Bar-Ilan University
Estimation of DP Models
March, 2017

Moshe Buchinsky
Department of Economics
UCLA

Lecture Note 4

Appoximation and Interpolation Methods

Part I

Introduction

In this lecture we deal with a common problem in economics, namely approximation of functions.

Such a problem occurs when it is too costly either in terms of time or complexity to compute the true function or when the function is unknown

Usually the only thing required is to be able to compute the function at few points and formulate a guess for all other values.

Th e choice of 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.

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. We usually rely on interpolation, which requires to know the function at some points.

3. *Regressions*, an intermediate situation between the two preceding cases, as it usually relies on m moments to find n parameters of the approximating function.

Local approximations

The problem of the local approximation of a function

$$f : \mathbb{R} \to \mathbb{R}.$$

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.

Two nethods of interest:

- Taylor series expansion; and
- Padé approximation

Taylor series expansion

The most well-known and natural approximation

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

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

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

Now we need 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 k times differentiable to get a kth order approximation.

Then the error is at most of order $\mathcal{O}\left(\|x - x^*\|^{k+1}\right)$

A Reminder: A function $f : \mathbb{R}^n \to \mathbb{R}^k$ is $\mathcal{O}\left(x^l\right)$ if

$$\lim_{x \to 0} \frac{\|f(x)\|}{\|x\|^l} < \infty.$$

We may look at Taylor series expansion as an approximate the function by an infinite series.

For instance in the one dimensional case

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

where $\alpha_k = \frac{1}{k!} \frac{\partial F}{\partial x_i}(x^*)$.

As $n \to \infty$ the equation can be understood as a power series expansion of the F in the neighborhood of x^*.

A computer delivers $\mathsf{exp}(x)$ in similar way, since

$$\mathsf{exp}\left(x\right) \equiv \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

This representation gives rise to a very important theorem...

but first some preliminary definitions.

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

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

Therefore r 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 $|x| > r$.

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

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

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

Example, consider the tangent function $\tan(x)$.

It can be written as 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 $x^* = \pi/2$, for which $\cos(x^*) = 0$.

Theorem 2 *Let F be an analytic function in $x \in \mathbb{C}$. If F or any derivative of F exhibits a singularity at $x^0 \in \mathbb{C}$, then the radius of convergence in the complex plane of the Taylor series expansion of F in the neighborhood of x^**

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

is bounded from above by $\left\| x^* - x^0 \right\|$.

This is extremely important as it gives us a guideline to use Taylor series expansions.

It actually tells us that the series at x^* cannot deliver a reliable approximation for F at any point farther away from x^* than any singular point of F.

An example: Consider an approximation for

$$F(x) = \log(1 - x), \quad \text{where } x \in (-\infty, 1),$$

in a neighborhood of $x^* = 0$.

Note that $x^0 = 1$ is a singular value for $F(x)$.

The Taylor series expansion of $F(x)$ in the neighborhood of $x^* = 0$ is

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

What the theorem tells us is that this approximation may be used for values of x such that $\|x - x^*\|$ is below $\left\|x^* - x^0\right\| = \|0 - 1\| = 1$. That is, only for x such that $-1 < x < 1$.

In other words, the radius of convergence of the Taylor approximation to this function is $r = 1$.

Table 1 reports an example for the "true" value of $\log(1-x)$and its approximate value using 100 terms in the summation

Note that as soon as $\left\|x^* - x^0\right\|$ approaches the radius, the accuracy falls sharply

Table 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

The usefulness of the approach: This approach to approximation is particularly useful and quite widespread in economic dynamics

For example: The optimal growth model.

When preferences are logarithmic and technology is Cobb-Douglas, its dynamic is characterized by the following equations

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

$$\frac{1}{c_t} = \beta \frac{1}{c_{t+1}} \left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta \right) \tag{2}$$

It is customary. to linearize of log-linearize such an economy around the steady state

Assume that the steady state given by $c_{t+1} = c_t = c^*$ and $k_{t+1} = k_t = k^*$ for all t.

Linearization:

Denote by $\widehat{k}_t$ the deviation of k_t from its steady state level in period t, i.e., $\widehat{k}_t = k_t - k^*$ for all t.

Likewise, define $\widehat{c}_t = c_t - c^*$.

The first step of linearization is to re-express the system in terms of functions:

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

From which we build the Taylor expansion:

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

We have to compute the derivatives of each sub-function, and realize that in steady state $F\left(k^*, c^*, k^*, c^*\right) = 0$.

This yields the following system

$$\begin{pmatrix} \widehat{k}_{t+1} - \alpha k_t^{*\alpha-1} + \widehat{c}_t - (1-\delta)\,\widehat{k}_t \\ -\frac{1}{c^{*2}}\widehat{c}_t + \beta\frac{1}{c^{*2}}\left(\alpha k_t^{*\alpha-1} + 1 - \delta\right)\widehat{c}_{t+1} - \beta\frac{1}{c^*}\alpha\,(\alpha-1)\,k_t^{*\alpha-2}\widehat{k}_{t+1} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

which simplifies to

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

We then have to solve the implied linear dynamic system, but this is another story....

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, namely, they can be interpreted as elasticities.

Consider the one-dimensional function $f(x)$ and let's assume that we want to take a log-linear approximation of f around x^*

This would amount to have, as deviation, a log-deviation rather than a simple deviation:

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

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

$$f(x) = f(\mathsf{exp}(\mathsf{log}(x))),$$

which leads to the following first order Taylor expansion

$$\begin{aligned} f(x) &\simeq f(x^*) + f'(\mathsf{exp}(\mathsf{log}(x^*)))\,\mathsf{exp}(\mathsf{log}(x^*))\,\widehat{x} \\ &= f(x^*) + f'(x^*)\,x^*\widehat{x}. \end{aligned}$$

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

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

Regressions as approximation

This type of approximation is particularly common in economics as it just corresponds to *ordinary least square* (OLS).

The problem amounts to approximating a function F by another function G of exogenous variables.

We have a set of observable endogenous variables y_i, $i = 1, ..., N$, which we are willing to explain in terms of the set of exogenous variables $\mathfrak{X}_i = \left\{x_i^1, ..., x_i^k\right\}$, $i = 1, ..., N$.

This problem amounts to find a set of parameters θ that solves the problem

$$\min_{\theta \in \mathbb{R}^p} \sum_{i=1}^{N} \left(y_i - G\left(\mathfrak{X}_i; \theta\right)\right)^2 . \tag{3}$$

Now, θ is chosen so that on average $G\left(\mathfrak{X}_i; \theta\right)$ is close enough to y_i, such that G delivers a "good" approximation for the true function F

Number of data points: In numerical analysis we may be exactly identified, that is, the number of data points N, can be equal to the number of parameters p

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.

Example: Assume we have a sample made of 2 data points for a function that we want to approximate using a linear function.

The linear function is defined by an intercept, α, and the slope β.

In such a case, (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

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

or

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

This system then just amounts to find the null space of the matrix A

This 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 over-identified this is not the case anymore.

Selection of data points:

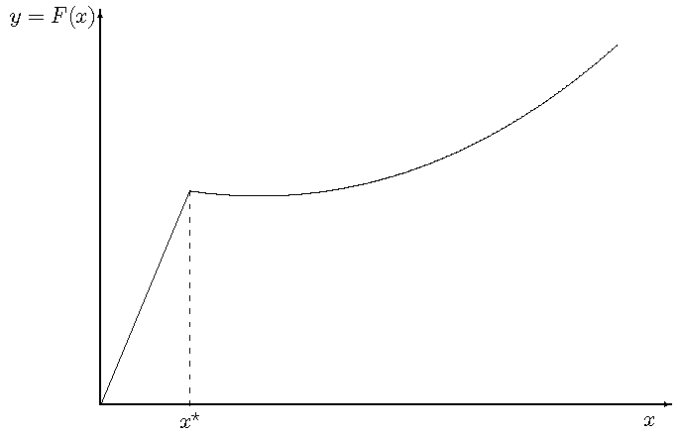
This is actually a major difference between econometrics and numerical approximations.

In the latter case we 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.

Consider the following function

Figure 1: Selection of points



It may be beneficial to concentrate a lot of points around the kink in x^*

Functional forms:

One key issue in the selection of the approximating function is a functional form.

One simple choice is $\mathfrak{X}_i = \left\{1, x_i, x_i^2, ..., x_i^p\right\}$.

However, this often turns to be a very bad choice as many problems then turn out to be ill-conditioned with such a choice.

This particularly can create a multicollinearity 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 a smooth moving average of investment decisions

$$k_{t+1} = i_t + (1-\delta)\, k_t \iff k_{t+1} = \sum_{l=0}^{\infty} (1-\delta)^l\, i_{t-l}.$$

Therefore, taking as a basis for exogenous variables powers of the capital stock is a bad idea.

To see that, assume that $\delta = .025$ and i_t is a white noise process with variance 0.1, then simulate a 1000 data points process for the capital stock.

The correlation matrix for k_t^j, $j = 1, ..., 4$, 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

A typical solution for this problem is to rely on orthogonal polynomials rather than monomials. (Will be discussed later.)

A second possibility is to rely on parsimonious approaches that do not require too much information in terms of function specification.

An alternative is to use neural networks.

But first we have to deal with a potential problem that one may face with all the examples.

The true decision rule is unknown, so we do not actually know the function we are dealing with.

However, the main properties of the decision rule are known, e.g., we know that it has to satisfy some conditions imposed by economic theory.

Example: Try to to find an approximate solution for the consumption decision rule in the deterministic optimal growth model.

Economic theory implies that consumption should satisfy the Euler equation

$$c_t^{-\sigma} = \beta c_{t+1}^{-\sigma} \left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta \right), \tag{4}$$

while the capital stock evolves according to

$$k_{t+1} = k_t^{\alpha} - c_t + (1 - \delta) k_t. \tag{5}$$

Assume that consumption may be approximated by

$$\phi\left(k_t, \theta\right) = \exp\left(\theta_0 + \theta_1 \log\left(k_t\right) + \theta_3 \log\left(k_t\right)^2\right)$$

over the interval $\left[\underline{k}, \overline{k}\right]$.

Our problem is then to find the triple $\{\theta_0, \theta_1, \theta_3\}$ that minimizes

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

This amounts to solving 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:

$$k_{t+1} = k_t^{\alpha} - \exp\left(\theta_0 + \theta_1 \log\left(k_t\right) + \theta_3 \log\left(k_t\right)^2\right) + \left(1 - \delta\right) k_t.$$

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 $\left[\underline{k}, \overline{k}\right]$, and an initial vector $\{\theta_0, \theta_1, \theta_3\}$.

2. For each k_i, $i = 1, ..., N$, and given $\{\theta_0, \theta_1, \theta_3\}$, compute

$$\begin{aligned} c_t &= \phi\left(k_t, \theta\right) \quad \text{and} \\ k_{t+1} &= k_t^{\alpha} - \phi\left(k_t, \theta\right) + \left(1 - \delta\right) k_t. \end{aligned}$$

3. Compute

$$\begin{aligned} c_{t+1} &= \phi\left(k_{t+1}, \theta\right) \quad \text{and the quatity} \\ \Re\left(k_t, \theta\right) &= \phi\left(k_t, \theta\right)^{-\sigma} - \beta\phi\left(k_{t+1}, \theta\right)^{-\sigma}\left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta\right) \end{aligned}$$

4. If the quantity

$$\sum_{t=1}^{N} \Re\left(k_t, \theta\right)^2$$

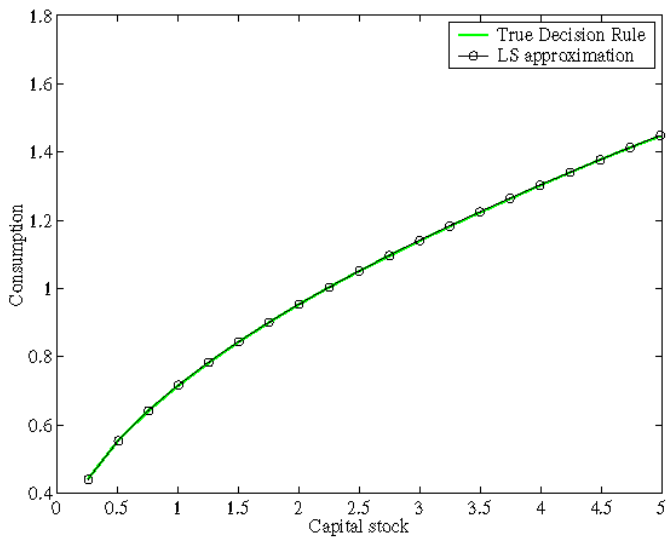
is minimal then stop, else update $\{\theta_0, \theta_1, \theta_3\}$ and go back to step 2.

Example: compute the approximate decision rule for the deterministic optimal growth model with $\alpha = .3$, $\beta = .95$, $\delta = .1$, and $\sigma = 1.5$.

Assume that k_t may deviate up to 90% from its capital stock in steady state i.e., $\underline{k} = .1k^*$ and $\overline{k} = 1.9k^*$

Used 20 data points.

Figure 2: Least-square approximation of consumption



The solution obtained is rather accurate.

We actually have

$$\frac{1}{N}\sum_{i=1}^{N}\left|\frac{c_i^{true}-c_i^{LS}}{c_i^{true}}\right| = 8.7434e^{-4} \quad \text{and} \quad \max_{i=1,\ldots,N}\left|\frac{c_i^{true}-c_i^{LS}}{c_i^{true}}\right| = .00514$$

Interpretation: 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.

Neural Network:

A neural network may be simply viewed as a particular type of functions, flexible enough to fit fairly general functions.

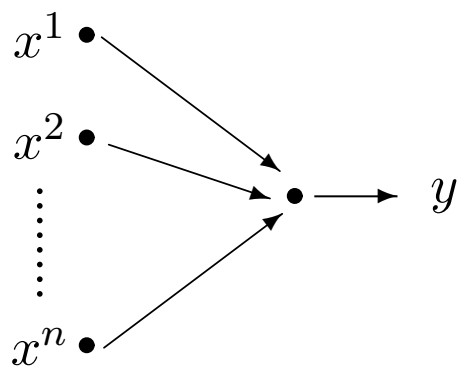
A neural network may be simply understood using the standard metaphor of human brain.

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.

Example: panel (a) of Figure 3, borrowed from Judd [1998], illustrates the *single-layer* neural network

Figure 3: Neural Networks—Panel A



(a)

The functional form of the *single-layer* neural network is given by

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

where x is the vector of inputs

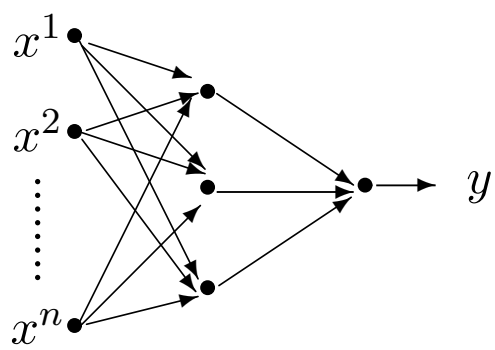
h and g are scalar functions.

A common choice for g is $g(x) = x$.

If we set h to be the identity, then we are back to the standard OLS model with monomials

A second, and more interesting type of neural network is the *hidden-layer feed-forward* neural network depicted in panel (b) of Figure 3

Figure 3: Neural Networks—Panel B



(b)

The associated function form for the *hidden-layer feedforward* neural network is given by

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

In this case, h is called the *hidden-layer activation function* and it serves as a "squasher" function

That is, h is monotonically non-decreasing function that maps $\mathbb{R}$ onto $[0,1]$.

Three very popular functions for h are:

1. The heaviside step function

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

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 $\left\{\gamma_j, \theta_i^j;\ i = 1, ..., n;\ j = 1, ..., m\right\}$.

This can be done by a non-linear least squares, that is, solving

$$\min_{\{\theta,\gamma\}} \sum_{l=1}^{N} \left(y_l - G\left(x_l, \theta, \gamma\right)\right)^2 .$$

A nice feature of neural networks is that they deliver accurate approximation with relatively few parameters.

The flexibility of the approximating functions is what does the job.

Neural network are extremely powerful in that they offer a universal approximation method.

This is established in the following theorem by Hornik, Stinchcombe and White [1989].

Theorem 3 *Let $f : \mathbb{R}^n \to \mathbb{R}$ 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 $1 \leq p \leq \infty$; or (ii) h is a "squashing" function (non-decreasing, with $\lim_{x\to\infty} h(x) = 1$, $\lim_{x\to-\infty} h(x) = 0$). Let*

$$\begin{aligned}\Sigma^n(h) \;=\; \{g : \mathbb{R}^n \to \mathbb{R},\; g(x) = \sum_{j=1}^{n} \theta_j h\left(x' w^j + a_j\right),\; a_j, \theta_j \in \mathbb{R},\\ \text{and } w^j \in \mathbb{R}^n,\; w^j \neq 0,\; m = 1, 2, ...\}\end{aligned}$$

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 $g \in \Sigma^n(h)$ such that

$$\sup_{x\in K} |f(x) - g(x)| \leq \varepsilon \quad \text{and} \quad \int_K |f(x) - g(x)|\,d\mu \leq \varepsilon.$$

This theorem states that for a broad class of functions, neural networks deliver an accurate approximation to any continuous function.

We may use any squashing function of the type described above, or any simple function that satisfies condition (i).

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, consider the 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

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

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

$$\sum_{i=1}^{N} \left(F(x_i) - \widetilde{F}(x_i, \Theta) \right)^2$$

   if this quantity is minimal then stop, else update Θ and go back to 2.

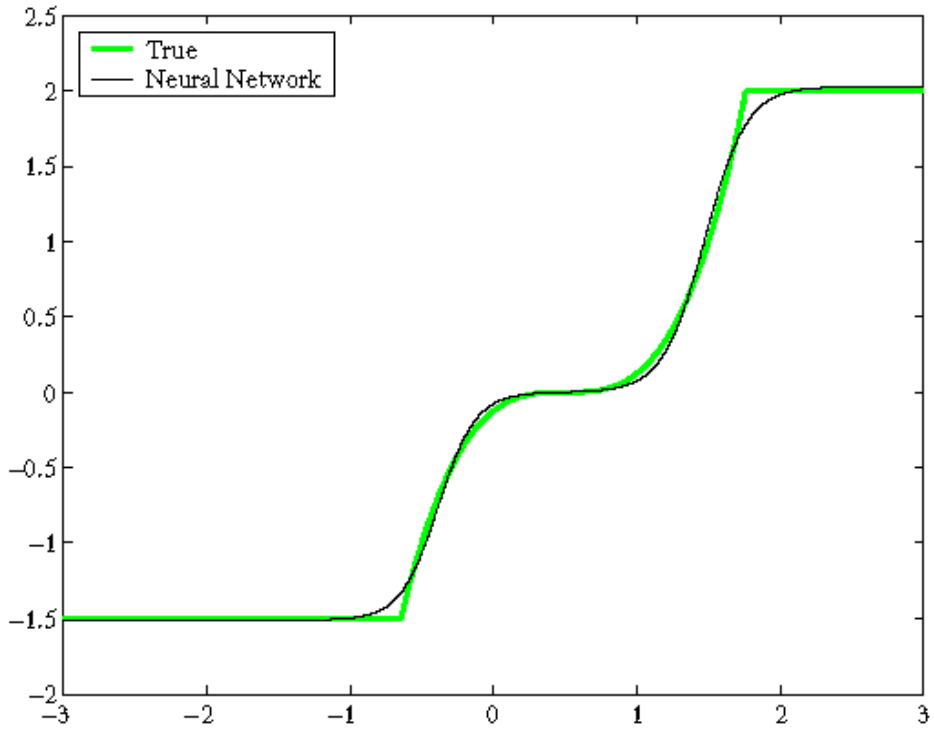
The last step can be performed using a non-linear minimizer.

Figure 4 plots the approximation where $N = 1000$ and the solution vector Θ yields the values reported in

Table 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

Figure 4: Neural Network Approximation



Note from the form of the Θ that the first layer handle positive values for x, while the second layer takes care of the negative part of the function.

Generally, the the function IS approximated quite well by this simple neural network:

$$\begin{aligned}
\mathcal{E}_2 &= \left[\frac{1}{N}\sum_{i=1}^{N}\left(F\left(x_i\right)-\widetilde{F}\left(x_i,\widehat{\Theta}\right)\right)^2\right]^{1/2} = .0469 \quad \text{and} \\
\mathcal{E}_\infty &= \max_i\left|F\left(x_i\right)-\widetilde{F}\left(x_i,\widehat{\Theta}\right)\right| = .220
\end{aligned}$$

All the methods so far 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 more powerful methods which are simpler to implement.

But....

We first need to define some important preliminary concepts which are related to *orthogonal polynomials*.

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.

It solves the multicollinearity problem we encountered in OLS.

It greatly simplifies the computation of the approximation in a number of problems.

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 $\omega(x) = \left(1 - x^2\right)^{-1/2}$ over the interval $[-1, 1]$. Indeed, $\lim_{x \to -1} \omega(x) = \sqrt{2}/2$ and $\omega'(x) > 0$,so that $\omega(x)$ is positive everywhere over the whole interval. Furthermore,

$$\int_{-1}^{1} \left(1 - x^2\right)^{-1/2} dx = \arcsin(x)|_{-1}^{1} = \pi.$$

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*

$$\langle f_1, f_2 \rangle = \int_a^b f_1(x) f_2(x) \omega(x) dx.$$

Example: assume that $f_1(x) = 1$ and $f_2(x) = x$, and $\omega(x) = \left(1 - x^2\right)^{-1/2}$.

Then, the inner product over the interval $[-1, 1]$ is

$$\langle f_1, f_2 \rangle = \int_{-1}^{1} \frac{x}{\left(1 - x^2\right)^{-1/2}} dx = -\sqrt{1 - x^2}\Big|_{-1}^{1} = 0.$$

That is, the inner product of 1 and x wrt $\omega(x)$ on $[-1, 1]$ is identically null.

This, in fact, defines the *orthogonality property*.

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

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

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

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

Common families of orthogonal polynomials (see Judd [1998]) and their recursive formulation are given here:

Table 3: Orthogonal polynomials (definitions)

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

Table 4: Orthogonal polynomials (recursive representation)

Family	0	1	Recursion
Legendre	1	$P_1(x) = x$	$P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x)$
Chebychev	1	$T_1(x) = x$	$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x)$
Laguerre	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	1	$H_1(x) = 2x$	$H_{n+1}(x) = 2x H_n(x) - 2n H_{n-1}(x)$

Least square orthogonal polynomial approximation

Definition 8 *Let $F : [a, b] \to \mathbb{R}$ 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 $\omega(x)$ is the degree n polynomial that solves*

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

The weight $\omega(x)$ may be given the same interpretation as in GMM estimation.

Setting $\omega(x) = 1$, which amounts to put the same weight on any x then corresponds to a simple OLS approximation.

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

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

The first order conditions wrt to c_i are given by

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

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

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.

We will focus on Chebychev approximation.

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

Examples:

- *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 just 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

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

Interpolation methods

Up to now, we have seen that there exist methods to compute the value of a function at some particular points.

In many cases we might also be interested in getting the function at some other points.

This is the problem of *interpolation*.

Linear interpolation

Assume you have a collection of data points $\mathcal{C} = \{(x_i, y_i) \,|i = 1, ..., n\}$.

Then for any $x \in [x_{i-1}, x_i]$, we can compute y as

$$y = \underbrace{\frac{y_i - y_{i-1}}{x_i - x_{i-1}}}_{\text{slope}} x + \underbrace{\frac{x_i y_{i-1} - x_{i-1} y_i}{x_i - x_{i-1}}}_{\text{Intercept}}.$$

This method is very useful in many of applications

But, it can be inefficient for different:

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 perform badly for non-linear functions.

Lagrange interpolation

This type of approximation consider a collection of data

$$\mathcal{C} = \{(x_i, y_i) \,|i = 1, ..., n\},$$

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 $P_i\left(x_j\right) = 0$.

The interpolation is then given by

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

An obvious problem is that we if the number of point is high enough, this type of interpolation is totally intractable.

Indeed, just to compute a single $P_i(x)$ this already requires $2(n-1)$ subtractions and n multiplications.

Then this has to be constructed for all n data points to compute all needed $P_i(x)$.

Then to compute $P(x)$ we need n additions and n multiplications.

Overall, this interpolation requires $3n^2$ operations!

Instead, one may actually attempt to compute directly:

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

which may be obtained by solving the linear system

$$\begin{cases} y_1 = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 + \cdots + \alpha_{n-1} x_1^{n-1} \\ y_2 = \alpha_0 + \alpha_1 x_2 + \alpha_2 x_2^2 + \cdots + \alpha_{n-1} x_2^{n-1} \\ \vdots \\ y_n = \alpha_0 + \alpha_1 x_n + \alpha_2 x_n^2 + \cdots + \alpha_{n-1} x_n^{n-1} \end{cases},$$

or

$$A\alpha = y,$$

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

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

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

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

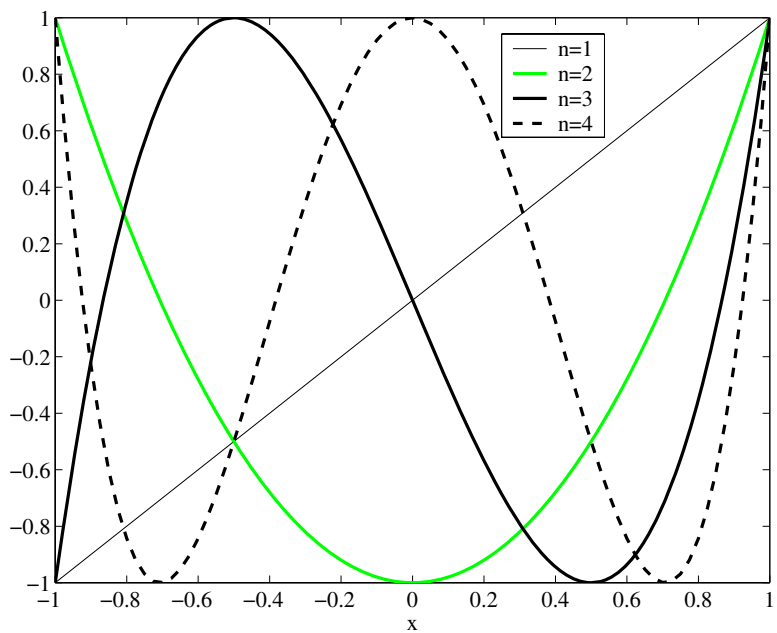
The method may be quite demanding from a computational point of view, as we have to invert an $n \times n$ matrix.

There then exist much more efficient methods, and in particular the so-called *Chebychev approximation* that works very well for smooth functions.

Chebychev approximation

Chebychev approximation uses Chebychev polynomials as a basis for the polynomials:

Figure 5: Chebychev polynomials



These polynomials are described by the recursion

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

This admits as solution

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

These polynomials form an orthogonal basis with respect to the weighting function $\omega(x) = \left(1 - x^2\right)^{-1/2}$ over the interval [-1, 1].

Nevertheless, this interval may be generalized to $[a, b]$, by transforming the data using the formula

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

Beyond the standard orthogonality property, Chebychev polynomials exhibit a discrete orthogonality property:

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

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

The following theorem establishes the usefulness of Chebychev approximation.

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

$$c_i = \frac{2}{\pi} \int_{-1}^{1} \frac{F(x) T_i(x)}{\sqrt{1 - x^2}} dx, \quad \text{for } i = 1, ..., n,$$

and

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

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

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

That is, the approximation $g_n(x)$ will arbitrarily close to $F(x)$ as the degree of approximation n increases to ∞.

In effect, since $\varepsilon \frac{\log(n)}{n^k} \to 0$ as $n \to \infty$, we have that $g_n(x)$ converges uniformly to $F(x)$.

Furthermore, 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

$$|c_i| \leq \frac{c}{i^k} \quad \text{for } i \geq 1.$$

That is, the theorem states that the smoother the function to be approximated is (i.e., the greater k is), the faster is the pace at which the coefficients will drop off.

Consequently we can achieve a high enough accuracy using less coefficients.

So far 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 nth degree Chebychev polynomial.

We have m data points to use in computing the approximation.

Using $m > n$ data points, we can compute the $(n-1)$th order Chebychev approximation relying on the Chebychev regression algorithm described below.

When $m = n$, the algorithm reduces to the so-called Chebychev interpolation formula.

We have a function $F : [a, b] \to \mathbb{R}$. The objective is to construct a degree $n \leq m$ polynomial approximation for F on $[a, b]$:

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

The Algorithm:

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

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

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

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

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

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

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

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

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 can be interpreted in terms of an OLS problem. That is, because of the orthogonality property of the Chebychev polynomials the coefficients α_i are given by

$$\alpha_i = \frac{cov\left(y, T_i\left(x\right)\right)}{var\left(T_i\left(x\right)\right)} \quad \text{for } i = 0, ..., n.$$

Or, in matrix notations

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

where

$$X = \begin{pmatrix} T_0\left(x_1\right) & T_1\left(x_1\right) & \dots & T_n\left(x_1\right) \\ T_0\left(x_2\right) & T_1\left(x_2\right) & \dots & T_n\left(x2\right) \\ \vdots & \vdots & \ddots & \vdots \\ T_0\left(xm\right) & T_1\left(x_m\right) & \dots & T_n\left(xm\right) \end{pmatrix} \quad \text{and} \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

We now report two examples implementing the above algorithm.

The first one deals with the smooth function

$$F(x) = x^{\theta}.$$

The second evaluates the accuracy for the non-smooth function:

$$F(x) = \min\left(\max\left(-1.5, (x - 1/2)^3\right), 2\right).$$

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.

Table 5 reports the coefficients:

Table 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

The table shows that adding terms in the approximation does not alter the coefficients of lower degree.

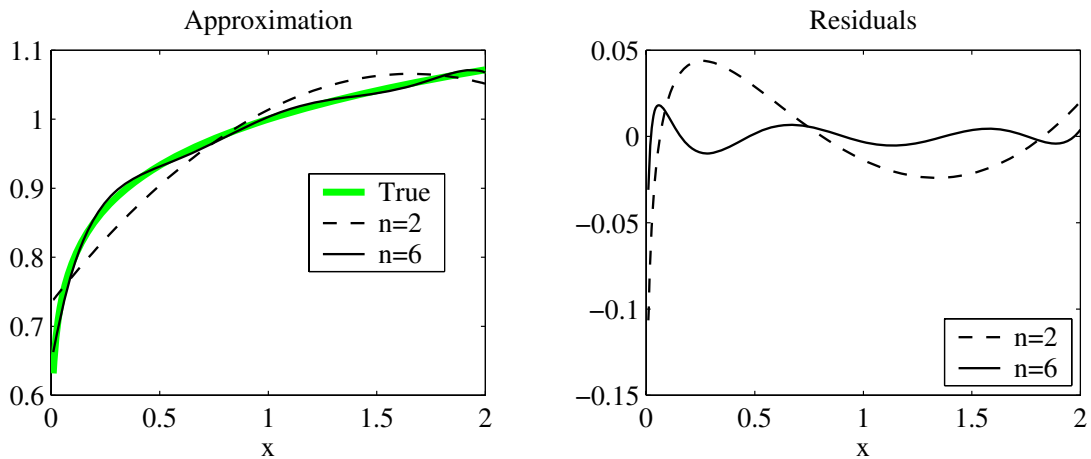
This merely reflects the orthogonality properties of the Chebychev polynomials.

This is of great importance: Once we obtain a high order approximation, obtaining lower orders is particularly simple.

This is the *economization principle*.

Figure 6 reports the true function and the corresponding approximation.

Figure 6: Smooth function: $F(x) = x^{0.1}$



A "good approximation" for the function is obtained at rather low degrees.

Indeed, the difference between the function and its approximation at order 6 is already excellent.

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
```

Non-smooth function:

In the case of the non-smooth function we consider

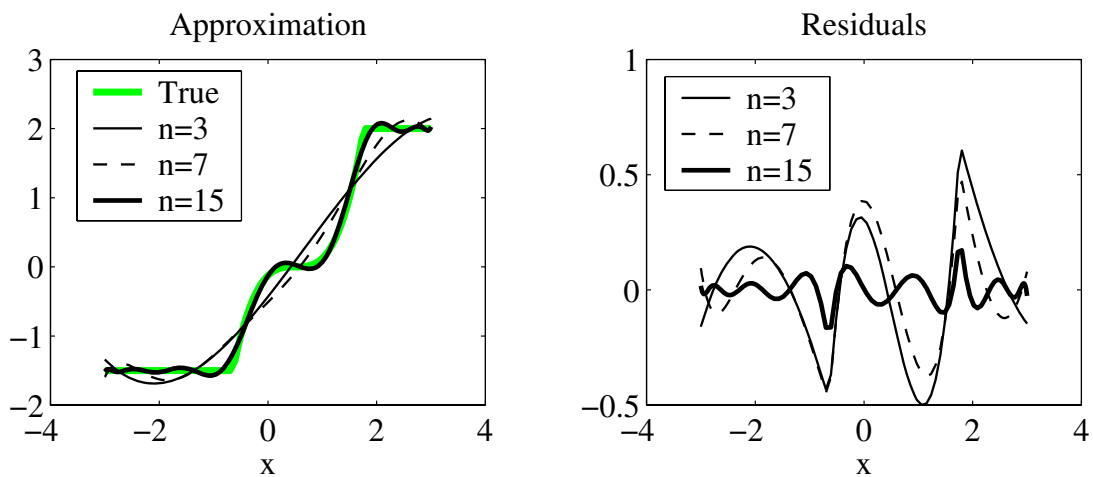
$$F(x) = \min\left(\max\left(-1.5, (x - 1/2)^3\right), 2\right).$$

Table 6 shows that the coefficients remain large even at degree 15.

Table 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 7: Non-smooth function:
$$F(x) = \min\left(\max\left(-1.5, (x - 1/2)^3\right), 2\right)$$



The residuals remain high at order 15.

That is, Chebychev approximations are well suited for smooth functions, but have difficulties in capturing kinks.

Nevertheless, increasing the order of the approximation drastically improves the approximation.

In the non-smooth function case, maybe 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 $x \in (\underline{x}, \overline{x}), G(x) \equiv \sum_{i=0}^{n} \beta_i T_i \left(2\frac{x-a}{b-z} - 1\right)$;
- for $x \in (\overline{x}, \infty), G(x) = 2$,

where $\underline{x}$ and $\overline{x}$

$$\begin{aligned} (\underline{x} - 1/2)^3 &= -1.5 \\ (\overline{x} - 1/2)^3 &= 2 \end{aligned}$$

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.

Piecewise interpolation

We have actually already seen piecewise approximation method: the linear interpolation method.

A more powerful and efficient method uses *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 $\mathcal{C}^{n-2}$ function on $[a, b]$,*

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

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

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

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

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

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

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

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

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.

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:

$$\{(x_i, y_i) : i = 0, ..., n\}.$$

We have in hand n intervals $[x_i, x_{i+1}]$, $i = 0, ..., n-1$, for which we search for n cubic splines:

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

The problem is to select $4n$ coefficients $\{a_i, b_i, c_i, d_i : i = 0, ..., n-1\}$ using $n+1$ nodes.

Hence, we need $4n$ identification conditions.

The first set of restrictions is provided by imposing the spline approximations to be exact at the nodes:

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

or simply

$$a_i = y_i \quad \text{for } i = 0, ..., n-1, \quad \text{and}$$
$$a_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3 = y_n.$$

The second set of restrictions: Continuity at the upper bound of each interval:

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

or, for $h_i = x_i - x_{i-1}$:

$$a_i = a_{i-1} + b_{i-1}h_i + c_{i-1}h_i^2 + d_{i-1}h_i^3 \quad \text{for } i = 1, ..., n-1. \tag{6}$$

Since we are dealing with a cubic spline interpolation, this requires the approximation to be $\mathcal{C}^2$.

Hence, the first and second order derivatives should be continuous.

This yields the following $n-1$ restrctions for the first order derivatives

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

or

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

Additional $n-1$ conditions for the second order derivatives

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

or

$$2c_i = 2c_{i-1} + 6d_{i-1}h_i \quad \text{for } i = 1, ..., n-1. \tag{7}$$

Altogether we have so far $4n-2$ equations, so two 2 more restrictions are needed.

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

   $$c_0 = c_n = 0.$$

   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.

   One may set

$$S_0'(x_0) = F'(x_0) \quad \text{and } S_{n-1}'(x_n) = F'(x_n).$$

   This is the so-called *Hermite spline*.

   However, the derivative of F may either not be known or does not exist.

   So, further source of information is needed.

3. 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:

$$\begin{aligned} S_0'(x_0) &= \frac{S_0(x_1) - S_0(x_0)}{x_1 - x_0} \quad \text{and} \\ S_{n-1}'(x_n) &= \frac{S_{n-1}(x_n) - S_{n-1}(x_{n-1})}{x_n - x_{n-1}}. \end{aligned}$$

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

$$b_0 = \frac{(y_1 - y_0)}{h_1} \quad \text{and } b_{n-1} = \frac{(y_{n-1} - y_n)}{h_n}.$$

We focus now on the natural cubic spline approximation, which imposes $c_0 = c_n = 0$.

First, note that the system the system of equation above has the recursive form

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

Substitution results into the expression for b_i gives:

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

Hence, (6) becomes

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

which we may rewrite as

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

Likewise, we have

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

Subtracting the last two equations yield

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

for $i=1,...,n-2$.

Taking (**??**) and (**??**) into account yields

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

for $i=1,...,n-2$.

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 is said to be tridiagonal (and therefore sparse). It is also symmetric and element-wise positive.

It is hence positive definite and therefore invertible

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

$$c = \begin{pmatrix} c_1 \\ \dots \\ c_{n-1} \end{pmatrix}, \quad \text{and}$$

$$B = \begin{pmatrix} \frac{3}{h_1}(y_2 - y_1) - \frac{3}{h_0}(y_1 - y_0) \\ \dots \\ \frac{3}{h_{n-1}}(y_n - y_{n-1}) - \frac{3}{h_{n-2}}(y_{n-1} - y_{n-2}) \end{pmatrix}.$$

So, we got all the c_i, $i = 1, n-1$, and can compute the b's and d's as

$$\begin{aligned} b_{i-1} &= \frac{y_i - y_{i-1}}{h_i} - \frac{1}{3}\left(c_i + 2c_{i-1}\right) h_i \quad \text{for } i = 1, ..., n-1; \quad \text{and} \\ b_{n-1} &= \frac{y_n - y_{n-1}}{h_n} - \frac{2c_{n-1}}{3h_n} \end{aligned}$$

and

$$\begin{aligned} d_{i-1} &= \frac{1}{3h_i}\left(c_i - c_{i-1}\right) h_i \quad \text{for } i = 1, ..., n-1; \quad \text{and} \\ d_{n-1} &= -\frac{2c_{n-1}}{3h_n}. \end{aligned}$$

Finally we have had $a_i = y_i$, for $i = 0, ..., n-1$.

Once the approximation is obtained, the evaluation of the approximation has to be undertaken.

The only difficult part which interval the value of the argument we want to evaluate belongs to.

That is, we have to find $i \in \{0, ..., n-1\}$ such that $x \in [x_i, x_{i+1}]$.

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 $\Delta = (b-a)/(n-1)$, for $i = 0, ..., n-1$.

In such a case, it is particularly simple to determine the interval as i is given by

$$i = [(a - x)/\Delta].$$

Nevertheless, there are some cases where it may be efficient to use non-uniform grid.

For instance, in the case of the non-smoothed function we consider above it would be useful to consider the following simple 4 nodes grid

$$\left\{-3, .5 - \sqrt[3]{1.5}, .5 + \sqrt[3]{2}, 3\right\},$$

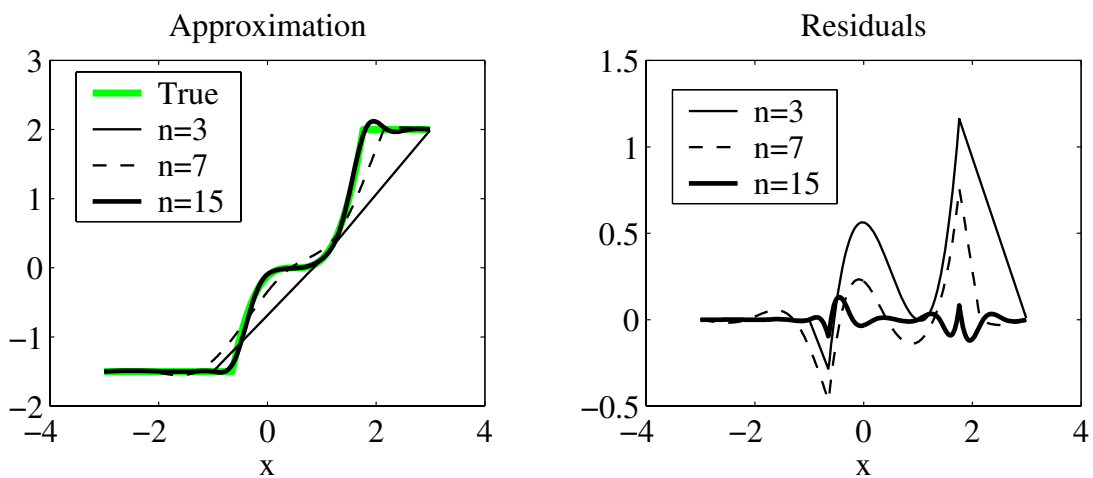
We saw that this grid yields a perfect approximation (recall that the central part of the function is cubic!)

As an example of spline approximation, Figure 8 reports the spline approximation to the non-smooth function

$$F(x) = \min\left(\max\left(-1.5, (x - 1/2)^3\right), 2\right),$$

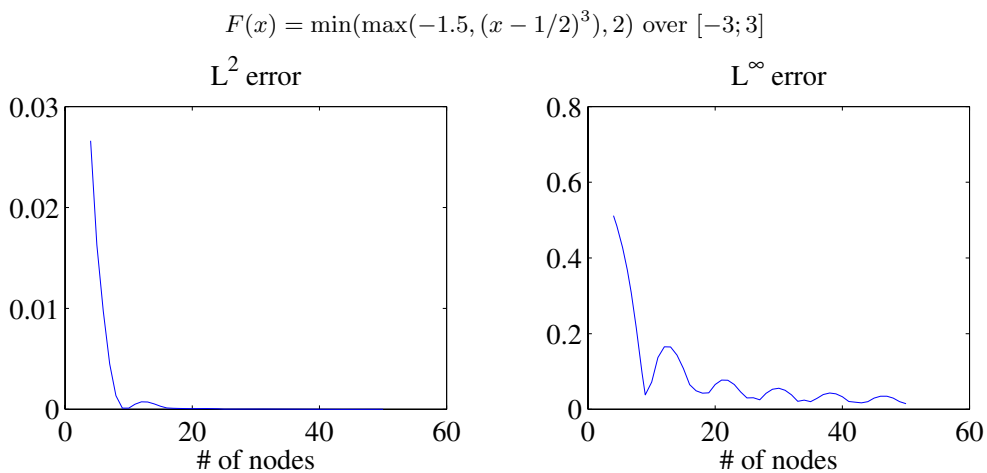
considering a uniform grid over the $[-3, 3]$ interval with $3, 7$ and 15 nodes.

Figure 8: Cubic spline approximation



In order to gauge the potential of spline approximation, we report in the upper panel of Figure 9 the L^2 and L^∞ error of approximation.

Figure 9: Approximation errors (Panel A)



The L^2 approximation error is given by $\|F(x) - S(x)\|$.

the L^∞ is given by $\max |F(x) - S(x)|$.

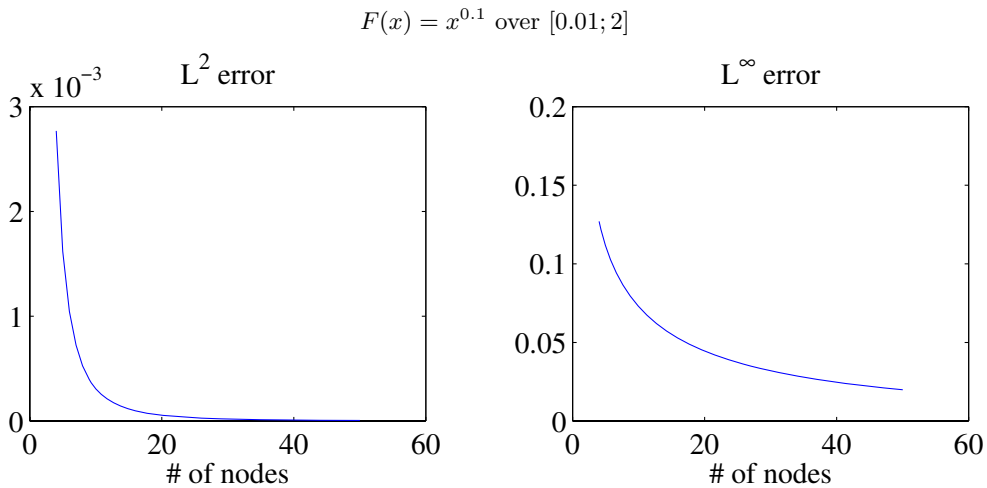
Increasing the number of nodes improves the approximation in that the error is driven to zero.

But, the convergence is not monotonic in the case of the L^∞ error.

This is because F, is not even $\mathcal{C}^1$ on the overall interval.

When we consider a smooth function this convergence is monotonic, as can be seen from the lower panel that report it for the function $F(x) = x^{\cdot 1}$ over the interval $[0.01, 2]$:

Figure 9: Approximation errors (Panel B)



All this is actually illustrated in the following Theorem:

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

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

and

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

This theorem actually gives upper bounds on the spline approximation.

These bounds decrease at a fast pace (power of 4) as the number of 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 $\mathcal{C}^\infty$ or that do not possess high order derivatives.

   Indeed, Theorem (7) indicates that the error term depends only on fourth order derivatives.

   Even if the fifth order derivative were badly behaved an accurate approximation may still 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 this method stems from the fact 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.

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 that F is a concave function, but it is costly to compute, so we do want to approximate it.

However, as seen from the previous examples, some methods generate oscillations in the approximation.

This can create an important problem since it implies that the approximation is not strictly concave, which is crucial in characterizing the dynamics of the variable x.

Also, the approximation of a strictly increasing function may be locally decreasing.

All this may create (1) some divergent path; (2) some spurious steady state; and consequently (3) spurious dynamics.

It is therefore crucial to develop shape preserving methods, in particular the curvature and monotonicity properties.

Shape preserving approximations

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 using information on both the level and the slope of the function to be approximated.

We consider two situations:

1. *Hermite interpolation*: Assumes that we have information on both the level and the slope of the function to be approximated.

2. *Lagrange data interpolation*: Assumes that no information on the slope of the function is available.

Both method was originally developed using quadratic splines.

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

$$y_i = F(x_i) \quad \text{and } z_i = F'(x_i), \quad i = 1, 2.$$

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 8 *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 monotonically increasing (decreasing) on the interval $[x_1, x_2]$, so is $S(x)$.

Furthermore, $z_1 > z_2 \quad (z_1 < z_2)$ indicates concavity (convexity), that is, $S(x)$ is such that

$$S''(x) = (z_2 - z_1)/(x_2 - x_1) < 0 \quad (> 0).$$

Problem: the conditions of the lemma are way too stringent, so the procedure need to be somewhat changed.

This may be done by adding a node between x_1 and x_2 and construct another spline that satisfies the lemma.

Lemma 9 *For every* $x^* \in (x_1, x_2)$ *there exist a unique quadratic spline that solves*

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

with a node at x^**. This spline is given by*

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

where

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

and

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

If the latter lemma fully characterize the quadratic spline, it gives no information on x^*, which therefore remains to be selected.

The point x^* need to be 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 $z_1 \overline{z} \geq 0 \quad (\leq 0)$.

This is equivalent to

$$2(y - y_1) \gtreqless (x^* - x_1) z_1 + (x_2 - x^*) z_2 \quad \text{if } z_1, z_2 \gtreqless 0.$$

This essentially deals with the monotonicity problem.

To tackle the curvature issue 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

$$(z_2 - \Delta)(z_1 - \Delta) \geq 0,$$

it the presence of an inflexion point in the interval $[x_1, x_2]$.

Hence, the interpolant cannot be neither convex nor concave.

Conversely, if $|z_2 - \Delta| < |z_1 - \Delta|$ and x^* satisfies

$$x_1 < x^* \leq \overline{x} \equiv x_1 + \frac{2\left(x_2 - x_1\right)\left(z_2 - \Delta\right)}{\left(z_2 - z_1\right)}$$

then $S\left(x\right)$ is convex (concave) if $z_1 < z_2 \quad (z_1 > z_2)$.

Furthermore, if $z_1 z_2 > 0$ it is also monotone.

If, in contrast, $|z_2 - \Delta| > |z_1 - \Delta|$ and x^* satisfies

$$\underline{x} \equiv x_2 + \frac{2\left(x_2 - x_1\right)\left(z_1 - \Delta\right)}{\left(z_2 - z_1\right)} \leq x^* < x_2$$

then $S\left(x\right)$ is convex (concave) if $z_1 < z_2 \quad (z_1 > z_2)$.

These set of results gives us the range of values for x^* that will insure that shape properties will be preserved.

1. Check if lemma 8 is satisfied. If so, set $x^* = x_2$ and set $S(x)$ as in lemma 9. Then STOP. Else, go to 2.

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

3. if $(z_1 - \Delta)(z_2 - \Delta) \geq 0$ set $x^* = (x_1 + x_2)/2$ and STOP. Else, go to 4.

4. if $|z_1 - \Delta| < |z_2 - \Delta| \geq 0$ set $x^* = (x_1 + \overline{x})/2$ and STOP. Else, go to 5.

5. if $|z_1 - \Delta| \geq |z_2 - \Delta| \geq 0$ set $x^* = (x_2 + \underline{x})/2$ and STOP.

Now we have at hand a value for x^* in $[x_1, x_2]$.

We then can apply it to each sub-interval to get $x_i^* \in [x_i, x_{i+1}]$ and then solve the general interpolation problem as explained in lemma 9.

Note here that everything assumes that with have Hermite data in hand – i.e., $\{x_i, y_i, \dot{z}_i : i = 0, .., n\}$.

However, in most cases the slope is typically unknown.

So, we need to adapt the algorithm to such situations.

Unknown slope: back to the Lagrange interpolation

Assume now that we do not have any data for the slope of the function.

That is we are only endowed with the Lagrange data

$$\{x_i, y_i : i = 0, .., n\}.$$

In such a case, we have to add the needed information—an estimate of the slope of the function—and proceed as in the case of the Hermite interpolation.

Schumaker proposes the following procedure for obtaining $\{\dot{z}_i : i = 0, .., n\}$.

Compute

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

and

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

for $i = 1, ..., n - 1$.

Then $\dot{z}_i : i = 0, ..., n$ can be recovered as

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

and

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

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

Up to now, all methods we have been studying are unidimensional whereas most of the model we deal with in economics involve more than $a$single variable.

We therefore need to extend the analysis to higher dimensional problems.

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 the data points

$$\{P_1, P_2, P_3, P_4\} = \{(1,0), (-1,0), (0,1), (0,-1)\} \quad \text{in } \mathbb{R}^2$$

and the corresponding data $z_i = F(P_i), \quad i = 1, ..., 4.$

Assume 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 of 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*.

We then introduce a second way of dealing with this problem, using *complete polynomials*.

In each case, we concentrate on how to use the Chebychev approximation.

Tensor product bases

The idea here is to use the tensor product of univariate functions to form a basis of multivariate functions.

Suppose we want to approximate a function $F : \mathbb{R}^2 \longrightarrow \mathbb{R}$ using simple univariate monomials up to order 2:

Define $\mathcal{X} = \left\{1, x, x^2\right\}$ and $\mathcal{Y} = \left\{1, y, y^2\right\}$.

The tensor product basis is given by

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

i.e., all possible 2-terms products of elements belonging to $\mathcal{X}$ and $\mathcal{Y}$.

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

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

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

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, if 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 $n = 2$!)

Complete polynomials

Complete polynomials have the advantage of growing only polynomially as the dimension increases.

Intuitively, 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*

$$\mathcal{B}^c = \left\{ x_1^{k_1} \times ... \times x_n^{k_n} : k_1, ..., k_n \geq 0, \sum_{i=1}^{n} k_i \leq \kappa \right\}.$$

To illustrate this more clearly, we consider the example of the previous section, i.e., $\mathcal{X} = \left\{1, x, x^2\right\}$ and $\mathcal{Y} = \left\{1, y, y^2\right\}$, and assume that $\kappa = 2$.

In this case, we end up with a complete polynomials basis of the type

$$\mathcal{B}^c = \left\{1, x, y, x^2, y^2, xy\right\} = \mathcal{B} \setminus \left\{xy^2, x^2y, x^2y^2\right\}.$$

We already seen this in the Taylor's theorem for multivariate case:

$$\begin{aligned} F(x) \simeq\ & F(x^*) + \sum_{i=1}^{n} \frac{\partial F}{\partial x_i}(x^*)(x_i - x_{i^*}) \\ & \vdots \\ & + \frac{1}{k!} \sum_{i_i=1}^{n} \dots \sum_{i_k=1}^{n} \frac{\partial^k F}{\partial x_{i_1} \dots \partial x_{i_k}}(x^*)\left(x_{i_1} - x_{i_1}^*\right)\left(x_{i_k} - x_{i_k}^*\right) \end{aligned}$$

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

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

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 an 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 is the rate at which the size of the basis increases.

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!

The complete polynomials and tensor product bases deliver the same degree of asymptotic convergence.

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.

It takes advantage of a very nice feature of orthogonal polynomials: They preserve the orthogonality property even if we extend them to higher dimensions.

Assume now that we want to compute the Chebychev approximation of a 2-dimensional function $F(x, y)$ over the interval $[a_x; b_x] \times [a_y; b_y]$.

Further assume (for simplicity) that we want to use a tensor product basis.

Then, the algorithm is as follows:

1. Choose a polynomial order for $x\left(n_x\right)$ and $y\left(n_y\right)$.

2. Compute $m_x \geq n_x + 1$ and $m_y \geq n_y + 1$ Chebychev interpolation nodes on $[-1; 1]$

$$\begin{aligned} z_k^x &= \cos\left(\frac{2k-1}{2m_x}\pi\right), \quad k = 1, ..., m_x, \quad \text{and} \\ z_k^y &= \cos\left(\frac{2k-1}{2m_y}\pi\right), \quad k = 1, ..., m_y \end{aligned}$$

3. Adjust the nodes to fit in both interval

$$\begin{aligned} x_k &= a_x + (1 + z_k^x)\left(\frac{b_x - a_x}{2}\right), \quad k = 1, ..., m_x \quad \text{and} \\ y_k &= a_y + \left(1 + z_k^y\right)\left(\frac{b_y - a_y}{2}\right), \quad k = 1, ..., m_y \end{aligned}$$

4. Evaluate the function F at each node to form

$$\Omega \equiv \{\omega_{kl} = F(x_k, y_l) : k = 1, ..., m_x; l = 1, ..., m_y\}$$

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

$$\alpha_{ij}=\frac{\sum_{k=1}^{m_x}\sum_{l=1}^{m_y}\omega_{kl}T_i^x\left(z_k^x\right)T_j^y\left(z_l^y\right)}{\left(\sum_{k=1}^{m_x}T_i^x\left(z_k^x\right)^2\right)\left(\sum_{l=1}^{m_y}T_j^y\left(z_l^y\right)^2\right)},$$

or simply

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

6. Compute the approximation function as

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

or simply 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

We use 20 nodes for both x and y. That is, there are 400 possible interpolation nodes.

Applying the above algorithm we obtain a matrix of coefficients reported in Table 7.

Table 7: Matrix of Chebychev coefficients (tensor product basis)

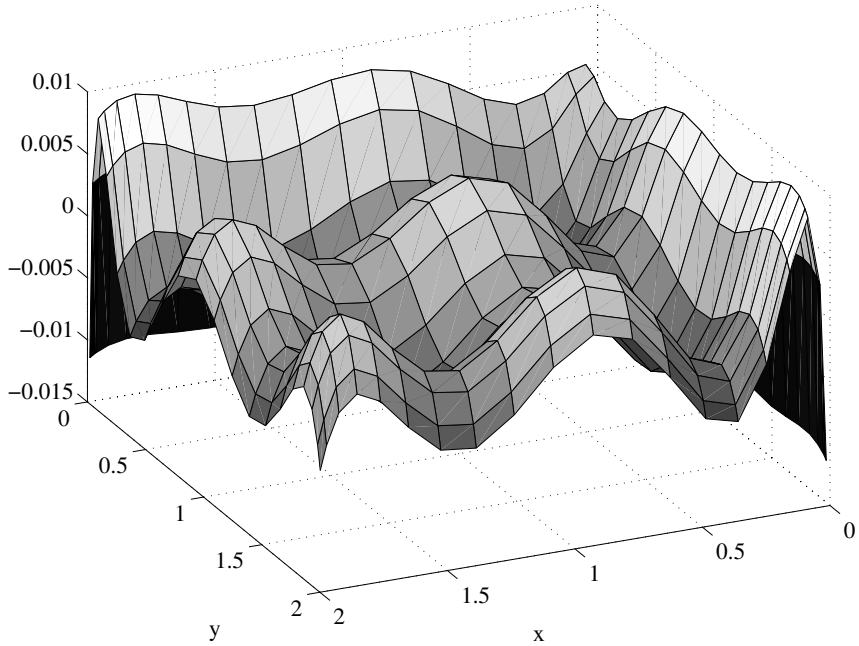
$k_x \backslash k_y$	0	1	2	3	4	5
0	2.4251	1.2744	-0.0582	0.0217	-0.0104	0.0057
1	1.2744	0.2030	-0.0366	0.0124	-0.0055	0.0029
2	-0.0582	-0.0366	0.0094	-0.0037	0.0018	-0.0009
3	0.0217	0.0124	-0.0037	0.0016	-0.0008	0.0005
4	-0.0104	-0.0055	0.0018	-0.0008	0.0004	-0.0003
5	0.0057	0.0029	-0.0009	0.0005	-0.0003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

Most of the coefficients that involve the cross-product of higher order terms are close to zero.

Hence, using a complete polynomial basis is likely to yield the same efficiency at a lower computational cost.

Figure 10 reports the graph of the residuals for the approximation.

Figure 10: Residuals: Tensor product basis



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');
```

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 $i + j \leq \kappa$. Let us compute is for $\kappa = 5$. This implies that the basis will consists of

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

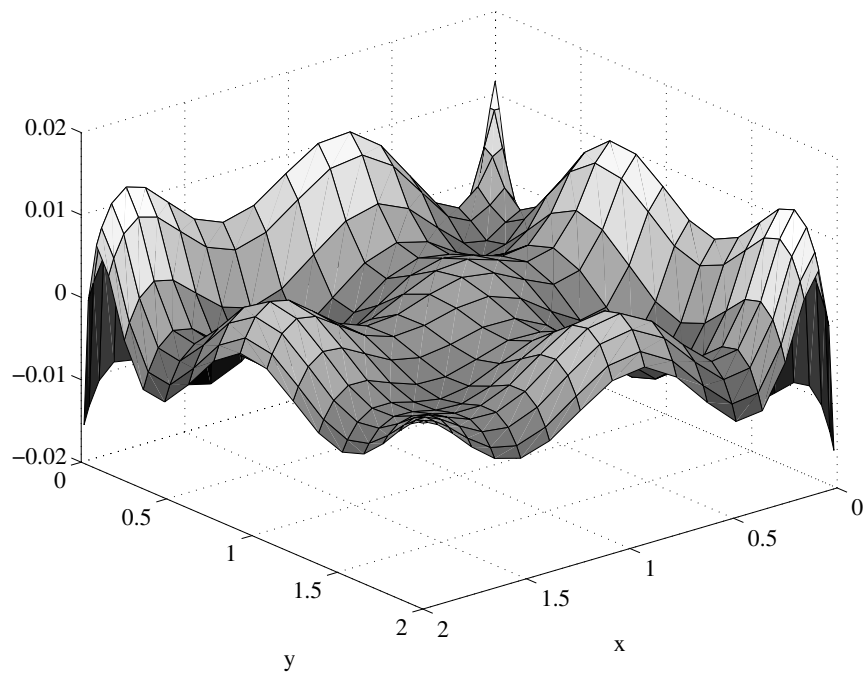
Table 8: Matrix of Chebychev coefficients (Complete polynomials basis)

$k_x \backslash k_y$	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	–	–	–	–	–

Note that, because of the orthogonality condition of Chebychev polynomials, the coefficients that remain are the same as the one we got in the tensor product basis.

Figure 11 report the residuals from the approximation using the complete basis.

Figure 11: Residuals: Complete polynomials basis



The "constrained" approximation yields quantitatively similar results to the tensor product basis.

The matlab code section, reports 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
```