2:end));
69 xlabel('t'); ylabel('c');
70 title('Simulation: consumption'); legend('c_t');
```

mainSGM.m

The file `mainSGM.m` executes the collocation algorithm. First, it generates the functional space for the basis functions with `fundefn` from CompEcon, by using parameters defined in code 2.5. Then generates a grid in the state space (k, A) by choosing standard interpolation nodes related to the basis' type choice, with `funnode`. Notice the use of the function `gridmake` which stacks gridpoints in a $n_k n_A \times 2$ matrix, where n_k and n_A are the number of gridpoints in each dimension. In order to set initial conditions for the basis' coefficients, we guess a solution and then regress this solution on the basis functions (defined using `funbas`). In the first round, we just guess that capital tomorrow is equal to capital today, while in the second round we use the solution found on a coarse grid/approximation as a guess. Finally, we find basis' coefficients `park` such that the residual function is set to zero, by solving the set of nonlinear

first order conditions (set in the function file `focs.SGM` described below) with the Broyden's method. Notice that there are as many equations as gridpoints, hence we solve in the first round a system of 25 equations, while in the second the system has 100 equations. This method is well suited for up to 500 equations, but then it may cause some ill-conditioning. In those cases, it is better to use a combination of nonlinear solvers and fixed-point methods.

Listing 2.6: mainSGM.m

```
for oo = 1: rounds_approx
  RoundAppr = oo;     % this tells you at which round of
     approximation we are

  % Range on which we approximate the solution:
  LowerBound = [k_min A_min ];
  UpperBound = [k_max A_max];

  % Approximation order
  Order = Order_vector(:,oo);

  % for reference, need this for guess after round 1
  if oo >= 2
    fspace_old = fspace;
  end

  disp(' '); disp(' ' );
  disp(sprintf('RoundAppr %d, # gridpoints = [%d %d]',...
    RoundAppr, Order(1), Order(2)));
  disp(' ');

  % the following lines generate basis function space
  % we can choose among chebychev polynomials, splines of
     different
  % orders and piecewise linear functions

  if(strcmp(approxtype,'spli'))
    fspace = fundefn(approxtype,Order,LowerBound,UpperBound,
       splineorder);
```

Listing 2.6 (continued): pfi_AM.m

```
  else
    fspace = fundefn(approxtype,Order,LowerBound,UpperBound,[]);
  end;

  % the following commands create gridpoints
  nodes = funnode(fspace);
  Grid = gridmake(nodes);

  % Set initial conditions
  if (RoundAppr == 1)
    knext = Grid(:,1);
  else    % if we are at second approx round, we use the solution
      of the first round
    % as initial conditions on the new larger grid
    knext = funeval(park, fspace_old, Grid);
  end;

  % generate basis functions Basis at Grid :
  Basis = funbas(fspace,Grid);

  % set initial value for parameters of the approximation
  park =  Basis\knext;

  % solve FOCs with Broyden method for nonlinear equations
  [park,info] =  broydn('focsSGM',park,1e-8,0,1,Grid,fspace);
  disp(sprintf(' info = %d',info)); % if info=0, everything went
     fine, o/w the Broyden algorithm didn't converge
  disp(sprintf('    '));

end;
```

focsSGM.m

The function `focsSGM.m` contains the residuals to be put to zero by the nonlinear solver. Expectations are calculated using quadrature parameters set in

`solveSGM.m`. The equation to be solved is scaled such that it is unit free, i.e. the value is in fact in percentage. Hence if the algorithm gets say 10^{-8}, the error is in the order of $10^{-8}\%$ of the marginal utility of consumption.

Listing 2.7: focsSGM.m

```
%% FOCS for the stochastic growth model

function equ = focsSGM(park, Grid,fspace);

global alpha betta sig rho delta A_bar
% global LowerBound UpperBound
global nQuadr QuadrWeights QuadrPoints

LowerBound = fspace.a;
UpperBound = fspace.b;

%rename grid
k = Grid(:,1);
A = Grid(:,2);

% evaluate policy functions
knext = funeval(park,fspace, Grid);
fofk = exp(A).*(k.^alpha) + (1-delta).*k;
% c = fofk - knext;
c = max(fofk - knext, zeros(length(Grid),1));

n = length(k);
% generate nQuadr replications of the Grid, one for each
   realization of shock:
Grid_knext = kron(knext,ones(nQuadr,1));

% Expected value of next period A, corresponding to Grid:
ExpA =  rho*A;
% all realizations of next A:
GridANext = kron(ExpA,ones(nQuadr,1)) + ...
  kron(ones(n,1),QuadrPoints);
% truncate it to state space:
```

Listing 2.7 (continued): pfi_AM.m

```
GridANext = min(max(GridANext,LowerBound(2)),UpperBound(2));

GridNext = [Grid_knext GridANext];

% calculate variables at t+1
knextnext = funeval(park,fspace, GridNext);
fofknext = exp(GridANext).*(Grid_knext.^alpha) + (1-delta).*
    Grid_knext;
 cnext = max(fofknext - knextnext, zeros(length(Grid_knext),1));
% cnext = fofknext - knextnext;
mucnext = muc(cnext);
mpknext = mpk(GridNext);
% calculate expectations with quadrature
exp_mucnext = (QuadrWeights'*reshape(mpknext.*mucnext,nQuadr,n))
    ';

% equation to be solved: Euler equation
equ = (muc(c) - betta.*exp_mucnext)./muc(c);

% avoid strange solutions
if (any(cnext<0))  || (any(c<0)) || (any(knext<0))  || (any(
    knextnext<0))
  equ(1) = 1e100;
end;
```

2.5.5 Tricks and difficulties

For the purpose of illustrating the methodology, the simple stochastic growth model is helpful. However, being so simple, it does not involve any major difficulty that is usually associated with solving models with projection methods. In the following we illustrate a few of them.

Convergence

One problem with projection methods is that there is not a theorem guaranteeing convergence of the procedure. Hence, many times the researcher has to "guide" the convergence process with ad-hoc solutions. One widely used method is *homotopy*, i.e. start from the solution of a simpler model and iteratively modify the $\mathcal{N}$ operator to match the new, more complicated model. Another option is using bounds for the first iterations, once we are sure we have a good guess.

Dependence on the initial guess

Many times the solution we obtain depends on the initial guess, since we often rely on local methods for solving the system of functional equations. Hence, it is important to get a good guess. Moreover, it is crucial to check that we always converge to the same solution even if we start from different initial guesses, keeping in mind what said above about convergence issues.

Ill-conditioning

In most cases, the solution of a model with projection methods will make use of non-linear solvers, and consequently either Jacobian or Hessian matrices will have to be computed. Ill-conditioning is a very frequent problem in highly non-linear models, and there are no easy fixes. Sometimes, a smarter choice of the gridpoints helps. Increasing the number of approximation rounds also could make the problem less pronounced.

Speed considerations and models with large state spaces

In highly-dimensional models, speed may be a concern. However, recent work has shown that projection methods can be used with large state spaces efficiently, with an appropriate choice of the gridpoints. One option is to use the Smoliak algorithm for choosing a sparse grid (see Malin et al. (2011)). Another recent approach is to use the ergodic set of the model and pick gridpoints in it (see Judd et al. (2012)).

2.5.6 Exercise: solving the RBC model with collocation over the Bellman equation

We have solved the model under the assumption that first order conditions are necessary and sufficient. There are cases in which this is not possible, therefore one would then think that the only option is to use other dynamic programming techniques. In fact, VFI and PFI can be combined with collocation. We ask you to do it with the RBC model: create a code that solves the Belmann equation with collocation.

2.A LIBM INSTALLATION

The library LIBM includes a few routines which will be used in the collocation algorithm. You just need to add it to the MATLAB path.

2.B COMPECON INSTALLATION

CompEcon Toolbox is a free set of routines for economists which accompanies Miranda and Fackler (2004), you can download it from the website

`http://www4.ncsu.edu/~pfackler/compecon/download.html`

We provide you with the most recent version, including also compiled MEX files. You should just add it to your MATLAB path, and it should work.

However, sometimes using a version of the MEX files installed in another machine does not work. Installation requires a C compiler, since many routines are coded in C and you will need to transform them in MEX files in order to be able to use them with MATLAB. We suggest you use Visual Studio, however any C compiler would do. There are a few freely available, for example MinGW.

First, you download and add the folder to your MATLAB path, making sure to use the option "Add with subfolders". Then, you run the file `mexall`, by just typing it in the MATLAB command window. The file will ask for a C compiler, and it will compile the C files into MEX files. In order to check if your installation worked, please run the file `solveSGM.m` in the collocation folder. Notice that this file requires the library LIBM, therefore the latter must

be already installed. If you receive any error message, or have any problem with the installation, please write an email to a.mele@surrey.ac.uk.

2.C SUMMARY OF FILES

VALUE FUNCTION ITERATION The VFI code is composed of five scripts. The script `parameters.m` sets the parameters' value and the grid. The file `vfi_AM.m` contains the main loop that iterates over the Bellman equation. `markov.m` is an utility for generating Markov chains. The file `figures.m` plots graphs. Finally, the script `do_vfi_AM.m` runs all the scripts.

POLICY FUNCTION ITERATION The PFI code is composed of five scripts. The script `parameters.m` sets the parameters' value and the grid. The file `pfi_AM.m` contains the main loop that iterates over the policy function. `markov.m` is an utility for generating Markov chains. The file `figures.m` plots graphs. Finally, the script `do_pfi_AM.m` runs all the scripts.

PROJECTION METHOD The collocation code that solves the stochastic growth model has six files. `mainSGM.m` is the main file, solving the functional equation with Broyden's method. `focsSGM.m` is a function file with the projections (in this case, they correspond to the FOCs for the problem at hand). The file `simul_SGM.m` simulates the solution. There are then two utility files, `muc.m` and `mpk.m`, which respectively calculate the marginal utility of consumptiona and the marginal product of capital. Finally, `solveSGM.m` set parameters and grid, calls the main file, then simulates the series and produces a series of graphs that illustrate the results.

BIBLIOGRAPHY

Braun, T., L. M. Korber, and Y. Waki (2012). Some unpleasant properties of log-linearized solutions when the nominal rate is zero. Federal Reserve Bank of Atlanta working paper 2012-5a.

Chang, Y. and S.-B. Kim (2007). Heterogeneity and aggregation: Implications for labor-market fluctuations. *American Economic Review 97*(5), 1939–1956.

Judd, K. L. (1992, December). Projection methods for solving aggregate growth models. *Journal of Economic Theory 58*(2), 410–452.

Judd, K. L. (1998). *Numerical Methods in Economics*. MIT Press.

Judd, K. L., L. Maliar, and S. Maliar (2012, November). Merging Simulation and Projection Approaches to Solve High-Dimensional Problems. NBER Working Papers 18501, National Bureau of Economic Research, Inc.

Ljungqvist, L. and T. J. Sargent (2012). *Recursive Macroeconomic Theory*. MIT Press.

Malin, B. A., D. Krueger, and F. Kubler (2011). Solving the multi-country real business cycle model using a smolyak-collocation method. *Journal of Economic Dynamics and Control 35*(2), 229 – 239. Computational Suite of Models with Heterogeneous Agents II: Multi-Country Real Business Cycle Models.

Marcet, A. and R. Marimon (2011). Recursive contracts. Technical report.

Miao, J. (2014). *Economic Dynamics in Discrete Time*. The MIT Press.

Miranda, M. J. and P. L. Fackler (2004). *Applied Computational Economics and Finance*, Volume 1 of *MIT Press Books*. The MIT Press.

Stokey, N., R. J. Lucas, and E. Prescott (1989). *Recursive methods in economic dynamics*. Harvard University Press.