Lecture Notes 3

Approximation Methods

In this chapter, we deal with a very important problem that we will encounter in a wide variety of economic problems: approximation of functions. Such a problem commonly occurs when it is too costly either in terms of time or complexity to compute the true function or when this function is unknown and we just need to have a rough idea of its main properties. Usually the only thing that is required then is to be able to compute this function at one or a few points and formulate a guess for all other values. This leaves us with some choice concerning either the local or global character of the approximation and the level of accuracy we want to achieve. As we will see in different applications, choosing the method is often a matter of efficiency and ease of computing.

Following Judd [1998], we will consider 3 types of approximation methods

1. Local approximation, which essentially exploits information on the value of the function in one point and its derivatives at the same point. The idea is then to obtain a (hopefully) good approximation of the function in a neighborhood of the benchmark point.

2. L^p approximations, which actually find a *nice* function that is close to the function we want to evaluate in the sense of a L^p norm. Ideally, we would need information on the whole function to find a good approximation, which is usually infeasible – or which would make the problem

of approximation totally irrelevant! Therefore, we usually rely on interpolation, which then appears as the other side of the same problem, but only requires to know the function at some points.

3. Regressions, which may be viewed as an intermediate situation between the two preceding cases, as it usually relies — exactly as in econometrics, on m moments to find n parameters of the approximating function.

3.1 Local approximations

The problem of the local approximation of a function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is to make use of information about the function at a particular point $x_0 \in \mathbb{R}$, to produce a good approximation of f in a neighborhood of x_0. Among the various available method 2 are of particular interest: the Taylor series expansion and Padé approximation.

3.1.1 Taylor series expansion

Taylor series expansion is certainly the most wellknown and natural approximation to any student.

The basic framework: This approximation relies on the standard Taylor's theorem:

Theorem 1 *Suppose $F : \mathbb{R} \longrightarrow \mathbb{R}$ is a C^{k+1} function, then for $x^\star \in \mathbb{R}^n$, we have*

$$\begin{aligned}
F(x) &= F(x^\star) + \sum_{i=1}^{n} \frac{\partial F}{\partial x_i}(x^\star)(x_i - x_i^\star) \\
&+ \frac{1}{2}\sum_{i=1}^{n}\sum_{i_1=1}^{n} \frac{\partial F}{\partial x_{i_1} \partial x_{i_2}}(x^\star)(x_{i_1} - x_{i_1}^\star)(x_{i_2} - x_{i_2}^\star) + \ldots \\
&+ \frac{1}{k!}\sum_{i_1=1}^{n} \ldots \sum_{i_k=1}^{n} \frac{\partial F}{\partial x_{i_1} \ldots \partial x_{i_1}}(x^\star)(x_{i_1} - x_{i_1}^\star) \ldots (x_{i_k} - x_{i_k}^\star) \\
&+ \mathscr{O}\left(\|x - x^\star\|^{k+1}\right)
\end{aligned}$$

The idea of Taylor expansion approximation is then to form a polynomial approximation of the function f as described by the Taylor's theorem. This approximation method therefore applies to situations where the function is at least n times differentiable to get a n–th order approximation. If this is the case, then we are sure that the error will be at most of order $\mathscr{O}\left(\|x - x^\star\|^{k+1}\right)$.[1].

In fact, we may look at Taylor series expansion from a slightly different perspective and acknowledge that it amounts to approximate the function by an infinite series. For instance in the one dimensional case, this amounts to write

$$F(x) \simeq \sum_{k=0}^{n} \alpha_k (x - x^\star)^k$$

where $\alpha_k = \frac{1}{k!}\frac{\partial F}{\partial x_i}(x^\star)$. As n tends toward infinity, the latter equation may be understood as a power series expansion of the F in the neighborhood of $x^\star$. This is a natural way to think of this type of approximation if we just think of the exponential function for instance, and the way a computer delivers $\exp(x)$. Indeed, the formal definition of the exponential function is traditionally given by

$$\exp(x) \equiv \sum_{i=0}^{\infty} \frac{x^k}{k!}$$

The advantage of this representation is that we are now in position to give a very important theorem concerning the relevance of such approximation. Nevertheless, we need to report some preliminary definitions.

Definition 1 *We call the radius of convergence of the complex power series, the quantity r defined by*

$$r = \sup\left\{ |x| : \left| \sum_{k=0}^{\infty} \alpha_k x^k \right| < \infty \right\}$$

r therefore provides the maximal radius of $x \in \mathbb{C}$ for which the complex series converges. That is for any $x \in \mathbb{C}$ such that $|x| < r$, the series converges while it diverges for any $x \in \mathbb{C}$ such that $|x| > r$.

[1]Let us recall at this point that a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^k$ is $\mathscr{O}\left(x^\ell\right)$ if $\lim_{x \to 0} \frac{\|f(x)\|}{\|x\|^\ell} < \infty$

Definition 2 *A function $F : \Omega \subset \mathbb{C} \longrightarrow \mathbb{C}$ is said to be analytic, if for every $x^\star \in \Omega$ there exists a sequence α_k and a radius r such that*

$$F(x) = \sum_{k=0}^{n} \alpha_k (x - x^\star)^k \text{ for } \|x - x^\star\| < r$$

Definition 3 *Let $F : \Omega \subset \mathbb{C} \longrightarrow \mathbb{C}$ be a function, and $x^\star \in \Omega$. $x^\star$ is a singularity of F is F is analytic on $\Omega - \{x^\star\}$ but not on Ω.*

For example, let us consider the tangent function $\tan(x)$. $\tan(x)$ is defined by the ratio of two analytic functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

since $\cos(x)$ and $\sin(x)$ may be written as

$$\begin{aligned} \cos(x) &= \sum_{i=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} \\ \sin(x) &= \sum_{i=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \end{aligned}$$

However, the function obviously admits a singularity at $x^\star = \pi/2$, for which $cos(x^\star) = 0$.

Given these definition we can now enounce the following theorem:

Theorem 2 *Let F be an analytic function in $x \in \mathbb{C}$. If F or any derivative of F exhibits a singularity at $x^o \in \mathbb{C}$, then the radius of convergence in the complex plane of the Taylor series expansion of F in the neighborhood of $x^\star$*

$$\sum_{k=0}^{\infty} \frac{1}{k!} \frac{\partial^k F}{\partial x^k}(x^\star)(x - x^\star)^k$$

is bounded from above by $\|x^\star - x^o\|$.

This theorem is extremely important as it gives us a guideline to use Taylor series expansions, as it actually tells us that the series at $x^\star$ cannot deliver any accurate, and therefore reliable, approximation for F an any point farther away from $x^\star$ than any singular point of F. An extremely simple example to understand this point is provided by the approximation of the function $F(x) = \log(1-x)$, where $x \in (-\infty, 1)$, in a neighborhood of $x^\star = 0$. First of all, note that $x^o = 1$ is a singular value for $F(x)$. The Taylor series expansion of $F(x)$ in the neighborhood of $x^\star = 0$ writes

$$\log(1-x) \simeq \varphi(x) \equiv -\sum_{k=1}^{\infty} \frac{x^k}{k}$$

What the theorem tells us is that this approximation may be used for values of x such that $\|x - x^\star\|$ is below $\|x^\star - x^o\| = \|0 - 1\| = 1$, that is for x such that $-1 < x < 1$. In other words, the radius of convergence of the Taylor approximation to this function is $r = 1$. As an numerical illustration, we report in table 3.1, the “true” value of $\log(1-x)$, its approximate value using 100 terms in the summation, and the absolute deviation of this approximation from the “true” value.[2] As can be seen from the table, as soon as $\|x^\star - x^o\|$ approaches the radius, the accuracy of the approximation by the Taylor series expansion performs more and more poorly.

Table 3.1: Taylor series expansion for $\log(1-x)$

x	$\log(1-x)$	$\varphi_{100}(x)$	ε
-0.9999	0.69309718	0.68817193	0.00492525
-0.9900	0.68813464	0.68632282	0.00181182
-0.9000	0.64185389	0.64185376	1.25155e-007
-0.5000	0.40546511	0.40546511	1.11022e-016
0.0000	0.00000000	0.00000000	0
0.5000	-0.69314718	-0.69314718	2.22045e-016
0.9000	-2.30258509	-2.30258291	2.18735e-006
0.9900	-4.60517019	-4.38945277	0.215717
0.9999	-9.21034037	-5.17740221	4.03294

[2]The word true lies between quotes as it has been computed by the computer and is therefore an approximation!

The usefulness of the approach: This approach to approximation is particularly useful and quite widespread in economic dynamics. For instance, anyone that has ever tried to study the dynamic properties of any macro model — let's say the optimal growth model — has once encountered the method. Let's take for instance the optimal growth model which dynamics may be summarized — when preferences are logarithmic and technology is Cobb–Douglas — by the two following equations[3]

$$\begin{aligned} k_{t+1} &= k_t^{\alpha} - c_t + (1-\delta)k_t && (3.1)\\ \frac{1}{c_t} &= \beta\frac{1}{c_{t+1}}\left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right) && (3.2)\end{aligned}$$

It is a widespread practice to linearize of log–linearize such an economy around the steady state, which we will define later on, to study the local dynamic properties of the equilibrium. Let's assume that the steady state has already been found and is given by $c_{t+1} = c_t = c^{\star}$ and $k_{t+1} = k_t = k^{\star}$ for all t.

Linearization: let us denote by $\widehat{k}_t$ the deviation of the capital stock k_t with respect to its steady state level in period t, such that $\widehat{k}_t = k_t - k^{\star}$. Likewise, we define $\widehat{c}_t = c_t - c^{\star}$. The first step of linearization is to reexpress the system in terms of functions. Here, we can define

$$F(k_{t+1}, c_{t+1}, k_t, c_t) = \left\{ \begin{array}{l} k_{t+1} - k_t^{\alpha} + c_t - (1-\delta)k_t \\ \\ \frac{1}{c_t} - \beta\frac{1}{c_{t+1}}\left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right) \end{array}\right.$$

and then build the Taylor expansion

$$\begin{aligned} F(k_{t+1}, c_{t+1}, k_t, c_t) \simeq\ & F(k^{\star}, c^{\star}, k^{\star}, c^{\star}) + F_1(k^{\star}, c^{\star}, k^{\star}, c^{\star})\widehat{k}_{t+1} + F_2(k^{\star}, c^{\star}, k^{\star}, c^{\star})\widehat{c}_{t+1} \\ & + F_3(k^{\star}, c^{\star}, k^{\star}, c^{\star})\widehat{k}_t + F_4(k^{\star}, c^{\star}, k^{\star}, c^{\star})\widehat{c}_t \end{aligned}$$

[3]Since we just want to make the case of Taylor series expansions, we do not need to be any more precise on the origin of these two equations. Therefore, we will take them as given, but we will come back to their determination in the sequel.

We therefore just have to compute the derivatives of each sub–function, and realize that the steady state corresponds to a situation where $F(k^\star,c^\star,k^\star,c^\star)=0$. This yields the following system

$$\left\{\begin{array}{l}\widehat{k}_{t+1}-\alpha k^{\star\alpha-1}\widehat{k}_t+\widehat{c}_t-(1-\delta)\widehat{k}_t=0\\ -\frac{1}{c^{\star 2}}\widehat{c}_t+\beta\frac{1}{c^{\star 2}}\left(\alpha k^{\star\alpha-1}+1-\delta\right)\widehat{c}_{t+1}-\beta\frac{1}{c^\star}\alpha(\alpha-1)k^{\star\alpha-2}\widehat{k}_{t+1}=0\end{array}\right.$$

which simplifies to

$$\left\{\begin{array}{l}\widehat{k}_{t+1}-\alpha k^{\star\alpha-1}\widehat{k}_t+\widehat{c}_t-(1-\delta)\widehat{k}_t=0\\ -\widehat{c}_t+\beta\left(\alpha k^{\star\alpha-1}+1-\delta\right)\widehat{c}_{t+1}-\beta\alpha(\alpha-1)\frac{c^\star}{k^\star}k^{\star\alpha-1}\widehat{k}_{t+1}=0\end{array}\right.$$

We then have to solve the implied linear dynamic system, bu this is another story that we will deal with in a couple of chapters.

Log–linearization: Another common practice is to take a log–linear approximation to the equilibrium. Such an approximation is usually taken because it delivers a natural interpretation of the coefficients in front of the variables: these can be interpreted as elasticities. Indeed, let's consider the following one–dimensional function $f(x)$ and let's assume that we want to take a log–linear approximation of f around $x^\star$. This would amount to have, as deviation, a log–deviation rather than a simple deviation, such that we can define

$$\widehat{x}=\log(x)-\log(x^\star)$$

Then, a restatement of the problem is in order, as we are to take an approximation with respect to $\log(x)$:

$$f(x)=f(\exp(\log(x)))$$

which leads to the following first order Taylor expansion

$$f(x)\simeq f(x^\star)+f'(\exp(\log(x^\star)))\exp(\log(x^\star))\widehat{x}=f(x^\star)+f'(x^\star)x^\star\widehat{x}$$

If we apply this technic to the growth model, we end up with the following system

$$\left\{\begin{array}{l}\widehat{k}_{t+1}-\frac{1-\beta(1-\delta)}{\beta}\widehat{k}_t+\left(\frac{1-\beta(1-\delta)}{\alpha\beta}-\delta\right)\widehat{c}_t-(1-\delta)\widehat{k}_t=0\\ -\widehat{c}_t+\widehat{c}_{t+1}-(\alpha-1)(1-\beta(1-\delta))\widehat{k}_{t+1}=0\end{array}\right.$$

3.2 Regressions as approximation

This type of approximation is particularly common in economics as it just corresponds to *ordinary least square* (OLS) . As you know the problem essentially amounts to approximate a function F by another function G of exogenous variables. In other words, we have a set of observable endogenous variables y_i, $i = 1 \ldots N$, which we are willing to explain in terms of the set of exogenous variables $\mathscr{X}_i = \{x_i^1, \ldots, x_i^k\}$, $i = 1 \ldots N$. This problem amounts to find a set of parameters Θ that solves the problem

$$\min_{\Theta \in \mathbb{R}^p} \sum_{i=1}^{N} (y_i - G(\mathscr{X}_i; \Theta))^2 \tag{3.3}$$

The idea is therefore that Θ is chosen such that on average $G(\mathscr{X}; \Theta)$ is close enough to y, such that G delivers a "good" approximation for the true function F in the region of x that we consider. This is actually nothing else than econometrics!

There are however several choices that can be made and that give us much more freedom than in econometrics. We now consider investigate these choices.

Number of data points: Econometricians never have the choice of the points they can use to reveal information on the F function, as data are given by history. In numerical analysis this constraint is relaxed, we may be exactly identified for example, meaning that the number of data points, N, is exactly equal to the number of parameters, p, we want to reveal. Never would a good econometrician do this kind of thing! Obviously, we can impose a situation where $N > p$ in order to exploit more information. The difference between these two choices should be clear to you, in the first case we are sure that the approximation will be exact in the selected points, whereas this will not necessarily be the case in the second experiment. To see that, just think of the following: assume we have a sample made of 2 data points for a function that we want to approximate using a linear function. we actually need 2 parameters to define a linear function: the intercept, α and the slope β. In such a case, (3.3) rewrites

$$\min_{\{\alpha,\beta\}}(y_1-\alpha-\beta x_1)^2+(y_2-\alpha-\beta x_2)^2$$

which yields the system of orthogonality conditions

$$\begin{aligned}(y_1-\alpha-\beta x_1)+(y_2-\alpha-\beta x_2) &= 0\\ (y_1-\alpha-\beta x_1)x_1+(y_2-\alpha-\beta x_2)x_2 &= 0\end{aligned}$$

which rewrites

$$\begin{pmatrix}1 & 1\\ x_1 & x_2\end{pmatrix}\begin{pmatrix}y_1-\alpha-\beta x_1\\ y_2-\alpha-\beta x_2\end{pmatrix}\equiv A.v=0$$

This system then just amounts to find the null space of the matrix A, which, in the case $x_1\neq x_2$ (which we can always impose as we will see next), imposes $v_i=0$, $i=1,2$. This therefore leads to

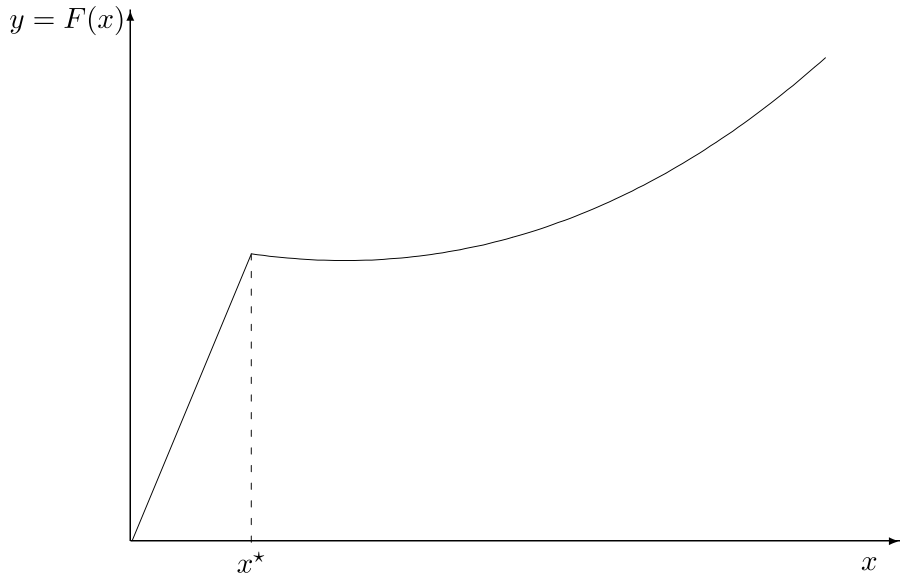
$$\begin{aligned}y_1 &= \alpha+\beta x_1\\ y_2 &= \alpha+\beta x_2\end{aligned}$$

such that the approximation is exact. When the system is overidentified this is not the case anymore.

Selection of data points: This is actually a major difference between econometrics and numerical approximations. We do control the space over which we want to take an approximation. In particular, we can spread the data points wherever we want in order to control information. For instance, let us consider the particular case of the function depicted in figure (3.1). As this function exhibits a kink in $x^\star$ it may be beneficial to concentrate a lot of points around $x^\star$ in order to reveal as much information as possible on the kink.

Functional forms: One key issue in the selection of the approximating function is a functional form. In most of the cases, a sequence of monomials

Figure 3.1: Selection of points



of different degrees is selected: $\mathscr{X}_i = \{1, x_i, x^2, \dots, x^p\}$, One advantage of this selection is basically simplicity. However, this often turns to be a very bad choice as many problems then turn out to be ill–conditioned with such a choice. The most obvious objection we may formulate against this specification choice is related to the multicolinearity problem. Assume for instance that you want to approximate a production function that depends on employment and the capital stock. The capital stock is basically an extremely smooth variable as, by construction, it is essentially a smooth moving average of investment decisions

$$k_{t+1} = i_t + (1-\delta)k_t \Longleftrightarrow k_{t+1} = \sum_{\ell=0}^{\infty}(1-\delta)^\ell i_{t-\ell}$$

Therefore, taking as a basis for exogenous variables powers of the capital stock is basically a bad idea. To see that, let us assume that $\delta = 0.025$ and i_t is a white noise process with volatility 0.1, then let us simulate a 1000 data points process for the capital stock and let us compute the correlation matrix for k_t^j,

$j = 1, \dots, 4$, we get:

	k_t	k_t^2	k_t^3	k_t^4
k_t	1.0000	0.9835	0.9572	0.9315
k_t^2	0.9835	1.0000	0.9933	0.9801
k_t^3	0.9572	0.9933	1.0000	0.9963
k_t^4	0.9315	0.9801	0.9963	1.0000

implying a very high correlation between the powers of the capital stock, therefore raising the possibility of occurrence of multicolinearity. A typical answer to this problem is to rely on orthogonal polynomials rather than monomials. We will discuss this issue extensively in the next section. A second possibility is to rely on parcimonious approaches that do not require too much information in terms of function specification. An alternative is to use *neural networks*.

Before going to Neural Network approximations, we first have to deal with a potential problem you may face with all the examples I gave is that when we solve a model: the true decision rule is unknown, such that we do not know the function we are dealing with. However, the main properties of the decision rule are known, in particular, we know that it has to satisfy some conditions imposed by economic theory. As an example, let us attempt to find an approximate solution for the consumption decision rule in the deterministic optimal growth model. Economic theory teaches us that consumption should satisfy the Euler equation

$$c_t^{-\sigma} = \beta c_{t+1}^{-\sigma} \left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right) \tag{3.4}$$

knowing that the capital stock evolves as

$$k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t \tag{3.5}$$

Let us assume that consumption may be approximated by

$$\phi(k_t, \theta) = \exp\left(\theta_0 + \theta_1 \log(k_t) + \theta_2 \log(k_t)^2\right)$$

over the interval $[\underline{k}; \overline{k}]$. Our problem is then to find the triple $\{\theta_0, \theta_1, \theta_2\}$ such that

$$\sum_{t=1}^{N} \left[\phi(k_t, \theta)^{-\sigma} - \beta \phi(k_{t+1}, \theta)^{-\sigma} \left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right)\right]^2$$

is minimum. This actually amounts to solve a non–linear least squares problem. However, a lot of structure is put on this problem as k_{t+1} has to satisfy the law of motion for capital:

$$k_{t+1} = k_t^\alpha - \exp\left(\theta_0 + \theta_1 \log(k_t) + \theta_2 \log(k_t)^2\right) + (1-\delta)k_t$$

The algorithm then works as follows

1. Set a grid of N data points, $\{k_i\}_{i=1}^N$, for the capital stock over the interval $[\underline{k};\overline{k}]$, and an initial vector $\{\theta_0, \theta_1, \theta_2\}$.

2. for each k_i, $i = 1, \dots, N$, and given $\{\theta_0, \theta_1, \theta_2\}$, compute

$$c_t = \phi(k_t, \theta)$$

   and

$$k_{t+1} = k_t^\alpha - \phi(k_t, \theta) + (1-\delta)k_t$$

3. Compute

$$c_{t+1} = \phi(k_{t+1}, \theta)$$

   and the quantity

$$\mathscr{R}(k_t, \theta) \equiv \phi(k_t, \theta)^{-\sigma} - \beta\phi(k_{t+1}, \theta)^{-\sigma}\left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right)$$

4. If the quantity

$$\sum_{t=1}^{N} \mathscr{R}(k_i, \theta)^2$$

   is minimal then stop, else update $\{\theta_0, \theta_1, \theta_2\}$ and go back to 2.

As an example, I computed the approximate decision rule for the deterministic optimal growth model with $\alpha = 0.3$, $\beta = 0.95$, $\delta = 0.1$ and $\sigma = 1.5$. I consider that the model may deviate up to 90% from its capital stock steady state — $\underline{k} = 0.1k^\star$ and $\overline{k} = 1.9k^\star$ — and used 20 data points. Figure 3.2 reports the approximate decision rule versus the "true" decision rule.[4] As can be seen

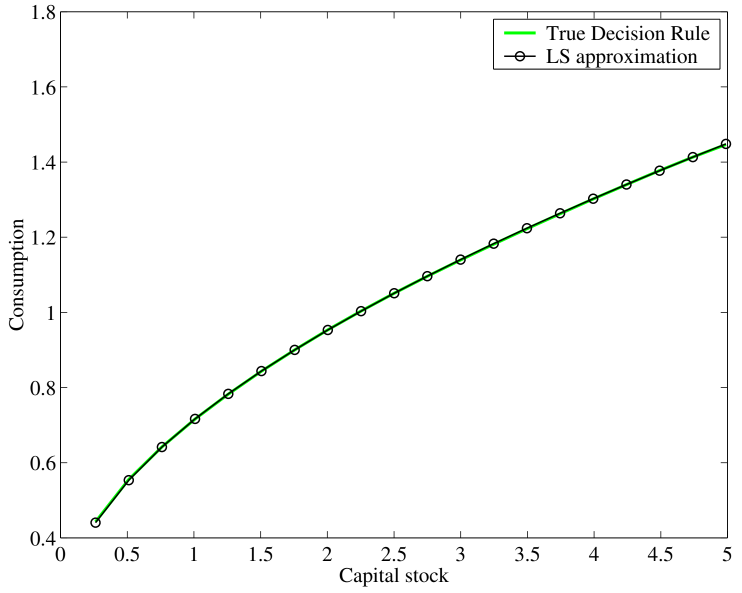
[4]The "true" decision rule was computed using value iteration, which we shall study later.

from the graph, the solution we obtained with our approximation is rather accurate and we actually have

$$\frac{1}{N}\sum_{i=1}^{N}\left|\frac{c_i^{true}-c_i^{LS}}{c_i^{true}}\right| = 8.743369e^{-4} \text{ and } \max_{i=\{1,\ldots,N\}}\left|\frac{c_i^{true}-c_i^{LS}}{c_i^{true}}\right| = 0.005140$$

such that the error we are making is particularly small. Indeed, using the approximate decision rule, an agent would make a maximal error of 0.51% in its economic calculus — *i.e.* 51 cents for each 100$ spent on consumption.

Figure 3.2: Least–square approximation of consumption

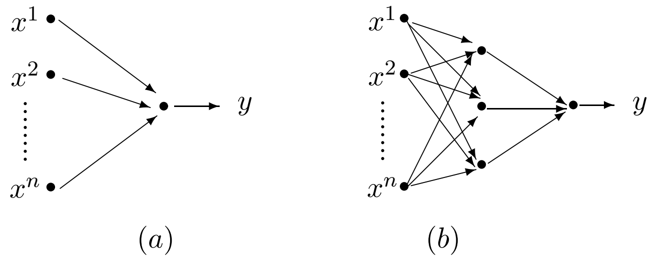


A neural network may be simply viewed as a particular type of function, which are flexible enough to fit fairly general functions. A neural network may be simply understood using the standard metaphor of human brain. There is an input, x, which is processed by a node. Each node is actually a function that transforms the input into an output which is then itself passed to another node. For instance, panel (a) of figure 3.3, borrowed from Judd [1998], illustrates the *single–layer* neural network whose functional form is

$$G(x,\theta) = h\left(\sum_{i=1}^{n} \theta_i g(x^{(i)})\right)$$

where x is the vector of inputs, and h and g are scalar functions. A common choice for g in the case of the dingle layer neural network is $g(x) = x$. Therefore, if we set h to be identity too, we are back to the standard OLS model with monomials. A second a perhaps much more interesting type of neural

Figure 3.3: Neural Networks



network is depicted in panel (b) of figure 3.3. This type of neural network is called the *hidden–layer feedforward* network. In this specification, we are closer to the idea of network as the transformed input is fed to another node that process it to get information. The associated function form is given by

$$G(x,\theta,\gamma) = f\left(\sum_{j=1}^{m} \gamma_j h\left(\sum_{i=1}^{n} \theta_i^j g(x^{(i)})\right)\right)$$

In this case, h is called the *hidden–layer activation function.* h should be a "squasher" function — *i.e.* a monotically nondecreasing function that maps $\mathbb{R}$ onto $[0;1]$. Three very popular functions are

1. The heaviside step function

$$h(x) = \begin{cases} 1 & \text{for } x \geqslant 0 \\ 0 & \text{for } x < 0 \end{cases}$$

2. The sigmoid function
$$h(x) = \frac{1}{1+\exp(-x)}$$
3. Cumulative distribution functions, for example the normal cdf
$$h(x) = \frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{x} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Obtaining an approximation then simply amounts to determine the set of coefficients $\{\gamma_j, \theta_i^j; i=1\ldots n, j=1,\ldots,m\}$. This can be simply achieved running non–linear least squares, that is solving
$$\min_{\{\theta,\gamma\}} \sum_{\ell=1}^{N} \left(y_\ell - G(x_\ell,\theta,\gamma)\right)^2$$

One particular nice feature of neural networks is that they deliver accurate approximation using only few parameters. This characteristic is related to the high flexibility of the approximating functions they use. Further, as established by the following theorem by Hornik, Stinchcombe and White [1989], neural network are extremely powerful in that they offer a universal approximation method.

Theorem 3 *Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a continuous function to be approximated. Let h be a continuous function, $h : \mathbb{R} \to \mathbb{R}$, such that either (i) $\int_{-\infty}^{\infty} h(x)dx$ is finite and non zero and h is L^p for $1 \leqslant p \leqslant \infty$ or (ii) h is a "squashing" function (nondecreasing, $\lim_{x\to\infty} h(x) = 1$,$\lim_{x\to-\infty} h(x) = 0$). Let*
$$\begin{aligned}\Sigma^n(h) &= \Bigg\{ g : \mathbb{R}^n \to \mathbb{R},\ g(x) = \sum_{j=1}^{n} \theta_j h(w^j.x + a_j), \\ &\qquad a_j, \theta_j \in \mathbb{R},\ \textit{and}\ w^j \in \mathbb{R}^n, w^j \neq 0, m = 1,2,\ldots \Bigg\}\end{aligned}$$
be the set of all possible single hidden–layer feedforward neural networks, using h as the hidden layer activation function. Then, for all $\varepsilon > 0$, probability measure μ, and compact sets $K \subset \mathbb{R}^n$, there is a $g \in \Sigma(h)$ such that
$$\sup_{x\in K} |f(x) - g(x)| \leqslant \varepsilon \ \textit{and} \ \int_K |f(x)-g(x)|d\mu \leqslant \varepsilon$$

This theorem is of great importance as it provides a universal approximation result that states that for a broad class of functions, neural networks deliver an accurate approximation to any continuous function. Indeed, we may use for instance any squashing function of the type we described above, or any simple function that satisfies condition *(i)*. Nevertheless one potential limitation of the approach lies into the fact that we have to conduct non–linear estimation, which may be cumbersome under certain circumstances.

As an example, we will deal with the function

$$F(x) = \min\left(\max\left(-\frac{3}{2}, \left(x - \frac{1}{2}\right)^3\right), 2\right)$$

over the interval $[-3; 3]$ and consider a single hidden–layer feedforward network of the form

$$\widetilde{F}(x, \theta, \omega, \alpha) = \frac{\theta_1}{1 + \exp(-(\omega_1 x + \alpha_1)} + \frac{\theta_2}{1 + \exp(-(\omega_2 x + \alpha_2)}$$

The algorithm is then straightforward

1. Generate N values for $x \in [-3; 3]$ and compute $F(x)$
2. Set initial values for $\Theta_0 = \{\theta_i, \omega_i, \alpha_i; i = 1, 2\}$
3. Compute
$$\sum_{i=1}^{N}(F(x_i) - \widetilde{F}(x_i, \Theta))^2$$
if this quantity is minimal then stop, else update Θ and go back to 2.

The last step can be performed using a non–linear minimizer, such that we are actually performing non–linear least–squares. Figure 3.4 plots the approximation where $N = 1000$ and the solution vector Θ yields the values reported in table 3.2. Note that from the form of the Θ vector, we can deduce that the first layer handle positive values for x, therefore corresponding to the left part of the function, while the second layer takes care of the negative part of the

Table 3.2: Neural Network approximation

θ_1	ω_1	α_1	θ_2	ω_2	α_2
2.0277	6.8424	-10.0893	-1.5091	-7.6414	-2.9238

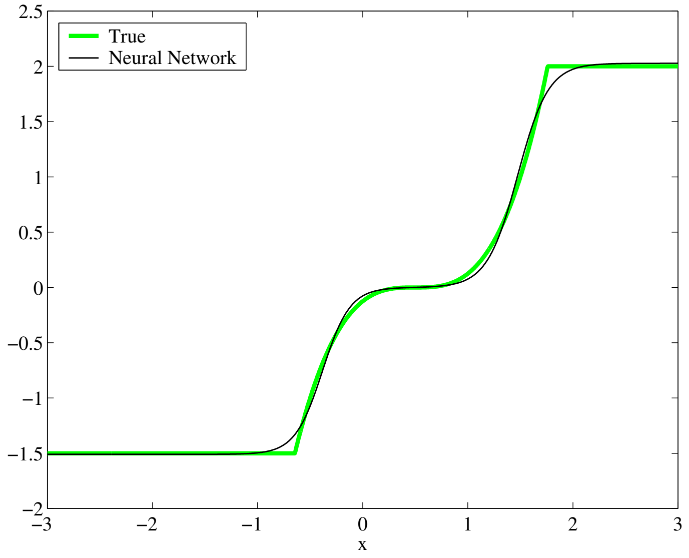
function. Further, it appears that the function is no so badly approximated by this simple neural network, as

$$\mathscr{E}_2 = \frac{1}{N}\left(\sum_{i=1}^{N}(F(x_i) - \widetilde{F}(x_i,\widehat{\Theta})^2\right)^{\frac{1}{2}} = 0.0469$$

and

$$\mathscr{E}_\infty = \max_i \left|F(x_i) - \widetilde{F}(x_i,\widehat{\Theta})\right| = 0.2200$$

Figure 3.4: Neural Network Approximation



All these methods actually relied on regressions. They are simple, but may be either totally unreliable and ill–conditioned in a number of problem, or difficult to compute because they rely on non–linear optimization. We will now consider methods which will actually be more powerful and somehow simpler to implement. They however require introducing some important preliminary concepts, among which that of *orthogonal polynomials*

3.2.1 Orthogonal polynomials

This class of polynomial possesses — by definition — the orthogonality property which will prove to be extremely efficient and useful in a number of problem. For instance, this will solve the multicolinearity problem we encountered in OLS. This property will also greatly simplify the computation of the approximation in a number of problems we will deal with in the sequel. First of all we need to introduce some preliminary concepts.

Definition 4 (Weighting function) *A weighting function $\omega(x)$ on the interval $[a;b]$ is a function that is positive almost everywhere on $[a;b]$ and has a finite integral on $[a;b]$.*

An example of such a weighting function is $\omega(x)=(1-x)^{-1/2}$ over the interval [-1;1]. Indeed, $\lim_{x\to -1}\omega(x)=\sqrt{2}/2$, and $\omega'(x)>0$ such that $\omega(x)$ is positive everywhere over the whole interval. Further

$$\int_{-1}^{1}(1-x^2)^{-1/2}\mathrm{d}x=\arcsin(x)\Big|_{-1}^{1}=\pi$$

Definition 5 (Inner product) *Let us consider two functions $f_1(x)$ and $f_2(x)$ both defined at least on $[a;b]$, the inner product with respect to the weighting function $\omega(x)$ is given by*

$$\langle f_1,f_2\rangle=\int_a^b f_1(x)f_2(x)\omega(x)\mathrm{d}x$$

For example, assume that $f_1(x)=1$, $f_2(x)=x$ and $\omega(x)=(1-x^2)^(-1/2)$, then the inner product over the interval [-1;1] is

$$\langle f_1,f_2\rangle=\int_{-1}^{1}\frac{x}{\sqrt{1-x}}\mathrm{d}x=-\sqrt{1-x^2}\Big|_{-1}^{1}=0$$

Hence, in this case, we have that the inner product of 1 and x with respect to $\omega(w)$ on the interval [-1;1] is identically null. This will actually define the orthogonality property.

Definition 6 (Orthogonal Polynomials) *The family of polynomials $\{P_n(x)\}$ is mutually orthogonal with respect to $\omega(x)$ iff*

$$\langle P_i, P_j \rangle = 0 \text{ for } i \neq j$$

Definition 7 (Orthonormal Polynomials) *The family of polynomials $\{P_n(x)\}$ is mutually orthonormal with respect to $\omega(x)$ iff it is orthogonal and*

$$\langle P_i, P_i \rangle = 1 \text{ for all } i$$

Table 3.3 reports the most common families of orthogonal polynomials (see Judd [1998] for a more detailed exposition) and table 3.4 their recursive formulation

Table 3.3: Orthogonal polynomials (definitions)

Family	$\omega(x)$	$[a;b]$	Definition
Legendre	1	$[-1;1]$	$P_n(x) = \frac{(-1)^n}{2^n n!}\frac{\mathrm{d}^n}{\mathrm{d}x^n}(1-x^2)^n$
Chebychev	$(1-x^2)^{-1/2}$	$[-1;1]$	$T_n(x) = \cos(n\cos^{-1}(x))$
Laguerre	$\exp(-x)$	$[0,\infty)$	$L_n(x) = \frac{\exp(x)}{n!}\frac{\mathrm{d}^n}{\mathrm{d}x^n}(x^n\exp(-x))$
Hermite	$\exp(-x^2)$	$(-\infty,\infty)$	$H_n(x) = (-1)^n\exp(x^2)\frac{\mathrm{d}^n}{\mathrm{d}x^n}\exp(-x^2)$

3.3 Least square orthogonal polynomial approximation

We will now discuss one of the most common approach to approximation that goes back to the easy OLS approach we have seen previously.

Table 3.4: Orthogonal polynomials (recursive representation)

Family	0	1	Recursion
Legendre	$P_0(x) = 1$	$P_1(x) = x$	$P_{n+1}(x) = \frac{2n+1}{n+1}xP_n(x) - \frac{n}{n+1}P_{n-1}(x)$
Chebychev	$T_0(x) = 1$	$T_1(x) = x$	$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$
Laguerre	$L_0(x) = 1$	$L_1(x) = 1 - x$	$L_{n+1}(x) = \frac{2n+1-x}{n+1}L_n(x) - \frac{n}{n+1}L_{n-1}(x)$
Hermite	$H_0(x) = 1$	$H_1(x) = 2x$	$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$

Definition 8 *Let $F : [a,b] \longrightarrow \mathbb{R}$ be a function we want to approximate, and $g(x)$ a polynomial approximation of F. The least square polynomial approximation of F with respect to the weighting function $\omega(w)$ is the degree n polynomial that solves*

$$\min_{\deg(g)\leqslant n} \int_a^b (F(x) - g(x))^2 \omega(x) dx$$

This equation may be understood adopting an econometric point of view, as $\omega(w)$ may be given a weighting matrix interpretation we find in GMM estimation. Indeed, $\omega(x)$ furnishes an indication on how much we care about approximation errors as a function of x. Therefore, setting $\omega(x) = 1$, which amounts to put the same weight on any x then corresponds to a simple OLS approximation.

Let us now assume that $g(x) = \sum_{i=0}^{n} c_i\varphi_i(x)$, where $\{\varphi_k(x)\}_{k=0}^{n}$ is a sequence of orthogonal polynomials, the least square problem rewrites

$$\min_{\{c_i\}_{i=0}^n} \int_a^b \left(F(x) - \sum_{i=0}^{n} c_i\varphi_i(x))\right)^2 \omega(x)\mathrm{d}x$$

for which the first order conditions are given by

$$\int_a^b \left(F(x) - \sum_{i=0}^{n} c_i\varphi_i(x))\right) \varphi_i(x)\omega(x)\mathrm{d}x = 0 \text{ for } i = 0, \ldots, n$$

which yields

$$c_i = \frac{\langle F, \varphi_i \rangle}{\langle \varphi_i, \varphi_i \rangle}$$

Therefore, we have

$$F(x) \simeq \sum_{i=0}^{n} \frac{\langle F, \varphi_i \rangle}{\langle \varphi_i, \varphi_i \rangle} \varphi_i(x)$$

There are several examples of the use of this type of approximation. Fourier approximation is an example of those which is suitable for periodic functions. Nevertheless, I will focus in a coming section on one particular approach, which we will use quite often in the next chapters: Chebychev approximation.

Beyond least square approximation, there exists other approaches that departs from the least square by the norm they use:

- Uniform approximation, which attempt to solve

  $$\lim_{n \to \infty} \max_{x \in [a;b]} |F(x) - p_n(x)| = 0$$

  The main difference between this approach and L^2 approximation is that contrary to L^2 norms that put no restrictions on the approximation on particular points, the uniform approximation imposes that the approximation of F at each x is exact, whereas L^2 approximations juste requires the total error to be as small as possible.

- Minimax approximations, which rest on the L^∞ norm, such that these approximations attempt to find an approximation that provides the best uniform approximation to the function F that is we search the degree n polynomial that achieves

  $$\min_{\deg(g) \leqslant n} \|F(x) - g(x)\|_\infty$$

  There are theorems in approximation theory that states that the quality of this approximation increases polynomially as the degree of polynomials increases, and the polynomial rate of convergence is faster for smoother functions. Nevertheless, if this approximation can be as good as needed for F, no use of its derivatives is made, such that the derivatives may be very poorly approximated. Finally, such approximation methods are extremely difficult to compute.

3.4 Interpolation methods

Up to now, we have seen that there exist methods to compute the value of a function at some particular points, but in most of the cases we are not only interested by the function at some points but also between these points. This is the problem of interpolation.

3.4.1 Linear interpolation

This method is totally straightforward and known to everybody. Assume you have a collection of data points $\mathscr{C} = \{(x_i, y_i)|i = 1, \ldots, n\}$. Then for any $x \in [x_{i-1}, x_i]$, we can compute y as

$$y = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}x + \frac{x_i y_{i-1} - x_{i-1} y_i}{x_i - x_{i-1}}$$

Although, such approximations have proved to be very useful in a lot of applications, it can be particularly inefficient for different reasons:

1. It does not deliver **an** approximating function but rather a collection of approximations for each interval;

2. it requires to identify the interval where the approximation is to be computed, which may be particularly costly when the interval is not uniform;

3. it can obviously perform very badly for non–linear functions.

Therefore, there obviously exist alternative interpolation methods that perform better, but which are not necessarily more efficient.

3.4.2 Lagrange interpolation

This type of approximation is quite demanding as it consider a collection of data $\mathscr{C} = \{(x_i, y_i)|i = 1, \ldots, n\}$ with distinct x_i — called the Lagrange data. Lagrange interpolation amounts to find a degree $n-1$ polynomial $P(x)$, such that $y_i = P(x_i)$. Therefore, the method is exact for each point. Lagrange interpolating polynomials are defined by

$$P_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Note that $P_i(x_i) = 1$ and $P_i(x_j) = 0$. The interpolation is then given by

$$P(x) = \sum_{i=1}^{n} y_i P_i(x)$$

The obvious problem that we can immediately see from this formula is that if the number of point is high enough, this type of interpolation is totally untractable. Indeed, just to compute a single $P_i(x)$ this already requires $2(n-1)$ substractions, n multiplications. Then this has to be constructed for the n data points to compute all needed $P_i(x)$. Then to compute $P(x)$ we need n additions and n multiplications. Therefore, this requires $3n^2$ operations! Therefore, one actually may actually attempt to compute directly:

$$P(x) = \sum_{i=0}^{n-1} \alpha_i x^i$$

which may be obtained by solving the linear system

$$\left\{ \begin{array}{rcl} y_1 & = & \alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 + \ldots + \alpha_{n-1} x_1^{n-1} \\ y_2 & = & \alpha_0 + \alpha_1 x_2 + \alpha_2 x_2^2 + \ldots + \alpha_{n-1} x_2^{n-1} \\ & \vdots & \\ y_n & = & \alpha_0 + \alpha_1 x_n + \alpha_2 x_n^2 + \ldots + \alpha_{n-1} x_n^{n-1} \end{array} \right.$$

or

$$A\alpha = y$$

where $\alpha = \{\alpha_0, \alpha_1, \dots, \alpha_{n-1}\}'$ and A is the so–called *Vandermonde matrix* for x_i, $i = 1, \dots, n$

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}$$

If the Lagrange formula guarantees the existence of the interpolation, the following theorem guarantees its uniqueness.

Theorem 4 *Provided the interpolating points are distincts, there is a unique solution to the Lagrange interpolation problem.*

The proof of the theorem may be sketched as follows. Assume that besides $P(x)$, there exists another interpolating polynomial $P^\star(x)$ of degree at most $n-1$ that also interpolates the n data points. Then, by construction, $P(x) - P^\star(x)$ is at most of degree $n-1$ and is zeros at all n nodes. But the only polynomial of degree $n-1$ that has n zeros is 0, such that $P(x) = P^\star(x)$. Here again, the method may be quite demanding from a computational point of view, as we have to invert a matrix of size $(n \times n)$. There then exist much more efficient methods, and in particular the so–called Chebychev approximation that works very well for smooth functions.

3.5 Chebychev approximation

Chebychev approximation uses ... Chebychev polynomials as a basis for the polynomials (see figure 3.5). These polynomials are described by the recursion

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \text{ with } T_0(x) = 1, T_1(x) = x$$

which admits as solution

$$T_n(x) = \cos(n \cos^{-1}(x))$$

as aforementioned these polynomials form an orthogonal basis with respect to the weighting function $\omega(x) = (1-x^2)^{-1/2}$ over the interval $[-1; 1]$. Nevertheless, this interval may be generalized to $[a; b]$, by transforming the data using the formula

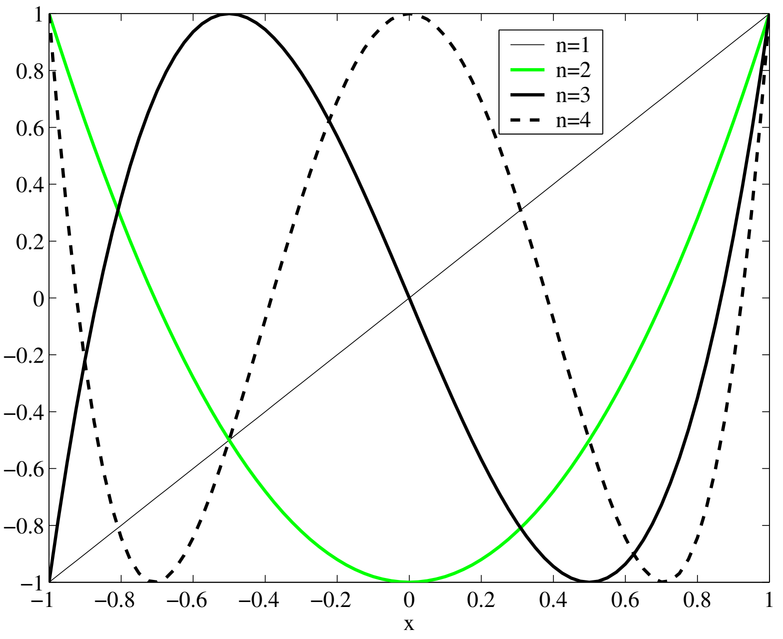
$$x = 2\frac{y-a}{b-a} - 1 \text{ for } y \in [a;b]$$

Beyond the standard orthogonality property, Chebychev polynomials exhibit a discrete orthogonality property which writes as

$$\sum_{i=1}^{n} T_i(r_k)T_j(r_k) = \begin{cases} 0 & \text{for } i \neq j \\ n & \text{for } i = j = 0 \\ \frac{n}{2} & \text{for } i = j \neq 0 \end{cases}$$

where r_k, $k = 1, \ldots, n$ are the roots of $T_n(x) = 0$.

Figure 3.5: Chebychev polynomials



A first theorem will state the usefulness of Chebychev approximation.

Theorem 5 *Assume F is a C^k function over the interval $[-1;1]$, and let*

$$c_i = \frac{2}{\pi}\int_{-1}^{1} \frac{F(x)T_i(x)}{\sqrt{1-x^2}}dx \textit{ for } i = 0 \ldots n$$

and

$$g_n(x) \equiv \frac{c_0}{2} + \sum_{i=1}^{n} c_i T_i(x)$$

Then, there exists $\varepsilon < \infty$ such that for all $n \geqslant 2$

$$\|F(x) - g_n(x)\|_\infty \leqslant \varepsilon \frac{\log(n)}{n^k}$$

This theorem is of great importance as it actually states that the approximation $g_n(x)$ will be as close as we might want to $F(x)$ as the degree of approximation n increases to ∞. In effect, since the approximation error is bounded by above by $\varepsilon\frac{\log(n)}{n^k}$ and since the latter expression tends to zero as n tends toward ∞, we have that $g_n(x)$ converges uniformly to $F(x)$ as n increases. Further, the next theorem will establish a useful property on the coefficients of the approximation.

Theorem 6 *Assume F is a C^k function over the interval $[-1;1]$, and admits a Chebychev representation*

$$F(x) = \frac{c_0}{2} + \sum_{i=1}^{\infty} c_i T_i(x)$$

then, there exists c such that

$$|c_i| \leqslant \frac{c}{i^k} \textit{ for } j \geqslant 1$$

This theorem is particularly important as it states that the smoother the function to be approximated is (the greater k is), the faster is the pace at which coefficients will drop off. In other words, we will be able to achieve a high enough accuracy using less coefficients.

At this point, we have established that Chebychev approximation can be accurate for smooth functions, but we still do not know how to proceed to get a good approximation. In particular, a very important issue is the selection of interpolating data points, the so–called *nodes*. This is the main problem of interpolation: *how to select nodes such that we minimize the interpolation error?* The answer to this question is particularly simple in the case of Chebychev interpolation: the nodes should be the zeros of the n^{th} degree Chebychev polynomial.

We are then endowed with data points to compute the approximation. Using $m > n$ data points, we can compute the $(n-1)^{th}$ order Chebychev approximation relying on the Chebychev regression algorithm we will now describe in details. When $m = n$, the algorithm reduces to the so–called Cebychev interpolation formula. Let us consider the following problem: Let $F : [a;b] \longrightarrow \mathbb{R}$, let us construct a degree $n \leqslant m$ polynomial approximation of F on $[a;b]$:

$$G(x) \equiv \sum_{i=0}^{n} \alpha_i T_i \left(2\frac{x-a}{b-a} - 1 \right)$$

1. Compute $m \geqslant n+1$ Chebychev interpolation nodes on $[-1;1]$, which are the roots of the degree m Chebychev polynomial

$$r_k = -cos\left(\frac{2k-1}{2m}\pi\right) \text{ for } k = 1\ldots,m$$

2. Adjust the nodes, r_k, to fit in the $[a;b]$ interval

$$x_k = (r_k+1)\frac{b-a}{2} + a \text{ for } k = 1\ldots,m$$

3. Evaluate the function F at each approximation node x_k, to get a collection of ordinates

$$y_k = F(x_k) \text{ for } k = 1\ldots,m$$

4. Compute the collection of $n+1$ coefficients $\alpha = \{\alpha_i; i = 0\ldots n\}$ as

$$\alpha_i = \frac{\displaystyle\sum_{k=1}^{m} y_k T_i(r_k)}{\displaystyle\sum_{k=1}^{m} T_i(r_k)^2}$$

5. Form the approximation

$$G(x) \equiv \sum_{i=0}^{n} \alpha_i T_i \left(2\frac{x-a}{b-a} - 1 \right)$$

Note that step 4 actually can be interpreted in terms of an OLS problem as — because of the orthogonality property of the Chebychev polynomials — α_i is given by

$$\alpha_i = \frac{\text{cov}(y, T_i(x))}{\text{var}(T_i(x))}$$

which may be recast in matricial notations as

$$\alpha = (X'X)^{-1}X'Y$$

where

$$X = \begin{pmatrix} T_0(x_1) & T_1(x_1) & \cdots & T_n(x_1) \\ T_0(x_2) & T_1(x_2) & \cdots & T_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ T_0(x_m) & T_1(x_m) & \cdots & T_n(x_m) \end{pmatrix} \text{ and } Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

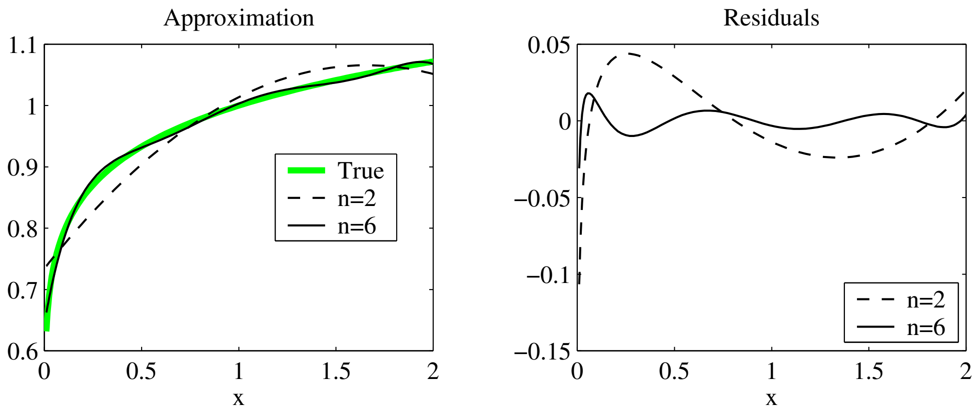
We now report two examples implementing the algorithm. The first one deals with a smooth function of the type $F(x) = x^\theta$. The second one evaluate the accuracy of the approximation in the case of a non–smooth function: $F(x) = \min(\max(-1.5, (x - 1/2)^3), 2)$.

In the case of the smooth function, we set $\theta = 0.1$ and approximate the function over the interval $[0.01; 2]$. We select 100 nodes and evaluate the accuracy of degree 2 and 6 approximation. Figure 3.6 reports the true function and the corresponding approximation, table 3.5 reports the coefficients. As can be seen from the table, adding terms in the approximation does not alter the coefficients of lower degree. This just reflects the orthogonality properties of the Chebychev polynomials, that we saw in the formula determining each α_i. This is of great importance, as it states that once we have obtained a high order approximation, obtaining lower orders is particularly simple. This is the *economization principle*. Further, as can be seen from the figure, a "good approximation" to the function is obtained at rather low degrees. Indeed, the difference between the function and its approximation at order 6 is already good.

Table 3.5: Chebychev Coefficients: Smooth function

	n=2	n=6
c_0	0.9547	0.9547
c_1	0.1567	0.1567
c_2	-0.0598	-0.0598
c_3	–	0.0324
c_4	–	-0.0202
c_5	–	0.0136
c_6	–	-0.0096

Figure 3.6: Smooth function: $F(x) = x^{0.1}$



Matlab Code: Smooth Function Approximation

```
m   = 100;                        % number of nodes
n   = 6;                          % degree of polynomials
rk  = -cos((2*[1:m]-1)*pi/(2*m)); % Roots of degree m polynomials
a   = 0.01;                       % lower bound of interval
b   = 2;                          % upper bound of interval
xk  = (rk+1)*(b-a)/2+a;           % nodes
Y   = xk.^0.1;                    % compute the function at nodes
%
% Builds Chebychev polynomials
%
Tx      = zeros(m,n+1);
Tx(:,1) = ones(m,1);
Tx(:,2) = xk(:);
for i=3:n+1;
    Tx(:,i) = 2*xk(:).*Tx(:,i-1)-Tx(:,i-2);
end
%
% Chebychev regression
%
alpha   = X\Y;                    % compute the approximation coefficients
G       = X*a;                    % compute the approximation
```

In the case of the non–smooth function we consider ($F(x) = \min(\max(-1.5, (x-1/2)^3), 2)$), the coefficients remain large even at degree 15 and the residuals remain high at order 15. This actually indicates that Chebychev approximations are well suited for smooth functions, but have more difficulties to capture kinks. Nevertheless, increasing the order of the approximation drastically, we can achieve a much better approximation.

In the later case, it seems that a piecewise approximation would perform better. Indeed, in this case, we may compute 3 approximation

- for $x \in (-\infty, \underline{x})$, $G(x) = -1.5$
- for $x \in (\underline{x}, \overline{x})$, $G(x) \equiv \sum_{i=0}^{n} \beta_i T_i\left(2\frac{x-a}{b-a} - 1\right)$
- for $x \in (\overline{x}, \infty)$, $G(x) = 2$

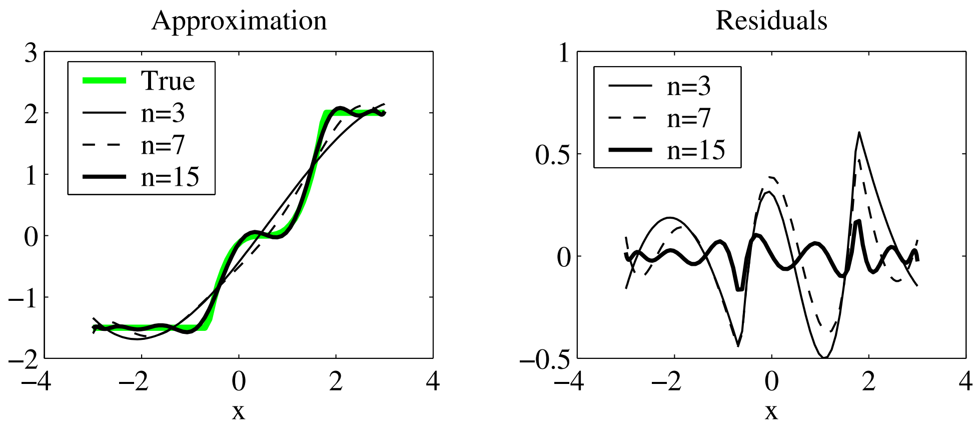
where $\underline{x}$ is such that

$$(\underline{x} - 1/2)^3 = -1.5$$

Table 3.6: Chebychev Coefficients: Non–smooth function

	n=3	n=7	n=15
c_0	-0.0140	-0.0140	-0.0140
c_1	2.0549	2.0549	2.0549
c_2	0.4176	0.4176	0.4176
c_3	-0.3120	-0.3120	-0.3120
c_4	–	-0.1607	-0.1607
c_5	–	-0.0425	-0.0425
c_6	–	-0.0802	-0.0802
c_7	–	0.0571	0.0571
c_8	–	–	0.1828
c_9	–	–	0.0275
c_{10}	–	–	-0.1444
c_{11}	–	–	-0.0686
c_{12}	–	–	0.0548
c_{13}	–	–	0.0355
c_{14}	–	–	-0.0012
c_{15}	–	–	0.0208

Figure 3.7: Non–smooth function: $F(x) = \min(\max(-1.5, (x - 1/2)^3), 2)$



and $\overline{x}$ satisfies

$$(\overline{x} - 1/2)^3 = 2$$

In such a case, the approximation would be perfect with $n = 3$. This suggests that piecewise approximation may be of interest in a number of cases.

3.6 Piecewise interpolation

We have actually already seen piecewise approximation method: the linear interpolation method. But there exist more powerful and efficient method that use *splines*. A spline can be any smooth function that is piecewise polynomial, but most of all it should be smooth at all nodes.

Definition 9 *A function $S(x)$ on an interval $[a;b]$ is a spline of order n if*

1. *$S(x)$ is a $\mathscr{C}^{n-2}$ function on $[a;b]$,*
2. *There exist a collection of ordered nodes $a = x_0 < x_1 < \ldots < x_m = b$ such that $S(x)$ is a polynomial of order $n-1$ on each interval $[x_i;x_{i+1}]$, $i = 0, \ldots, m-1$*

Examples of spline functions are

- Cubic splines: These splines functions are splines of order 4. These splines are the most popular and are of the form

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \text{ for } x \in [x_i; x_{i+1}]$$

- B^0–splines: These functions are splines of order 1:

$$B_i^0(x) = \begin{cases} 0, & x < x_i \\ 1, & x_i \leqslant x \leqslant x_{i+1} \\ 0, & x > x_{i+1} \end{cases}$$

- B^1–splines: These functions are splines of order 2 that actually describe tent functions:

$$B_i^1(x) = \begin{cases} 0, & x < x_i \\ \frac{x-x_i}{x_{i+1}-x_i}, & x_i \leqslant x \leqslant x_{i+1} \\ \frac{x_{i+2}-x}{x_{i+2}-x_{i+1}}, & x_{i+1} \leqslant x \leqslant x_{i+2} \\ 0, & x > x_{i+2} \end{cases}$$

  Such a spline reaches a peak at $x = x_{i+1}$ and is upward (downward) sloping for $x < x_{i+1}$ ($x > x_{i+1}$).

- Higher order spline functions are defined by the recursion:

$$B_i^n(x) = \left(\frac{x - x_i}{x_{i+n} - x_i}\right) B_i^{n-1}(x) + \left(\frac{x_{i+n+1} - x}{x_{i+n+1} - x_{i+1}}\right) B_{i+1}^{n-1}(x)$$

Cubic splines are the most widely used splines to interpolate functions. Therefore, we will describe the method in greater details in such a case. Let us assume that we are endowed with Lagrange data — *i.e.* a collection of nodes x_i and corresponding values for the function $y_i = F(x_i)$ to interpolate: $\{(x_i, y_i) : i = 0 \ldots n\}$. We therefore have in hand n intervals $[x_i; x_{i+1}]$, $i = 0, \ldots, n-1$ for which we search n cubic splines

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \text{ for } x \in [x_i; x_{i+1}]$$

The problem is then to select $4n$ coefficients $\{a_i, b_i, c_i, d_i : i = 0, \ldots, n-1\}$ using $n+1$ nodes. We therefore need $4n$ identification conditions to identify these $4n$ coefficients.

The first set of restrictions is given by the collection of restrictions imposing that the spline approximation is exact on the nodes

$$S(x_i) = y_i \text{ for } i = 0, \ldots, n-1 \text{ and } S_{n-1}(x_n) = y_n$$

which amounts to impose

$$a_i = y_i \text{ for } i = 0, \ldots, n-1 \tag{3.6}$$

and

$$a_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3 = y_n \quad (3.7)$$

The second set of restrictions imposes continuity of the function on the upper bound of each interval

$$S_i(x_i) = S_{i-1}(x_i) \text{ for } i = 1, \ldots, n-1$$

which implies, noting $h_i = x_i - x_{i-1}$

$$a_i = a_{i-1} + b_{i-1}h_i + c_{i-1}h_i^2 + d_{i-1}h_i^3 \text{ for } i = 1, \ldots, n-1 \quad (3.8)$$

This furnishes $2n$ identification restrictions, such that $2n$ additional restrictions are still needed. Since we are dealing with a cubic spline interpolation, this requires the approximation to be $\mathscr{C}^2$, implying that first and second order derivatives should be continuous. This yields the following $n-1$ conditions for the first order derivatives

$$S'_i(x_i) = S'_{i-1}(x_i) \text{ for } i = 1, \ldots, n-1$$

or

$$b_i = b_{i-1} + 2c_{i-1}h_i + 3d_{3i-1}h_i^2 \text{ for } i = 1, \ldots, n-1 \quad (3.9)$$

and the additional $n-1$ conditions for the second order derivatives

$$S''_i(x_i) = S''_{i-1}(x_i) \text{ for } i = 1, \ldots, n-1$$

or

$$2c_i = 2c_{i-1} + 6d_{3i-1}h_i \text{ for } i = 1, \ldots, n-1 \quad (3.10)$$

Equations (3.6)–(3.10) therefore define a system of $4n-2$ equations, such that we are left with 2 degrees of freedom. Hence, we have to impose 2 additional conditions. There are several ways to select such conditions

1. Natural cubic splines impose that the second order derivatives $S''_0(x_0) = S''_n(x_n) = 0$. Note that the latter is actually not to be calculated in our

problem, nevertheless this imposes conditions on both c_0 and c_n which will be useful in the sequel. In fact it imposes

$$c_0 = c_n = 0$$

An interpretation of this condition is that the cubic spline is represented by the tangent of S at x_0 and x_n

2. Another way to fix $S(x)$ would be to use potential information on the slope of the function to be approximated. In other words, one may set

   $$S'_0(x_0) = F'(x_0) \text{ and } S'_{n-1}(x_n) = F'(x_n)$$

   This is the so–called *Hermite spline.* However, in a number of situation such information on the derivative of F is either not known or does not exist (think of F not being differentiable at some points), such that further source of information is needed. One can then rely on an approximation of the slope by the secant line. This is what is proposed by the*secant Hermite spline*, which amounts to approximate $F'(x_0)$ and $F'(x_n)$ by the secant line over the corresponding interval:

   $$S'_0(x_0) = \frac{S_0(x_1) - S_0(x_0)}{x_1 - x_0} \text{ and } S'_{n-1}(x_n) = \frac{S_{n-1}(x_n) - S_{n-1}(x_{n-1})}{x_n - x_{n-1}}$$

   But from the identification scheme, we have $S_0(x_1) = S_1(x_1) = y_1$ and $S_{n-1}(x_n) = y_n$, such that we get

   $$b_0 = (y_1 - y_0)/h1 \text{ and } b_{n-1} = (y_n - y_{n-1})/h_n$$

Let us now focus on the natural cubic spline approximation, which imposes $c_0 = c_n = 0$. First, note that the system (3.6)–(3.9) has a recursive form, such that from (3.9) we can get

$$d_{i-1} = \frac{1}{3h_i}(c_i - c_{i-1}) \text{ for } i = 1, \ldots, n-1$$

Plugging this results in (3.9),we get

$$b_i - b_{1i-1} = 2c_{i-1}h_i + (c_i - c_{i-1})h_i = (c_i + c_{i-1})h_i \text{ for } i = 1, \ldots, n-1$$

and, (3.8) becomes

$$\begin{aligned}
a_i - a_{i-1} &= b_{i-1}h_i + c_{i-1}h_i^2 + \frac{1}{3}(c_i - c_{i-1})h_i^2 \\
&= b_{i-1}h_i + \frac{1}{3}(c_i + 2c_{i-1})h_i^2 \text{ for } i = 1, \dots, n-1
\end{aligned}$$

which we may rewrite as

$$\frac{a_i - a_{i-1}}{h_i} = b_{i-1} + \frac{1}{3}(c_i + 2c_{i-1})h_i \text{ for } i = 1, \dots, n-1$$

Likewise, we have

$$\frac{a_{i+1} - a_i}{h_{i+1}} = b_i + \frac{1}{3}(c_{i+1} + 2c_i)h_{i+1} \text{ for } i = 0, \dots, n-2$$

substracting the last two equations, when defined, we get

$$\frac{a_{i+1} - a_i}{h_{i+1}} - \frac{a_i - a_{i-1}}{h_i} = b_i - b_{i-1} + \frac{1}{3}(c_{i+1} + 2c_i)h_{i+1} - \frac{1}{3}(c_i + 2c_{i-1})h_i$$

for $i = 1, \dots, n-2$, which is then given, taking (3.6) and (3.7) into account, by

$$\frac{3}{h_{i+1}}(y_{i+1} - y_i) - \frac{3}{h_i}(y_i - y_{i-1}) = h_i c_{i-1} + 2(h_i + h_{i+1})c_i + h_{i+1}c_{i+1}$$

for $i = 1, \dots, n-2$. We however have the additional $n-1$–th identification restriction that imposes $c_0 = 0$ and the last restriction $c_n = 0$ We therefore end–up with a system of the form

$$Ac = B$$

where

$$A = \begin{pmatrix}
2(h_0+h_1) & h_1 & & & & \\
h_1 & 2(h_1+h_2) & h_2 & & & \\
& h_2 & 2(h_2+h_3) & h_3 & & \\
& & \ddots & & & \\
& & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} \\
& & & h_{n-2} & 2(h_{n-2}+h_{n-1})
\end{pmatrix}$$

$$c=\begin{pmatrix} c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} \text{ and } B=\begin{pmatrix} \frac{3}{h_1}(y_2-y_1)-\frac{3}{h_0}(y_1-y_0) \\ \vdots \\ \frac{3}{h_{n-1}}(y_n-y_{n-1})-\frac{3}{h_{n-2}}(y_{n-1}-y_{n-2}) \end{pmatrix}$$

The matrix A is then said to be tridiagonal (and therefore sparse) and is also symmetric and elementwise positive. It is hence positive definite and therefore invertible. We then got all the c_i, $i=1,\dots,n-1$ and can compute the b's and d's as

$$b_{i-1}=\frac{y_i-y_{i-1}}{h_i}-\frac{1}{3}(c_i+2c_{i-1})h_i \text{ for } i=1,\dots,n-1 \text{ and } b_{n-1}=\frac{y_n-y_{n-1}}{h_n}-\frac{2c_{n-1}}{3h_n}$$

and

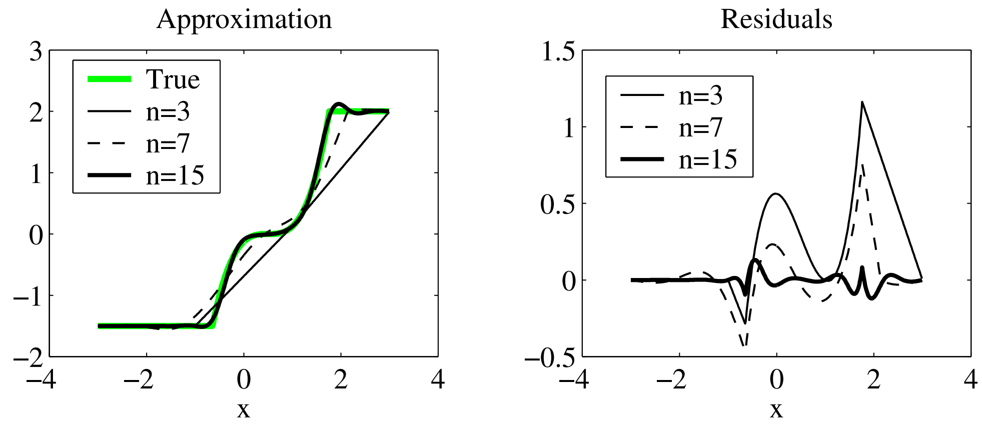
$$d_{i-1}=\frac{1}{3h_i}(c_i-c_{i-1}) \text{ for } i=1,\dots,n-1 \text{ and } d_{n-1}=-\frac{c_{n-1}}{3h_n}$$

finally we have had $a_i=y_i$, $i=0,\dots,n-1$ from the very beginning.

Once the approximation is obtained, the evaluation of the approximation has to be undertaken. The only difficult part in this job is to identify the interval the value of the argument of the function we want to evaluate belongs to — *i.e.* we have to find $i\in\{0,\dots,n-1\}$ such that $x\in[x_i,x_{i+1}]$. Nevertheless, as long as the nodes are generated using an invertible formula, there will be no cost to determine the interval. Most of the time, a uniform grid is used, such that the interval $[a;b]$ is divided using the linear scheme $x_i=a+i\Delta$, where $\Delta=(b-a)/(n-1)$, and $i=0,\dots,n-1$. In such a case, it is particularly simple to determine the interval as i is given by $\mathsf{E}[(x-a)/\Delta]$. Nevertheless there are some cases where it may be efficient to use non–uniform grid. For instance, in the case of the function we consider it would be useful to consider the following simple 4 nodes grid $\{-3,0.5-\sqrt[3]{1.5},0.5+\sqrt[3]{2},3\}$, as taking this grid would yield a perfect approximation (remember that the central part of the function is cubic!)

As an example of spline approximation, figure 3.8 reports the spline approximation to the non–smooth function $F(x)=\min(\max(-1.5,(x-1/2)^3),2)$ considering a uniform grid over the [-3;3] interval with 3, 7 and 15 nodes. In

Figure 3.8: Cubic spline approximation



order to gauge the potential of spline approximation, we report in the upper panel of figure 3.9 the L^2 and L^∞ error of approximation. The L^2 approximation error is given by $\|F(x)-S(x)\|$ while the L^∞ is given by $\max|F(x)-S(x)|$. It clearly appears that increasing the number of nodes improves the approximation in that the error is driven to zero. Nevertheless, it also appears that convergence is not monotonic in the case of the L^∞ error. This is actually related to the fact that F, in this case is not even $\mathscr{C}^1$ on the overall interval. In fact, as soon as we consider a smooth function this convergence is monotonic, as can be seen from the lower panel that report it for the function $F(x)=x^{0.1}$ over the interval [0.01;2]. This actually illustrates the following result.

Theorem 7 *Let F be a $\mathscr{C}^4$ function over the interval $[x_0;x_n]$ and S its cubic spline approximation on $\{x_0,x_1,\ldots,x_n\}$ and let $\delta \geqslant \max_i\{x_i-x_{i-1}\}$, then*

$$\|F-S\|_\infty \leqslant \frac{5}{384}\|F^{(4)}\|_\infty\delta^4$$

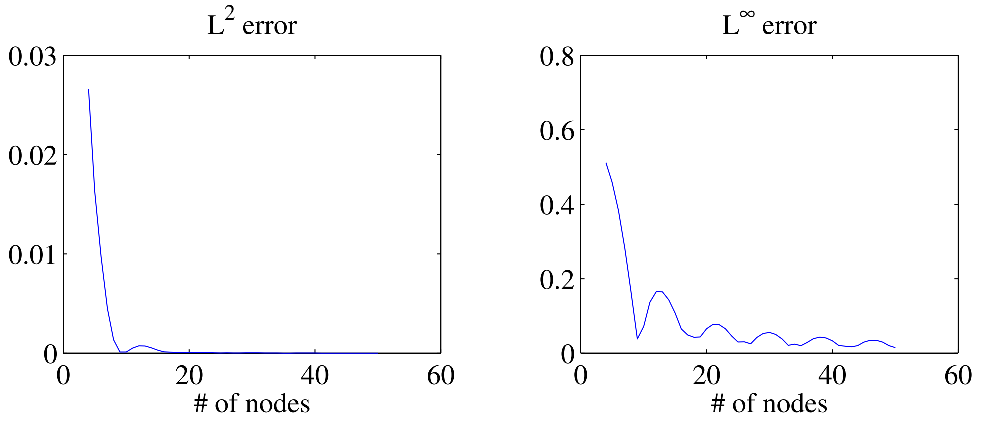
and

$$\|F'-S'\|_\infty \leqslant \frac{9+\sqrt(3)}{216}\|F^{(4)}\|_\infty\delta^3$$

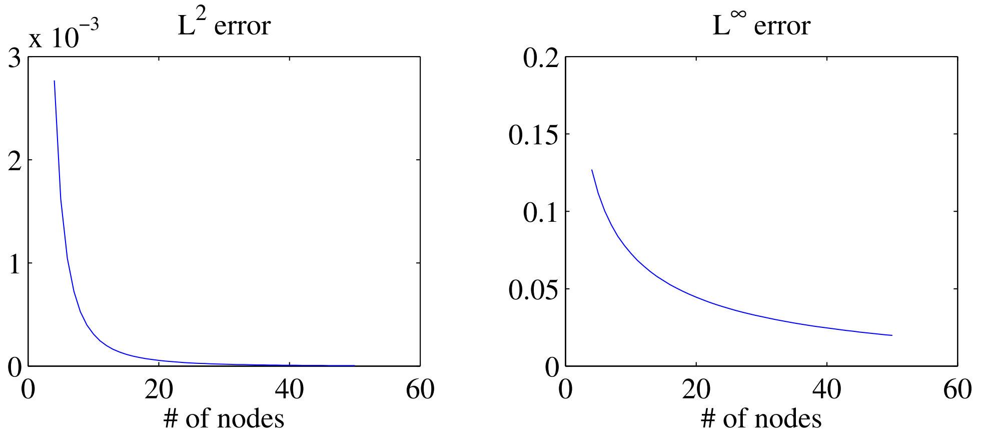
This theorem actually gives upper bounds to spline approximation, and indicates that these bounds decrease at a fast pace (power of 4) as the number of

Figure 3.9: Approximation errors

$F(x) = \min(\max(-1.5, (x - 1/2)^3), 2)$ over $[-3; 3]$



$F(x) = x^{0.1}$ over $[0.01; 2]$



nodes increases (as δ diminishes). Splines are usually viewed as a particularly good approximation method for two main reasons:

1. A good approximation may be achieved even for functions that are not $\mathscr{C}^\infty$ or that do not possess high order derivatives. Indeed, as indicated in theorem 7, the error term basically depends only on fourth order derivatives, such that even if the fifth order derivative were badly behaved then an accurate approximation may be obtained.

2. Evaluation of splines is particularly cheap as they involve most of the time at most cubic polynomials, the only costly part being the interval search step.

Matlab Code: Cubic Spline Approximation

```
nbx = 8;                                    % number of nodes
a   = -3;                                   % lower bound of interval
b   = 3;                                    % upper bound of interval
dx  = (b-a)/(n-1);                          % step in the grid
x   = [a:dx:b];                             % grid points
y   = min(max(-1.5,(x(i)-0.5)^3),2);

A   = spalloc((nbx-2),(nbx-2),3*nbx-8);      % creates sparse matrix A
B   = zeros((nbx-2),1);                      % creates vector B
A(1,[1 2])=[2*(dx+dx) dx];
for i=2:nbx-3;
   A(i,[i-1 i i+1])=[dx 2*(dx+dx) dx];
   B(i)=3*(y(i+2)-y(i+1))/dx-3*(y(i+1)-y(i))/dx;
end
A(nbx-2,[nbx-3 nbx-2])=[dx 2*(dx+dx)];

c   = [0;A\B];
a   = y(1:nbx-1);
b   = (y(2:nbx)-y(1:nbx-1))/dx-dx*([c(2:nbx-1);0]+2*c(1:nbx-1))/3;
d   = ([c(2:nbx-1);0]-c(1:nbx-1))/(3*dx);
S   = [a';b';c(1:nbx-1)';d'];                  % Matrix of spline coefficients
```

One potential problem that may arise with the type of method we have developed until now is that we have not imposed any particular restriction on the shape of the approximation relative to the true function. This may be of great importance in some cases. Let us assume for instance that we need to approximate the function $F(x_t)$ that characterizes the dynamics of variable x

in the following backward looking dynamic equation:

$$x_{t+1} = F(x_t)$$

Assume F is a concave function that is costly to compute, such that it is beneficial to approximate the function. However, as we have already seen from the previous examples, many methods generate oscillations in the approximation. This can create some important problems as it implies that the approximation is not strictly concave, which is in turn crucial to characterize the dynamics of variable x. Further, the approximation of a strictly increasing function may be locally decreasing. All this may create some divergent path, or even generate some spurious steady state, and therefore spurious dynamics. It is therefore crucial to develop shape preserving methods — preserving in particular the curvature and monotonicity properties — for such cases.

3.7 Shape preserving approximations

In this section, we will see an approximation method that preserves the shape of the function we want to approximate. This method was proposed by Schumaker [1983] and essentially amounts to exploit some information on both the level and the slope of the function to be approximated to build a smooth approximation. We will deal with two situations. The first one — Hermite interpolation — assumes that we have information on both the level and the slope of the function to approximate. The second one — that uses Lagrange data — assumes that no information on the slope of the function is available. Both method was originally developed using quadratic splines.

3.7.1 Hermite interpolation

This method assumes that we have information on both the level and the slope of the function to be approximated. Assume we want to approximate the function F on the interval $[x_1, x_2]$ and we know $y_i = F(x_i)$ and $z_i = F'(x_i)$, $i = 1, 2$. Schumaker proposes to build a quadratic function $S(x)$ on $[x_1; x_2]$

that satisfies

$$S(x_i) = y_i \text{ and } S'(x_i) = z_i \text{ for } i = 1, 2$$

Schumaker establishes first that

Lemma 1 *If*

$$\frac{z_1 + z_2}{2} = \frac{y_2 - y_1}{x_2 - x_1}$$

then the quadratic form

$$S(x) = y_1 + z_1(x - x_1) + \frac{z_2 - z_1}{2(x_2 - x_1)}(x - x_1)^2$$

satisfies $S(x_i) = y_i$ and $S'(x_i) = z_i$ for $i = 1, 2$.

The construction of this function is rather appealing. If z_1 and z_2 have the same sign then $S'(x)$ has the same sign as z_1 and z_2 over $[x_1; x_2]$:

$$S'(x) = z_1 + \frac{(z_2 - z_1)}{(x_2 - x_1)}(x - x_1)$$

Hence, if F is monotically increasing (decreasing) on the interval $[x_1; x_2]$, so is $S(x)$. Further, $z_1 > z_2$ ($z_1 < z_2$) indicates concavity (convexity), which $S(x)$ satisfies as $S''(x) = (z_2 - z_1)/(x_2 - x_1) < 0$ (> 0).

However, the conditions stated by this lemma are extremely stringent and do not usually apply, such that we have to adapt the procedure. This may be done by adding a node between x_1 and x_2 and construct another spline that satisfies the lemma.

Lemma 2 *For every $x^\star \in (x_1, x_2)$ there exist a unique quadratic spline that solves*

$$S(x_i) = y_i \textit{ and } S'(x_i) = z_i \textit{ for } i = 1, 2$$

with a node at $x^\star$. This spline is given by

$$S(x) = \begin{cases} \alpha_{01} + \alpha_{11}(x - x_1) + \alpha_{21}(x - x_1)^2 & \textit{for } x \in [x_1; x^\star] \\ \alpha_{02} + \alpha_{12}(x - x^\star) + \alpha_{22}(x - x^\star)^2 & \textit{for } x \in [x^\star; x_2] \end{cases}$$

where

$$\begin{array}{lll} \alpha_{01} = y_1 & \alpha_{11} = z_1 & \alpha_{21} = \frac{\overline{z} - z_1}{2(x^\star - x_1)} \\ \alpha_{02} = y_1 + \frac{\overline{z} + z_1}{2}(x^\star - x_1) & \alpha_{12} = \overline{z} & \alpha_{22} = \frac{z_2 - \overline{z}}{2(x_2 - x^\star)} \end{array}$$

where $\overline{z} = \frac{2(y_2 - y_1) - (z_1(x^\star - x_1) + z_2(x_2 - x^\star)}{x_2 - x_1}$

If the later lemma fully characterized the quadratic spline, it gives no information on $x^\star$ which therefore remains to be selected. $x^\star$ will be set such that the spline matches the desired shape properties. First note that if z_1 and z_2 are both positive (negative), then $S(x)$ is monotone if and only if $z_1\overline{z} \geqslant 0$ ($\leqslant 0$) which is actually equivalent to

$$2(y_2 - y_1) \gtreqless (x^\star - x_1)z_1 + (x_2 - x^\star)z_2 \text{ if } z_1, z_2 \gtreqless 0$$

This essentially deals with the monotonicity problem, and we now have to tackle the question of curvature. To do so, we compute the slope of the secant line between x_1 and x_2

$$\Delta = \frac{y_2 - y_1}{x_2 - x_1}$$

Then, if $(z_2 - \Delta)(z_1 - \Delta) \geqslant 0$, this indicates the presence of an inflexion point in the interval $[x_1; x_2]$ such that the interpolant cannot be neither convex nor concave. Conversely, if $|z_2 - \Delta| < |z_1 - \Delta|$ and $x^\star$ satisfies

$$x_1 < x^\star \leqslant \overline{x} \equiv x_1 + \frac{2(x_2 - x_1)(z_2 - \Delta)}{(z_2 - z_1)}$$

then $S(x)$, as described in the latter lemma, is convex (concave) if $z_1 < z_2$ ($z_1 > z_2$). Further, if $z_1 z_2 > 0$ it is also monotone.

If, on the contrary, $|z_2 - \Delta| > |z_1 - \Delta|$ and $x^\star$ satisfies

$$\underline{x} \equiv x_2 + \frac{2(x_2 - x_1)(z_1 - \Delta)}{(z_2 - z_1)} \leqslant x^\star < x_2$$

then $S(x)$, as described in the latter lemma, is convex (concave) if $z_1 < z_2$ ($z_1 > z_2$).

This therefore endow us with a range of values for $x^\star$ that will insure that shape properties will be preserved.

1. Check if lemma 1 is satisfied. If so set $x^\star = x_2$ and set $S(x)$ as in lemma 2. Then STOP else go to 2.

2. Compute $\Delta = y_2 - y_1/x_2 - x_1$

3. if $(z_1-\Delta)(z_2-\Delta)\geqslant 0$ set $x^\star=(x_1+x_2)/2$ and STOP else goto 4.
4. if $|z_1-\Delta|<|z_2-\Delta|$ set $x^\star=(x_1+\overline{x})/2$ and STOP else goto 5.
5. if $|z_1-\Delta|\geqslant|z_2-\Delta|$ set $x^\star=(x_2+\underline{x})/2$ and STOP.

We have then in hand a value for $x^\star$ for $[x_1;x_2]$. We then apply it to each sub–interval to get $x_i^\star\in[x_i;x_{i+1}]$ and then solve the general interpolation problem as explained in lemma 2.

Note here that everything assumes that with have Hermite data in hand — *i.e.* $\{x_i,y_i,z_i : i=0,\ldots,n\}$. However, the knowledge of the slope is usually not the rule and we therefore have to adapt the algorithm to such situations.

3.7.2 Unknown slope: back to Lagrange interpolation

Assume now that we do not have any data for the slope of the function, that is we are only endowed with Lagrange data $\{x_i,y_i : i=0,\ldots,n\}$. In such a case, we just have to add the needed information — an estimate of the slope of the function — and proceed exactly as in Hermite interpolation. Schumaker proposes the following procedure to get $\{z_i; i=1,\ldots,n\}$. Compute

$$L_i=\left[(x_{i+1}-x_i)^2+(y_{i+1}-y_i)^2\right]^{\frac{1}{2}}$$

and

$$\Delta_i=\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$$

for $i=1,\ldots,n-1$. Then z_i, $i=1,\ldots,n$ can be recovered as

$$z_i=\begin{cases}\dfrac{L_{i-1}\Delta_{i-1}+L_i\Delta_i}{L_{i-1}+L_i} & \text{if } \Delta_{i-1}\Delta_i>0\\ 0 & \text{if } \Delta_{i-1}\Delta_i\leqslant 0\end{cases}\qquad i=2,\ldots,n-1$$

and

$$z_1=-\frac{3\Delta_1-z_2}{2}\text{ and } z_n=\frac{3\Delta_{n-1}-s_{n-1}}{2}$$

Then, we just apply exactly the same procedure as described in the previous section.

Up to now, all methods we have been studying are uni–dimensional whereas most of the model we deal with in economics involve more than 1 variable. We therefore need to extend the analysis to higher dimensional problems.

3.8 Multidimensional approximations

Computing a multidimensional approximation to a function may be quite cumbersome and even impossible in some cases. To understand the problem, let us restate an example provided by Judd [1998]. Consider we have data points $\{P_1, P_2, P_3, P_4\} = \{(1,0), (-1,0), (0,1), (0,-1)\}$ in $\mathbb{R}^2$ and the corresponding data $z_i = F(P_i)$, $i = 1, \ldots, 4$. Assume now that we want to construct the approximation of function F using a linear combination of $\{1, x, y, xy\}$ defined as

$$G(x,y) = a + bx + cy + dxy$$

such that $G(x_i, y_i) = z_i$. Finding a, b, c, d amounts to solve the linear system

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}$$

which is not feasible as the matrix is not full rank.

This example reveals two potential problems:

1. Approximation in higher dimensional systems involves cross–product and therefore poses the problem of the selection of polynomial basis to be used for approximation,

2. More important is the selection of the grid of nodes used to evaluate the function to compute the approximation.

We now investigate these issues, by first considering the simplest way to attack the question — namely considering *tensor product bases* — and then moving to a second way of dealing with this problem — considering *complete polynomials*. In each case, we explain how Chebychev approximations can be obtained.

3.8.1 Tensor product bases

The idea here is to use the tensor product of univariate functions to form a basis of multivariate functions. In order to better understand this point, let us consider that we want to approximate a function $F : \mathbb{R}^2 \longrightarrow \mathbb{R}$ using simple univariate monomials up to order 2: $\mathscr{X} = \{1, x, x^2\}$ and $\mathscr{Y} = \{1, y, y^2\}$. The tensor product basis is given by

$$\{1, x, y, xy, x^2, y^2, x^2y, xy^2, x^2y^2\}$$

i.e. all possible 2–terms products of elements belonging to $\mathscr{X}$ and $\mathscr{Y}$.

We are now in position to define the *n–fold tensor product* basis for functions of n variables $\{x_1, \ldots, x_i, \ldots, x_n\}$.

Definition 10 *Given a basis for n functions of the single variable x_i: $\mathscr{P}_i = \{p_i^k(x_i)\}_{k=0}^{\kappa_i}$ then the tensor product basis is given by*

$$\mathscr{B} = \left\{ \prod_{k_1=0}^{\kappa_1} \ldots \prod_{k_n=0}^{\kappa_n} p_1^{k_1}(x_1) \ldots p_n^{k_n}(x_n) \right\}$$

An important problem with this type of tensor product basis is their size. For example, considering a m–dimensional space with polynomials of order n, we already get $(n+1)^m$ terms! This exponential growth in the number of terms makes it particularly costly to use this type of basis, as soon as the number of terms or the number of nodes is high. Nevertheless, it will often be satisfactory or sufficient for low enough polynomials (in practice n=2!) Therefore, one often rely on less computationally costly basis.

3.8.2 Complete polynomials

As aforementioned, tensor product bases grow exponentially as the dimension of the problem increases, complete polynomials have the great advantage of growing only polynomially as the dimension increases. From an intuitive point of view, complete polynomials bases take products of order lower than a priori given κ into account, ignoring higher terms of higher degrees.

Definition 11 *For $\kappa \in \mathbb{N}$ given, the complete set of polynomials of total degree κ in n variables is given by*

$$\mathscr{B}^c = \left\{ x_1^{k_1} \times \ldots \times x_n^{k_n} : \; k_1, \ldots, k_n \geqslant 0, \sum_{i=1}^{n} k_i \leqslant \kappa \right\}$$

To see this more clearly, let us consider the example developed in the previous section ($\mathscr{X} = \{1, x, x^2\}$ and $\mathscr{Y} = \{1, y, y^2\}$) and let us assume that $\kappa = 2$. In this case, we end up with a complete polynomials basis of the type

$$\mathscr{B}^c = \left\{1, x, y, x^2, y^2, xy\right\} = \mathscr{B} \backslash \{xy^2, x^2y, x^2y^2\}$$

Note that we have actually already encountered this type of basis, as this is typically what is done by Taylor's theorem for many dimensions

$$\begin{aligned} F(x) \quad & \simeq \quad F(x^\star) + \sum_{i=1}^{n} \frac{\partial F}{\partial x_i}(x^\star)(x_i - x_i^\star) \\ & \quad \vdots \\ & \quad + \frac{1}{k!} \sum_{i_1=1}^{n} \ldots \sum_{i_k=1}^{n} \frac{\partial F}{\partial x_{i_1} \ldots \partial x_{i_1}}(x^\star)(x_{i_1} - x_{i_1}^\star) \ldots (x_{i_k} - x_{i_k}^\star) \end{aligned}$$

For instance, considering the Taylor expansion to the 2–dimensional function $F(x, y)$ around $(x^\star, y^\star)$ we get

$$\begin{aligned} F(x, y) \quad \simeq \quad & F(x^\star, y^\star) + F_x(x^\star, y^\star)(x - x^\star) + F_y(x^\star, y^\star)(y - y^\star) \\ & + \frac{1}{2} \Bigg(F_{xx}(x^\star, y^\star)(x - x^\star)^2 + 2F_{xy}(x^\star, y^\star)(x - x^\star)(y - y^\star) \\ & + F_{yy}(x^\star, y^\star)(y - y^\star)^2 \Bigg) \end{aligned}$$

which rewrites

$$F(x, y) = \alpha_0 + \alpha_1 x + \alpha_2 y + \alpha_3 x^2 + \alpha_4 y^2 + \alpha_5 xy$$

such that the implicit polynomial basis is the complete polynomials basis of order 2 with 2 variables.

The key difference between tensor product bases and complete polynomials bases lies essentially in the rate at which the size of the basis increases. As aforementioned, tensor product bases grow exponentially while complete polynomials bases only grow polynomially. This reduces the computational cost of approximation. But *what do we loose using complete polynomials rather than tensor product bases?* From a theoretical point of view, Taylor's theorem gives us the answer: Nothing! Indeed, Taylor's theorem indicates that the element in $\mathscr{B}^c$ delivers a approximation in the neighborhood of $x^\star$ that exhibits an asymptotic degree of convergence equal to k. The n–fold tensor product, $\mathscr{B}$, can deliver only a k^{th} degree of convergence as it does not contains all terms of degree $k+1$. In other words, complete polynomials and tensor product bases deliver the same degree of asymptotic convergence and therefore complete polynomials based approximation yields an as good level of accuracy as tensor product based approximations.

Once we have chosen a basis, we can proceed to approximation. For example, we may use Chebychev approximation in higher dimensional problems. Judd [1998] reports the algorithm for this problem. As we will see, it takes advantage of a very nice feature of orthogonal polynomials: they inherit their orthogonality property even if we extend them to higher dimensions. Let us then assume we want to compute the chebychev approximation of a 2–dimensional function $F(x,y)$ over the interval $[a_x;b_x]\times[a_y;b_y]$ and let us assume — to keep things simple for a while — that we use a tensor product basis. Then the algorithm is as follows

1. Choose a polynomial order for x (n_x) and y (n_y)
2. Compute $m_x \geqslant n_x+1$ and $m_y \geqslant n_y+1$ Chebychev interpolation nodes on $[-1;1]$
$$z_k^x=\cos\left(\frac{2k-1}{2m_x}\pi\right),\ k=1,\ldots,m_x$$
and
$$z_k^y=\cos\left(\frac{2k-1}{2m_y}\pi\right),\ k=1,\ldots,m_y$$

3. Adjust the nodes to fit in both interval

$$x_k = a_x + (1+z_k^x)\left(\frac{b_x-a_x}{2}\right),\ k=1,\ldots,m_x$$

and

$$y_k = a_y + (1+z_k^y)\left(\frac{b_y-a_y}{2}\right),\ k=1,\ldots,m_y$$

4. Evaluate the function F at each node to form

$$\Omega \equiv \{\omega_{k\ell} = F(x_k,y_\ell) : k=1,\ldots,m_x; \ell=1,\ldots,m_y\}$$

5. Compute the $(n_x+1)\times(n_y+1)$ Chebychev coefficients α_{ij}, $i=0,\ldots,n_x$, $j=0,\ldots,n_y$ as

$$\alpha_{ij} = \frac{\displaystyle\sum_{k=1}^{m_x}\sum_{\ell=1}^{m_y}\omega_{k\ell}T_i^x\left(z_k^x\right)T_j^y\left(z_\ell^y\right)}{\left(\displaystyle\sum_{k=1}^{m_x}T_i^x\left(z_k^x\right)^2\right)\left(\displaystyle\sum_{\ell=1}^{m_y}T_j^y\left(z_\ell^y\right)^2\right)}$$

which may be simply obtained in this case as

$$\alpha = \frac{T^x(z^x)'\Omega T^y(z^y)}{\|T^x(z^x)\|^2\times\|T^y(z^y)\|^2}$$

6. Compute the approximation as

$$G(x,y) = \sum_{i=0}^{n_x}\sum_{j=0}^{n_y}\alpha_{ij}T_i^x\left(2\frac{x-a_x}{b_x-a_x}-1\right)T_j^y\left(2\frac{y-a_y}{b_y-a_y}-1\right)$$

which may also be obtained as

$$G(x,y) = T^x\left(2\frac{x-a_x}{b_x-a_x}-1\right)\alpha\,T^y\left(2\frac{y-a_y}{b_y-a_y}-1\right)'$$

As an illustration of the algorithm we compute the approximation of the CES function

$$F(x,y) = [x^\rho + y^\rho]^{\frac{1}{\rho}}$$

on the $[0.01;2]\times[0.01;2]$ interval for $\rho=0.75$. We used 5–th order polynomials for both x and y and 20 nodes for both x and y, such that there are 400 possible interpolation nodes. Applying the algorithm we just described, we get the matrix of coefficients reported in table 3.7. As can be seen from the table, most of the coefficients are close to zero as soon as they involve the cross–product of higher order terms, such that using a complete polynomial basis would yield the same efficiency at a lower computational cost. Figure 3.10 reports the graph of the residuals for the approximation.

Table 3.7: Matrix of Chebychev coefficients (tensor product basis)

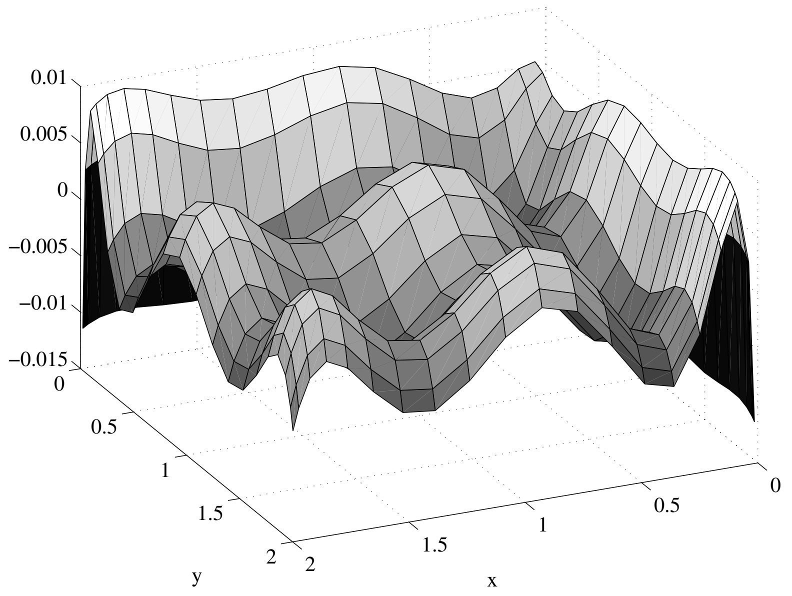
$k_x \setminus k_y$	0	1	2	3	4	5
0	2.4251	1.2744	-0.0582	0.0217	-0.0104	0.0057
1	1.2744	0.2030	-0.0366	0.0124	-0.0055	0.0029
2	-0.0582	-0.0366	0.0094	-0.0037	0.0018	-0.0009
3	0.0217	0.0124	-0.0037	0.0016	-0.0008	0.0005
4	-0.0104	-0.0055	0.0018	-0.0008	0.0004	-0.0003
5	0.0057	0.0029	-0.0009	0.0005	-0.0003	0.0002

Matlab Code: Chebychev Coefficients in $\mathbb{R}^2$ (Tensor Product Basis)

```
rho = 0.75;
mx  = 20;
my  = 20;
nx  = 5;
ny  = 5;
ax  = 0.01;
bx  = 2;
ay  = 0.01;
by  = 2;
%
% Step 1
%
rx  = cos((2*[1:mx]'-1)*pi/(2*mx));
ry  = cos((2*[1:my]'-1)*pi/(2*my));
%
% Step 2
%
x   = (rx+1)*(bx-ax)/2+ax;
y   = (ry+1)*(by-ay)/2+ay;
```

```
%
% Step 3
%
Y   = zeros(mx,my);
for ix=1:mx;
   for iy=1:my;
      Y(ix,iy) = (x(ix)^rho+y(iy)^rho)^(1/rho);
   end
end
%
% Step 4
%
Xx  = [ones(mx,1) rx];
for i=3:nx+1;
  Xx= [Xx 2*rx.*Xx(:,i-1)-Xx(:,i-2)];
end Xy  = [ones(my,1) ry];
for i=3:ny+1;
  Xy= [Xy 2*ry.*Xy(:,i-1)-Xy(:,i-2)];
end
T2x = diag(Xx'*Xx);
T2y = diag(Xy'*Xy);
a   = (Xx'*Y*Xy)./(T2x*T2y');
```

Figure 3.10: Residuals: Tensor product basis



If we now want to perform the same approximation using a complete polynomials basis, we just have to modify the algorithm to take into account the fact that when iterating on i and j we want to impose $i+j \leqslant \kappa$. Let us compute is for $\kappa=5$. This implies that the basis will consists of

$$\begin{aligned}
&1, T_1^x(.), T_1^y(.), T_2^x(.), T_2^y(.), T_3^x(.), T_3^y(.), T_4^x(.), T_4^y(.), T_5^x(.), T_5^y(.),\\
&T_1^x(.)T_1^y(.), T_1^x(.)T_2^y(.), T_1^x(.)T_3^y(.), T_1^x(.)T_4^y(.),\\
&T_2^x(.)T_1^y(.), T_2^x(.)T_2^y(.), T_2^x(.)T_3^y(.),\\
&T_3^x(.)T_1^y(.), T_3^x(.)T_2^y(.),\\
&T_4^x(.)T_1^y(.)
\end{aligned}$$

Table 3.8: Matrix of Chebychev coefficients (Complete polynomials basis)

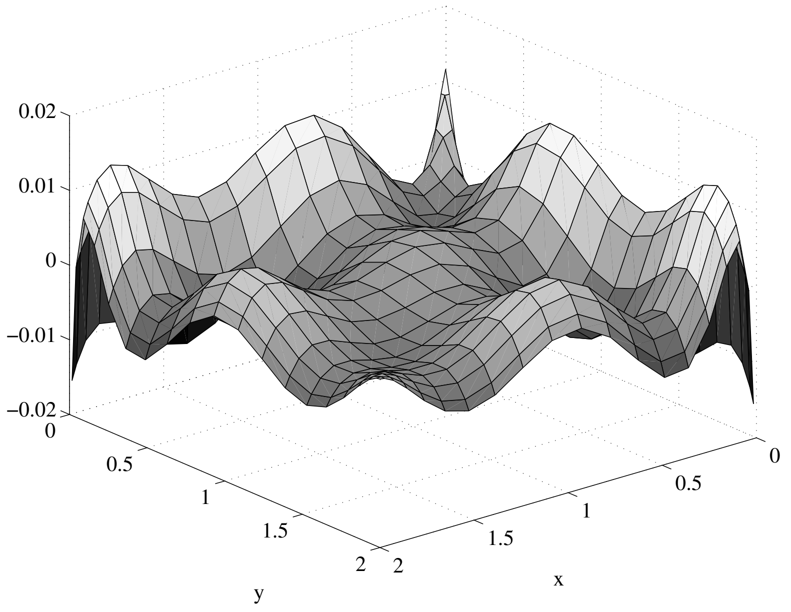
$k_x \setminus k_y$	0	1	2	3	4	5
0	2.4251	1.2744	-0.0582	0.0217	-0.0104	0.0057
1	1.2744	0.2030	-0.0366	0.0124	-0.0055	–
2	-0.0582	-0.0366	0.0094	-0.0037	–	–
3	0.0217	0.0124	-0.0037	–	–	–
4	-0.0104	-0.0055	–	–	–	–
5	0.0057	–	–	–	–	–

A first thing to note is that the coefficients that remain are the same as the one we got in the tensor product basis. This should not be any surprise as what we just find is just the expression of the Chebychev economization we already encountered in the uni–dimensional case and which is just the direct consequence of the orthogonality condition of chebychev polynomials. Figure 3.11 report the residuals from the approximation using the complete basis. As can be seen from the figure, this "constrained" approximation yields quantitatively similar results compared to the tensor product basis, therefore achieving almost the same accuracy while being less costly from a computational point of view. In the MATLAB code section, we just report the lines in `step 4` that are affected by the adoption of the complete polynomials basis.

Matlab Code: Complete Polynomials Specificities

```
a=zeros(nx+1,ny+1);
for ix=1:nx+1;
   iy = 1;
   while ix+iy-2<=kappa
         a(ix,iy)=(Xx(:,ix)'*Y*Xy(:,iy))./(T2x(ix)*T2y(iy));
         iy=iy+1;
    end
end
```

Figure 3.11: Residuals: Complete polynomials basis



3.9 Finite element approximations

Finite element are extremely popular among engineers (especially in aeronautics). This approach considers elements that are zero over most of the domain of approximation. Although they are extremely powerful in the case of 2–dimensional problems, they are more difficult to implement in higher dimensions. We therefore focus on the bi–dimensional case.

3.9.1 Bilinear approximations

A bilinear interpolation proposes to interpolate data linearly in both coordinate directions. Assume that we have the values of a function $F(x,y)$ at the four points

$$\begin{array}{cc} P_1=(-1,-1) & P_2=(-1,1) \\ P_3=(1,-1) & P_4=(1,1) \end{array}$$

A cardinal interpolation basis on $[-1;1]\times[-1;1]$ is provided by the set of functions

$$\begin{array}{cc} b_1(x,y)=\frac{1}{4}(1-x)(1-y) & b_2(x,y)=\frac{1}{4}(1-x)(1+y) \\ b_3(x,y)=\frac{1}{4}(1+x)(1-y) & b_4(x,y)=\frac{1}{4}(1+x)(1+y) \end{array}$$

All functions b_i are zero on all P_j, $i\neq j$, but on the point to which is associated the same index $(i=j)$. Therefore, an approximation of $F(x,y)$ on $[-1;1]\times[-1;1]$ is given by

$$F(x,y)\simeq F(-1,-1)b_1(x,y)+F(-1,1)b_2(x,y)+F(1,-1)b_3(x,y)+F(1,1)b_4(x,y)$$

If we have data on $[a_x;b_x]\times[a_y;b_y]$, we use the linear transformation we have already encountered a great number of times

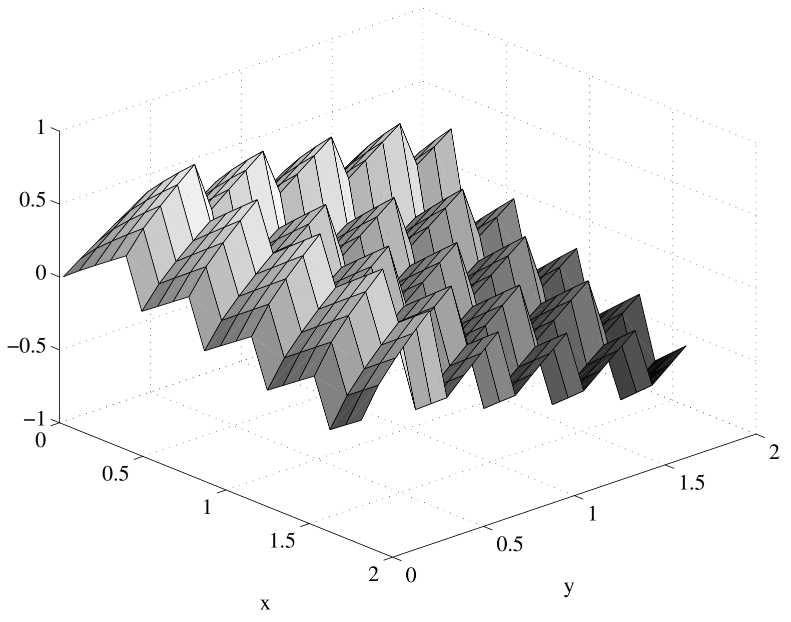
$$\left(2\frac{x-a_x}{b_x-a_x}-1,2\frac{y-a_y}{b_y-a_y}-1\right)$$

Then, if we have Lagrange data of the type $\{x_i,y_i,z_i\,:\,i=1,\ldots,n\}$, we proceed as follows

1. Construct a grid of nodes for x and y;
2. Construct the interpolant over each square applying the previous scheme;
3. Piece all interpolant together.

Note that an important issue of piecewise interpolation is related to the continuity of the approximation: individual pieces must meet continuously at common edges. In bilinear interpolation, this is insured by the fact that two interpolants overlap only at the edges of rectangles on which the approximation is a linear interpolant of 2 common end points. This would not be insured if we were to construct bi–quadratic or bi–cubic interpolations for example. Note that this type of interpolation scheme is typically what is done when a computer draw a 3d graph of a function. In figure 3.12, we plot the residuals of the bilinear approximation of the CES function we approximated in the previous section, with 5 uniform intervals (6 nodes such that $x = \{0.010, 0.408, 0.806, 1.204, 1.602, 2.000\}$ and $y = \{0.010, 0.408, 0.806, 1.204, 1.602, 2.000\}$). Like in the spline approximation procedure, the most difficult step once we have obtained an approximation is to determine the square the point for which we want an approximation belongs to. We therefore face exactly the same type of problems.

Figure 3.12: Residuals of the bilinear approximation



3.9.2 Simplicial 2D linear interpolation

This method essentially amounts to consider triangles rather than rectangles as an approximation basis. The idea is then to build triangles in the $x--y$ plane. To do so, and assuming that the lagrange data have already been transformed using the linear map described earlier, we set 3 points

$$P_1 = (0,0),\ P_2 = (0,1),\ P_3 = (1,0)$$

to which we associate 3 functions

$$b_1(x,y) = 1 - x - y,\ b_2(x,y) = y,\ b_3(x,y) = x$$

which are such that all functions b_i are zero on all P_j, $i \neq j$, but on the point to which is associated the same index ($i = j$). b_1 b_2 and b_3 are the cardinal functions on P_1, P_2, P_3. Let us now add the point $P_4 = (1,1)$, then we have the following cardinal functions for P_2, P_3, P_4:

$$b_4(x,y) = 1 - x,\ b_5(x,y) = 1 - y,\ b_6(x,y) = x + y - 1$$

Therefore, on the square P_1, P_2, P_3, P_4 the interpolant for F is given by

$$G(x,y) = \begin{cases} F(0,0)(1-x-y) + F(0,1)y + F(1,0)x & \text{if } x+y \leqslant 1 \\ \\ F(0,1)(1-x) + F(1,0)(1-y) + F(1,1)(x+y-1) & \text{if } x+y \geqslant 1 \end{cases}$$

It should be clear to you that if these methods are pretty easy to implement in dimension 3, it becomes quite cumber some in higher dimensions, and noth that much is proposed in the literature, are most of these methods were designed by engineers and physicists that essentially have to deal with 2, 3 at most 4 dimensional problems. We will see that this will be a limitation in a number of economic applications.

Bibliography

Hornik, K., M. Stinchcombe, and H. White, Multi–Layer Feedforward Networks are Universal Approximators, *Neural Networks*, 1989, *2*, 359–366.

Judd, K.L., *Numerical methods in economics*, Cambridge, Massachussets: MIT Press, 1998.

Schumaker, L.L., On Shape–preseving Quadratic Spline Interpolation, *SIAM Journal of Numerical Analysis*, 1983, *20*, 854–864.

Index

Contents

List of Figures

List of Tables