

Lecture 1: Numerical Methods for the solution of model economies.¹

Equilibria are typically defined as prices, allocations, and laws of motions for endogenous states such that all agents optimize subject to their budget constraints and market clearing and resource constraints are satisfied. I.e., solving for an equilibrium requires that we a) optimize and b) solve a system of (typically non-linear) equations. In addition, we need to c) aggregate individual decision rules, and b) form expectations, both tasks which require that we evaluate integrals. Finally, our unknowns are in many instances functions, rather than finite-dimensional objects and we need tools we can use to solve functional equations.

Consider, e.g.,

$$V(k, A) = \max_{k'} u(k(1 - \delta) + f(k, A) - k') + \beta \int V(k', A') dG(A')$$

Here the unknowns are $V(k, A)$ and $k'(k, A)$: the agent's value function and optimal decision rules. To solve this problem requires solving the functional equation above, optimizing over k' , and evaluating the integral. We take up each of these tasks separately.

¹Virgiliu Midrigan. This is a concise summary of solution methods discussed in Judd (1998) and Miranda and Fackler (2002). These, as well as Numerical Recipes by Press et. al (see <http://www.nr.com/oldverswitcher.html> for a free copy of the book with lots of code for C and Fortran), are a valuable resource for anyone interested in numerical methods.

1 Linear equations

These are the building block of most numerical work. Many non-linear solution methods and functional approximation techniques rely on solutions to linear equations. There are 2 issues with linear equations of the size we will typically use: a) rounding errors that arise due to limited precision of computers, and b) speed and memory requirements.

Consider the system of linear equations in x :

$$Ax = b$$

where A is $n \times n$, x is $n \times 1$ and b is $n \times 1$.

The typical solution method is $L-U$ factorization in which *Gaussian* elimination is used to factor A into the product of a (row-permuted) lower triangular matrix and an upper triangular matrix:

$$A = LU$$

Then the original equation becomes

$$L(Ux) = b$$

Letting $y = Ux$ we can solve $Ly = b$ recursively for y by exploring the fact that L is (row-permuted) lower triangular. For example, if

$$L = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \text{ then } y_1 = b_1/a_{11}, y_2 = (b_2 - a_{21}y_1)/a_{22}, y_3 = (b_3 - \sum_{i=1}^2 a_{3i}y_i)/a_{33}.$$

Given y , one proceeds to solving $Ux = y$ similarly.

The key to remember is that this L-U algorithm involves $n^3/3 + n^2$ long operations (multiplications and additions) to solve an $n \times n$ system. It is implemented in MATLAB using the `\` (slash operator): $x = A \backslash b$. The alternative approach is to directly compute the inverse of A and then computing $A^{-1}b$ (`x=inv(A)*b` in MATLAB) and involves $n^3 + n^2$ long operations. Clearly, the L-U algorithm is preferred, unless one needs to solve the linear equations multiple times with the same A matrix, but for different (more than 3-4) b vectors.

Rounding errors may pose a problem and lead to inaccurate solutions and instabilities when solving linear equations. Rounding errors arise because computers must store real numbers in the form $m2^e$, where m and e are integers. Representing numbers other than integers or integral multiples of powers of 2 involves rounding errors (loss of significant digits). These are especially large when we subtract a large number from another large number: $10000000.2 - 10000000.1$ is equal to 0.09999999962747 on my computer, or when adding a large number to a small number. MATLAB has a number of special functions that are relevant in assessing its precision. `REALMAX` and `REALMIN` give you the largest and smallest number that your computer can represent. `EPS` gives you the distance between 1 and the next largest number it can represent. Operations like `(realmax+1)-realmax` or `1+eps/2` would thus give you 0

and 1, respectively.

To assess how rounding errors affect the accuracy of a linear solution, one typically computes the condition number of a matrix A :

$$k = \|A\| \times \|A^{-1}\|$$

for an upper bound of the “elasticity” of the solution vector x with respect to the vector b (`cond(A)` in MATLAB). Large condition numbers indicate larger errors. A good rule of thumb is that each power of 10 in the condition number represents an additional digit lost in the computed solution vector x . For example, a conditioning number of 10^5 indicates that x will have 5 fewer significant digits than b . The condition number is closely related to the concept of near-multicolinearity in econometrics and is a measure of how close to singularity a matrix is. For example

$$\begin{bmatrix} 1 & 2.000001 \\ 2 & 4 \end{bmatrix} \text{ has a condition number of } 1.25 \times 10^7.$$

One example where ill-conditioned matrices can arrive is in polynomial interpolation. Consider interpolating (approximating) the function $\exp(x)$ on the interval $[-10,10]$ with a linear combination of low-order polynomials: $1, x, x^2, \dots, x^n$. Let $\phi(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$ denote the approximant. Coefficients c_0 to c_n are the unknowns we seek. One interpolation strategy is to choose these $n + 1$ coefficients in order to ensure that $\exp(x)$ coincides with $\phi(x)$ at $n + 1$ equally-spaced points in $[-10,10]$: x_1, \dots, x_{n+1} . Then we have

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^n \\ 1 & x_2 & x_2^2 & x_2^n \\ 1 & \cdot & \cdot & \cdot \\ 1 & x_{n+1} & x_{n+1}^2 & x_{n+1}^n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \cdot \\ c_n \end{bmatrix} = \Phi c = \begin{bmatrix} \exp(x_1) \\ \exp(x_2) \\ \cdot \\ \exp(x_n) \end{bmatrix}$$

As n increases, the condition number of Φ increases as well. For example, for $n = 5$, $\text{cond}(\Phi)$ is of order 10^4 , while for $n = 9$ it increases to 10^8 and for $n = 20$ it increases to 10^{25} rendering the interpolation inaccurate. For this reason using $1, x, x^2, \dots, x^n$ as a basis for interpolation isn't a terribly good idea. Figure 1 illustrates the accuracy of the approximant to $\frac{1}{1+x^2}$ using polynomial interpolation.

You may also want to check the sections on iterative (Gauss-Jacobi and Gauss-Seidel) methods to solve linear equations in Judd (98) or Miranda and Fackler (02). The idea here is to start from a guess of the solution x_0 and solve for x in each successive round $j + 1$ and for each row i using a univariate linear equation in x_i and previous round's guesses for $x_{k \neq i}$. These methods differ slightly in how x_j^k is updated.

$$x_i^{j+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{k \neq i} a_{ik} x_j^k \right)$$

These methods tend to be used for larger systems in which the computational burden of using and L-U factorization is large. They are implemented as `gseidel` and `gjacobi` in the Compecon library.

2 Convergence rates and stopping rules

As we move from linear to non-linear equations we will mostly rely on iterative rather than direct solution methods. As above, these methods work by starting from a guess for the solution, using a particular scheme to update the guess and continuing until we are confident in the accuracy of the solution. Two questions come up: 1) how fast do we expect a particular solution method to converge to the actual solution? and 2) how to devise a stopping rule that ensures that our iterative scheme converges to the actual (unknown) solution?

2.1 Rates of convergence

Rates of convergence are a key property of the various iterative solution methods we look at. Consider a sequence $x^i \in R^n$ that satisfies $\lim_{i \rightarrow \infty} x^i = \bar{x}$. x^i is said to converge at rate q to \bar{x} if $\lim_{i \rightarrow \infty} \frac{\|x^{i+1} - \bar{x}\|}{\|x^i - \bar{x}\|^q} < \infty$. If $q = 2$, the rate of convergence is quadratic. If $\lim_{i \rightarrow \infty} \frac{\|x^{i+1} - \bar{x}\|}{\|x^i - \bar{x}\|} < \beta < 1$, x^i is said to converge linearly at rate β . For example, $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots, \frac{1}{2^i}$ converges linearly at rate $\frac{1}{2}$. The sequence $\frac{1}{2}, \frac{1}{4}, \frac{1}{16}, \frac{1}{256}, \dots, \frac{1}{2^{2^k}}$ converges quadratically.

2.2 Stopping rules

Suppose theory tells us that $\frac{\|x^{i+1} - \bar{x}\|}{\|x^i - \bar{x}\|} \leq \beta$ for all i (as opposed to in the limit as above). Suppose however that \bar{x} is unknown. Can we use information about x^i to learn about how close we are to \bar{x} ? Notice that $\|x^{i+1} - \bar{x}\| \leq \beta \|x^i - \bar{x}\|$ implies (using the triangular inequality) that $\|x^i - \bar{x}\| \leq \frac{\|x^{i+1} - x^i\|}{1 - \beta}$. Thus comparing two

adjacent values of our sequence in the iteration scheme tells us how close we are to the actual solution. If we desire $\|x^i - \bar{x}\| \leq \varepsilon$, then we can stop our algorithm when $\frac{\|x^{i+1} - x^i\|}{1 - \beta} \leq \varepsilon$.

Alternatively, if all we know is $\lim_{i \rightarrow \infty} \frac{\|x^{i+1} - \bar{x}\|}{\|x^i - \bar{x}\|} < \beta$, we can hope that i is sufficiently large so that we are at a point at which $\frac{\|x^{i+1} - \bar{x}\|}{\|x^i - \bar{x}\|} \leq \beta$ holds and apply the same stopping rule as above. In addition, if β is not known, we can try to estimate it by computing $\hat{\beta} = \max_l \frac{\|x^{k-l} - x^k\|}{\|x^{k-l-1} - x^k\|}$. Finally, an even more ad-hoc approach is to simply stop when $\frac{\|x^{i+1} - x^i\|}{1 + \|x^i\|} \leq \varepsilon$. Finally, good stopping rules in, say, non-linear solvers, will check convergence of both the x^i series as well as compare $f(x^i)$ to 0 and stop only when both $\|x^{i+1} - x^i\|$ and $\|f(x^i)\|$ have converged.

3 Non-linear equations

Derivative-based and derivative-free methods to solve non-linear equations $f(x) = 0$ are available. The latter are used when derivatives are unavailable analytically, or costly to compute numerically, or when numerical instabilities make derivative-based methods impractical. We discuss bisection, a method applicable only when f is a function of a single variable, function iteration, simplex-based methods, as well as the derivative-based Newton methods.

3.1 Bisection

Let $f : R \rightarrow R$ be a continuous function. The problem is $f(x) = 0$. Suppose you know a pair $a < b$ s.t., $f(a)$ and $f(b)$ have different signs. The intermediate value

theorem says that f must take all values between $f(a)$ and $f(b)$, including 0. Given a and b the algorithm evaluates the function at the midpoint $f(\frac{a+b}{2})$. The original $[a,b]$ interval is then shrunk to either $[a, \frac{a+b}{2}]$ if $\text{sgn}f(a) \neq \text{sgn}f(\frac{a+b}{2})$ or $[\frac{a+b}{2}, b]$. By construction and continuity, the new interval is guaranteed to contain at least one zero of f . In this manner, an ever decreasing sequence of intervals is generated up to a tolerance level ε . The advantage of this method is its robustness as it is guaranteed to find one root of f in a known amount of iterations at most $\log(\frac{b-a}{\varepsilon})/\log(2)$ iterations are needed to shrink the original interval to size ε . For example, shrinking the interval from length 10 to 10^{-7} requires 27 iterations. The disadvantage is that it only works for one-dimensional problems and converges slower (linearly at rate $\frac{1}{2}$) than methods that explore derivative information. It is implemented as `bisect` in the `Compecon` library.

The method can also be extended to a multi-variate problem by using a Gauss-Jacobi or Gauss-Seidel procedure in which one start from a guess of the vector x and solves the univariate equation $f_i = 0$ for one element of x , taking the other elements as given.

3.2 Function iteration

This method is very widely used in economics (e.g., value function iteration), and is fairly robust, although it can be slow. Suppose we have $f : R^n \rightarrow R^n$ and are interested in solving for a fixed point of $f : x = f(x)$. Given a guess x^i one updates it using $\hat{x}^{i+1} = f(x^i)$. As long as f is continuous, convergence of x^i implies convergence to a fixed point of f . A version of the contraction mapping theorem says that if f is

differentiable on a closed, bounded, convex set $D \subset R^n$ s.t. $\max_{x \in D} \|\frac{\partial f(x)}{\partial x}\|_\infty < 1$, then a) the fixed point problem $x = f(x)$ has a unique solution $\bar{x} \in D$, b) the sequence $x^{i+1} = f(x^i)$ converges to \bar{x} and c) $\lim_{i \rightarrow \infty} \frac{\|x^{i+1} - \bar{x}\|_\infty}{\|x^i - \bar{x}\|_\infty} \leq \|\frac{\partial f(\bar{x})}{\partial x}\|_\infty$. Thus function iteration is linearly convergent at the rate that depends on the Jacobian of f evaluated at the solution.

The iterative rule above is sometimes modified to $x^{i+1} = \omega f(x^i) + (1 - \omega)x^i$. If the system is unstable, an $\omega \in (0, 1)$ may help dampen instabilities. If things converge too slowly, an $\omega > 1$ may speed things up. Also note that we can solve $g(x) = 0$ by letting $f(x) = g(x) + x$ and solving $x = f(x)$ as above. Finally, it is useful in practice to start a non-linear solution method by using a couple of rounds of function iteration in order to find a good set of starting values for the faster, but less stable derivative-based methods.

3.3 Newton's method

Given $f : R^n \rightarrow R^n$, Newton's method solves $f(x) = 0$ by starting from a guess x^i for the true solution, taking a Taylor approximation of f around x^i :

$$f(x) \approx f(x^i) + \frac{\partial f(x^i)}{\partial x}(x - x^i)$$

and updating the guess using the zero of this linear approximation²:

$$x^{i+1} = x^i - \left[\frac{\partial f(x^i)}{\partial x} \right]^{-1} f(x^i)$$

²This is a shortcut notation, in practice one uses the LU decomposition rather than explicitly inverting the Jacobian.

If it works, Newton's method is generally fast, as convergence is quadratic. The problem is that the method can oscillate or even diverge away from the solution (see Judd's book for several examples), which typically happens if f' changes signs too often (as often happens if one interpolates an unknown function using low-order polynomials (see Figure 1). Compecon implements Newton's method using `newton` and employs backstepping: the Newton step $dx = \left[\frac{\partial f(x^i)}{\partial x} \right]^{-1} f(x^i)$ is cut in half if the function diverges away from 0 relative to the previous iteration. `newton` requires that the user supply both function and Jacobian information.

3.4 Broyden's method

This is a quasi-Newton method that relies on successive approximations to the Jacobian $\frac{\partial f(x^i)}{\partial x}$, rather than using the actual Jacobian which in practice may be expensive to compute or approximated using finite-differences. The algorithm starts from a guess for the Jacobian A^0 (say $A^0 = I$), and at each iteration computes the Newton step $x^{i+1} = x^i - (A^i)^{-1} f(x^i)$. The Jacobian guess is then updated using the smallest possible change consistent with the secant condition: $f(x^{i+1}) - f(x^i) = A^{i+1}(x^{i+1} - x^i)$ ³ :

$$A^{i+1} = A^i + [f(x^{i+1}) - f(x^i) - A^i(x^{i+1} - x^i)] \frac{(x^{i+1} - x^i)'}{(x^{i+1} - x^i)'(x^{i+1} - x^i)}$$

Theory says that if the original guesses for x^0 and A^0 are sufficiently close to the

³In one dimension, this condition uniquely determines A^{i+1} . In n dimensions, $f(x^{i+1}) - f(x^i) = A^{i+1}(x^{i+1} - x^i)$ is a system of n equations in n^2 unknowns, so the secant condition does not uniquely pin down A^{i+1} .

solution \bar{x} and $\frac{\partial f(\bar{x})}{\partial x}$, then Broyden's method converges superlinearly (at rate greater than 1 but less than quadratic). This is a statement about convergence of the x^i series, not A^i . Thus Broyden's typically requires fewer iterations than bisection or function iteration and can, unlike Newton's, be used in the absence of Jacobian information. Its problems are similar to those of the Newton method and performs poorly if the function oscillates. These problems may be minimized if one supplies a guess close to the solution (say, by using homotopy methods), and by transforming the original problem to make it more linear. Homotopy methods involve solving a succession of (ever improving) approximations to the original problem by starting from an approximation whose solution is readily available and gradually using the solution to the original approximation in subsequent steps until one converges to the original problem. For example, suppose the solution to $f(x, \varepsilon) = 0$ is readily available when $\varepsilon = 0$ but we are interested in solving things for $\varepsilon = 1$. Then one solves $f(x, 0)$ to produce $x(\varepsilon = 0)$ and gradually increases ε to 1 using $x(\varepsilon^i)$ as a guess in solving $f(x, \varepsilon^{i+1}) = 0$.

4 Optimization

The problem is to $\max_x f(x)$ where $f : R^n \rightarrow R$. We again have methods that ignore and methods that explore information about the curvature of the function.

4.1 Golden search

This resembles the bisection method used in non-linear solution methods and searches for a maxima in the interval a, b , where $a < b$. Given two points $a < x_1 < x_2 < b$, one evaluates f at x_1 and x_2 and replaces the original interval $[a, b]$ with $[a, x_2]$ if $f(x_1) > f(x_2)$ or with $[x_1, b]$ if $f(x_1) < f(x_2)$. A local maximum must be contained in the new interval because a point in its interior is greater than the endpoints (unless the original bound is the local maximum). This algorithm is repeated by ever shrinking the interval until a desirable tolerance level. An efficient implementation carefully chooses x_1 and x_2 : $x_i = a + \alpha_i(b - a)$ where $\alpha_1 = \frac{3-\sqrt{5}}{2}, \alpha_2 = \frac{\sqrt{5}-1}{2}$ in order to ensure that a) the length of the new interval is independent of whether a or b is discarded and b) to ensure that only one of the two x_i change from one evaluation to another in order to minimize the number of function evaluations. As bisection, this method only work in the univariate case, but is guaranteed to pin down a local maximum in a finite number of iterations. It is implemented as `golden` in the `Compecon` library.

4.2 Nelder-Mead (simplex)

This method also ignores curvature information. It is slow and unreliable (not guaranteed to converge to a maximum even when convergence criteria are met), but can handle non-smooth objectives for which gradient-based methods are less reliable. The algorithm evaluates $f : R^n \rightarrow R$ at $n + 1$ points that form a simplex in n dimensions, and reflects the simplex away from the point with the lowest function value. If the reflection succeeds in finding a new point that is higher than all other original

points on the simplex, the reflection is expanded further, if not, the reflection is contracted toward the original point with the lowest function value. The algorithm stops when the simplex is sufficiently small and all reflections fail to improve upon the original simplex.

4.3 Quasi-Newton methods

The idea is to solve for a zero of $\frac{\partial f(x)}{\partial x}$ using Newton's method above, and works if f is globally concave and the Hessian available. The Newton's setup is updated according to

$$x^{i+1} = x^i - \left[\frac{\partial^2 f(x)}{\partial x \partial x'} \right]^{-1} \frac{\partial f(x)}{\partial x}$$

The method however is of limited use as the Hessian is rarely easily computed and may not necessarily be negative definite everywhere. Typical implementations of the method this approximate the hessian using negative-definite approximations. The Compecon toolbox has a function `qnewton`, that implements 3 different ways of approximating the Hessian: a) with the identity matrix (method of steepest descent), b) Davidson-Fletcher-Powell (DFP), c) Broyden-Fletcher-Goldfarb-Shano (BFGS). The last two differ in how the Hessian approximation is updated with BFGS typically considered superior to DFP. As with non-linear solvers, one does not necessarily have to take the entire Newton step, but can rather search over the length of the step that maximizes the objective.

4.4 Conjugate gradient methods

This is similar to the quasi-Newton methods but does not require evaluating or approximating the Hessian, and is thus very useful for large scale problems in which n is large and a matrix of size $n \times n$ may be infeasible to store. In essence, conjugate gradient methods improve (converge in a smaller number of iterations) upon the method of steepest descent in which the Hessian is approximated with the identity matrix by using information about the previous step in addition to gradient information, in updating the step direction.

5 Differentiation and integration

5.1 Differentiation

Derivatives are approximated using $\frac{f(x+h)-f(x-h)}{2h}$ for small h . Using the Taylor expansion $f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3)$ and $f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3)$ it is clear that $f'(x) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2)$. A two-sided approximation is thus more accurate than a one-sided one for which the error is $O(h)$. h is usually chosen to be equal to $\sqrt{\text{eps}}$, the square root of machine precision. One can compute Hessians by applying the approximation above twice to the function in question. A more efficient and accurate approach is coded as `fdhess` in the `Compecon` library. `fdjac` spits out the Jacobian.

5.2 Integration

5.2.1 Newton-Cotes

The Compecon library includes two familiar Newton-Cotes routines for computing integrals of a function $f(x)$ over $[a, b]$: the trapezoid and Simpson's rule. I discuss the trapezoid method in more detail, Simpson's method is very similar and discussed at the end.

The trapezoid rule divides the interval into subintervals of equal length, approximates f linearly and adds up the areas of each trapezoid under the line. Let $x_i = a + (i - 1)h$, where $h = \frac{b-a}{n-1}$, where n is the number of nodes used for approximation. The integral from x_i to x_{i+1} is approximated with $\int_{x_i}^{x_{i+1}} f(x)dx \approx \int_{x_i}^{x_{i+1}} (\alpha + \beta x) dx$, where $\beta = \frac{f(x_i) - f(x_{i+1})}{h}$ and $\alpha = f(x_{i+1}) - \beta x_{i+1}$. Thus $\int_{x_i}^{x_{i+1}} (\alpha + \beta x) dx = \frac{h}{2}[f(x_i) + f(x_{i+1})]$. and $\int_a^b f(x)dx = \sum_{i=1}^n \int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{i=1}^n \omega_i f(x_i)$, where $\omega_1 = \omega_n = \frac{h}{2}$ and $\omega_i = h$ for $i \neq 1, n$. The Compecon function `[x,w]=qnwtrap(n,a,b)` produces the evenly spread nodes, x and the weights associated with each node, w . Given these nodes and weights, the integral is computed using `w'*f(x)`.

To integrate over more than one dimension is straightforward. Suppose we want to approximate $\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x_1, x_2)dx_1dx_2$. As above, this can be approximated using n_i nodes in each dimension: $\sum_{i=1}^{n_2} \sum_{i=1}^{n_1} w_{1i}w_{2i}f(x_1, x_2)$. Given one-dimensional nodes and weights, X_1, X_2, w_1, w_2 , one can thus form the cartesian product $X = X_1 \times X_2$ and the tensor (Kronecker) product of w_1 and w_2 : $w = w_1 \otimes w_2$ and compute $\sum_{i=1}^{n_1 n_2} w_i f(X_i)$. `qnwtrap` can be used to integrate in more than 1 dimension. If one supplies a $1 \times d$ vector of nodes n along each dimension $1, \dots, d$, and $1 \times d$ vectors of bounds a and b , `[x,w]=qnwtrap(n,a,b)` produces the multi-variate nodes x (the

Cartesian product of univariate nodes) and weights w , both vectors of size $n_1 n_2 \dots n_d \times 1$, that can be used to compute the approximant of the integral $w' * f(x)$. Try reading the function `qnrtrap` for the exact details of how Compecon extends univariate nodes and weights to a multi-dimensional setting using just 2 lines of code that construct a Kronecker and Cartesian product.

Simpson's method works similarly, but approximates the original function using a quadratic rather than a line that passes through x_{i-1}, x_i, x_{i+1} . If the function is smooth, Simpson's method provides a much more accurate approximation (it exactly integrates any third-order polynomial, in contrast to the trapezoid rule that exactly integrates only a line). The method is implemented using `qnwsimp` in the Compecon library and its calling syntax and extension to the multivariate case is identical.

5.2.2 Gaussian quadrature

Gaussian quadrature nodes and weights are computed relative to a particular weighting function $w(x)$ and used to approximate $\int_a^b f(x)w(x)dx$. They are thus very useful in evaluating expectations as efficient ways of computing the approximation are available for a number of popular density functions (normal, log-normal, beta, uniform, gamma). Given an order of approximation n , one chooses n quadrature nodes x_1, \dots, x_n and n weights $\omega_1, \dots, \omega_n$ in order to satisfy $2n$ moment-matching conditions:

$$\int x^k w(x) dx = \sum_{i=1}^n \omega_i x_i^k \text{ for } k = 0, \dots, 2n - 1$$

The integral is then approximated using

$$\int f(x)w(x)dx \approx \sum_{i=1}^n \omega_i f(x_i)$$

Gaussian quadrature approximation of an integral is order $2n - 1$ exact. In other words, it exactly approximates any polynomial up to order $2n - 1$. When $w(x)$ is a density function, the moment-matching conditions that underly Gaussian quadrature give the method a very useful interpretation: the original random variable with density $w(x)$ is approximated using a discrete random variable with n mass points x_1, \dots, x_n and weights w_1, \dots, w_n that shares the first $2n - 1$ moments with the original distribution (one moment is lost in the requirement that the weights sum up to 1). Thus a 5-node discretization of a Normally distributed random variable using Gaussian quadrature produces a discrete random variable that accords with the first 9 moments of the Normal distribution.

The extension to multi-variate case using tensor products is straightforward, as above. Compecon has routines that produce Gaussian quadrature nodes and weights for the following distributions: Gaussian (`qwnorm`), Log-normal (`qnwlogn`), Uniform (`qnwunif`), Beta (`qnwbeta`), Gamma (`qnwgamma`), as well as Gauss-Legendre nodes and weights (for the weighting function $w(x) = 1$) (`qnwlege`). It also produces nodes and weights for the weighting function $w(x) = (1 - x^2)^{-\frac{1}{2}}$ (Gauss-Chebyshev quadrature) for which nodes and weights are easy to produce (available analytically). See Judd (98) for how one can use a change of variable to apply Gauss-Chebyshev quadrature to integrate any $\int f(x)dx$. This is coded as `qnwcheb` in Compecon library.

Gaussian quadrature is typically preferred when the function being integrated

possesses continuous derivatives. Intuitively, Gaussian quadrature efficiently chooses both nodes and weights for integration, whereas Newton-Cotes methods arbitrarily pick the nodes (equally-spaced and independent of the function to be approximated), as opposed to choosing them relative to a particular weighting function. However, Newton-Cotes methods limit the size of the error in the presence of kinks as by construction, errors in one interval of the state-space do not translate to other intervals.

5.2.3 Monte-Carlo Simulation

These rely on laws of large numbers to generate N draws from a particular distribution $w(x)$ and approximate $\int f(x)w(x)dx$ with $\frac{1}{N} \sum_{i=1}^N f(x_i)$. These methods have low convergence rates (i.e., the accuracy decreases slowly with N) and thus require a large N for accurate approximations. Judd and Miranda & Fackler discuss a number of sampling techniques that can be used to reduce the variance of the integral estimate in Monte-Carlo methods. In general, one uses these when working in many dimensions and the curse of dimensionality makes Gaussian quadrature and Newton-Cotes methods impractical.

6 Function approximation: Projection Methods

6.1 Interpolation

The simpler case is when a function $f : D \subset R^k \rightarrow R^m$ can be evaluated (at a non-zero marginal cost) at any point $x \in D$, but cannot be characterized analytically. For example, f can map model parameters into a set of moments whose com-

putation involves solving (numerically) a non-linear system of equations. Explicit characterization of f is thus unavailable, and evaluating f may be very expensive. Approximating it with \hat{f} may thus be useful, eg., if one needs to evaluate f a large number of times. Ideally, we would like to come up with an approximation method that minimizes the supremum norm of $f - \hat{f}$:

$$\|f - \hat{f}\|_{\infty} = \sup_{x \in D} |f(x) - \hat{f}(x)|$$

The Weierstrass theorem says that if f is continuous and real-valued, it can be well approximated by a polynomial p with arbitrary degree of accuracy. In other words, for any $\epsilon > 0$ there exists a polynomial p such that:

$$\|f - p\|_{\infty} < \epsilon$$

This motivates approximating f with a polynomial. Let's focus for now on the one-dimensional case: $k = 1$. As with integrals, extension to higher dimensions is straightforward using tensor products.

Notice that the original problem is infinite-dimensional. One makes this problem tractable by approximating f with a (usually linear) combination of n basis functions and choosing the unknown coefficients on these basis functions so that the approximant accords with the original function in a particular sense.

In practice, using the monomials x^n as a basis function and approximate f using $\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 x^3 + \dots$ isn't a terribly good idea because one would work with a ill-conditioned system of linear equations as discussed earlier, as x^n is collinear with

x^{n+1} for large n . As a result, one typically employs as a basis function polynomials that are orthogonal with respect to a particular basis function $w(x)$:

$$\int_D p_n(x)p_m(x)w(x) = 0$$

When $w(x) = (1-x^2)^{-\frac{1}{2}}$, the family of polynomials given by $\phi_n = \cos(n \cos^{-1} x)$ (for x in $[-1,1]$) is the so-called Chebyshev family that is very popular in practice because they possess great approximation properties. The Chebyshev basis functions can also be written as

$$\begin{aligned}\phi_1(x) &= 1 \\ \phi_2(x) &= x \\ \phi_3(x) &= 2x^2 - 1 \\ \phi_4(x) &= 4x^3 - 3x \\ \phi_5(x) &= 8x^4 - 8x^2 + 1 \\ \phi_n(x) &= 2x\phi_{n-1} - \phi_{n-2}\end{aligned}$$

See Figures 2 and 3 for a comparison of monomials and Chebyshev basis functions.

Given a choice of basis functions, we approximate f using $\hat{f} = \sum_{i=1}^n c_i \phi_i(x)$ where c_i are coefficients to be determined. To pin down the n unknown coefficients, one requires that the approximant \hat{f} accords with f at $l \geq n$ nodes x_1, \dots, x_l along the state-space. If $l = n$, the problem reduces to solving a linear system of equations $f(x_i) = \hat{f}(x_i)$ as the system is linear in α_i . If $l > n$, one chooses the α_i s to minimize

$$\sum_{i=1}^l \left[f(x_i) - \hat{f}(x_i) \right]^2 .$$

Notice, that if $l = n$ one can rewrite the problem as $\Phi c = F$ where Φ is an $n \times n$ matrix of the n Chebyshev polynomials evaluated at n nodes along the state-space, c is $n \times 1$ vector of unknown coefficients, and F is an $n \times 1$ vector that stores the original function evaluated at the n nodes. Given Φ and F , one uses a linear solver to pin down c if $l = n$ or $l > n$. Similarly, if $l > n$, let Φ be the $l \times n$ matrix of the n Chebyshev polynomials evaluated at n nodes along the state-space and the problem is again linear: $c = (\Phi' \Phi)^{-1} \Phi' F$

at n nodes along the state-space, c is $n \times 1$ vector of unknown coefficients, and F is an $n \times 1$ vector that stores the original function evaluated at the n nodes. Given Φ and F , one uses a linear solver to pin down c if $l = n$ or $l > n$. Similarly, if $l > n$, let Φ be the $l \times n$ matrix of the n Chebyshev polynomials evaluated at n nodes along the state-space and the problem is again linear: $c = (\Phi' \Phi)^{-1} \Phi' F$

The final question is: how do we choose the nodes x_i at which to compute the original function? It turns out that equally-spaced nodes have poor properties and do not guarantee that the approximation error decreases as the number of nodes used for interpolation increases. A well-known example is the approximation of the Runge function: $f(x) = \frac{1}{1+x^2}$. It turns out that Chebyshev nodes (zeros of Chebyshev polynomials), instead of equidistant nodes, provide a much better choice that is nearly optimal. These nodes are given (on the interval $[-1,1]$)⁴ by $x_i = \cos\left(\frac{2i-1}{2l}\pi\right)$ where l is the number of nodes used.

Chebyshev interpolation (use of Chebyshev basis functions at Chebyshev nodes) is the method of choice if f is sufficiently smooth (at least continuously differentiable). Rivlin's theorem states that $f : [a, b] \rightarrow R$ is C^k for some $k \geq 1$, and I_n is the n -degree polynomial interpolation at the zeros of the Chebyshev polynomials $\phi_n(x)$,

⁴Even though they are defined on $[-1,1]$, one can extend them to $[a,b]$ by evaluating the nodes z_i on $[-1,1]$ and adjusting the nodes to $[a,b]$ by adjusting the nodes $x_i = (z_k + 1) \frac{b-a}{2} + a$

then

$$\|f - I_n\|_\infty \leq \left(\frac{2}{\pi} \log(n+1) + 1\right) \frac{(n-k)!}{n!} \left(\frac{\pi}{2}\right)^k \left(\frac{b-a}{2}\right)^k \|f^{(k)}\|$$

Even if $k = 1$ (continuously differentiable function), this upper bound is

$$\frac{\left(\frac{2}{\pi} \log(n+1) + 1\right) \pi}{n} \frac{\|f^{(1)}\|}{4} (b-a)$$

This theorem also says that Chebyshev approximation works well if the function to be approximated is smooth i.e. the gradient sufficiently small everywhere.

The Compecon library has 2 routines, `chebnode` and `chebbas` for computing chebyshev nodes and basis functions that can be used for interpolation. Suppose I want to approximate $\exp(-x)$ on $[-3, 5]$ using a 7-th order polynomial and pinning down the 7 coefficients by ensuring that my approximant accords with $\exp(-x)$ at 15 chebyshev nodes. The code I use is

```
node=chebnode(15,-3,5); %compute 15 chebyshev nodes on [-3,5]
Phi=chebbas(7,-3,5,node); %compute a 15 by 7 basis matrix of 7 Chebyshev poly-
nomials evaluated at the 15 nodes
F=exp(-node); %evaluate the original function at 15 nodes
c=Phi\F; %use Matlab's linear solver to retrieve the 7 coefficients. (\ is a linear
solver and least squares minimizer in Matlab)
```

How do I evaluate the accuracy of my approximant. Ideally, I would like to compute $\|f - I_n\|_\infty$, but this is not feasible. I will thus do the following: construct a grid of points on $[a,b]$ that is of size much larger than $n = 7$, say $T = 1000$. Evaluate f and the approximant at the T nodes and compare the difference. In Matlab, you

can do this using:

```
T=1000;
x=linspace(-3,5,T)';
Phinew=chebbas(7,-3,5,x); %7th order Chebyshev approximant evaluated at x
Fapprox=Phinew*c; %evaluate approximant
Ftrue=exp(-x);
disp(max(abs(Fapprox-Ftrue))); %compute maximum difference
```

Also note that `chebbas` can evaluate derivatives or integrals of $\phi_n(x)$, which is very useful when one wants to solve differential equations or wants to choose coefficients so as to ensure that both the level, but also the derivatives of the original function are well approximated.

Extension to more than one dimension is straightforward. Suppose one wants to approximate $f(x, y)$. One first generates univariate nodes n_x and n_y and basis functions ϕ_i^x and ϕ_i^y as above and then constructs the cartesian product of the nodes (x, y) , computes the multi-variate basis functions as the tensor product of ϕ_i^x and ϕ_i^y and solves for the $n_x n_y$ unknown coefficients by requiring that the original function accords with the approximant: $I = \sum_{i_x=1}^{n_x} \sum_{i_y=1}^{n_y} c_{i_x i_y} \phi_{i_x}^x(x) \phi_{i_y}^y(y)$ at whatever choice of nodes (x, y) desired. Letting Φ_x and Φ_y denote the univariate basis functions evaluated at the one-dimensional nodes, the approximant I can be compactly written as Φc where $\Phi = \Phi_2 \otimes \Phi_1$ (or more generally $\Phi_k \otimes \Phi_{k-1} \otimes \dots \otimes \Phi_1$. Given univariate nodes x and y , one can form their Cartesian product using the Compecon function `gridmake(x,y)`. One problem that arises in multi-variable interpolation is the curse of dimensionality: the size of the Φ matrix to be inverted quickly (exponentially) grows

with the number of additional dimensions. One can circumvent this problem by using complete polynomials and/or by using sparse grid methods in which the function is approximated at a sparse grid of points rather than their Cartesian product. I have some (rough) notes on this on my webpage.

Compecon has a set of routines that automatically produces these tensor and Cartesian products, and also computes basis functions for cubic and linear spline interpolation in an arbitrary number of dimensions. Read the documentation for `fundef`, `fundefn`, `funeval` and `funbas` in the Compecon library. I have also coded up (fairly poorly) some extensions of these function that allow one to use sparse grid matrices. These are available in the Smolyak folder in the compecon toolbox on this course's webpage.

6.2 Splines

Chebyshev approximation performs poorly when the original function is not smooth, has discontinuous derivatives etc.. Because of its global nature (it attempts to provide a uniformly good approximation on the entire state-space), errors in a small interval can translate to other intervals. Splines approximate the original function by partitioning the state-space into smaller subintervals and approximating the function interval by interval using low-order polynomials (lines or cubic) that are required to satisfy conditions similar to those of smooth-pasting and value-matching (i.e., the approximant is required to be C^1). Again, these are coded as `splibas`, `splinode`, `splidef` in the Compecon library and one can use them in a manner that is very similar to how we have used Chebyshev polynomials above. Here, one typically chooses equally

spaced intervals, but if one believes that there is more action in some region of the state-space, one can easily concentrate more intervals in that region. Again, one can use `fundef`, `fundefn`, `funeval` and `funbas` for multi-dimensional extensions.

6.3 Functional equations

Given the tools above, solving functional equation is now easy. Suppose we want to solve for an f that satisfies $g(f) = 0$. One replaces the original f using a linear combination of known basis functions: Φc and chooses c in order to satisfy $g(\Phi c) = 0$ at, say, Chebyshev nodes, using a non-linear solver. As above, one transforms the original infinite-dimensional problem to a finite-dimensional one.

Interpolation of $1/(1+x^2)$ using polynomials

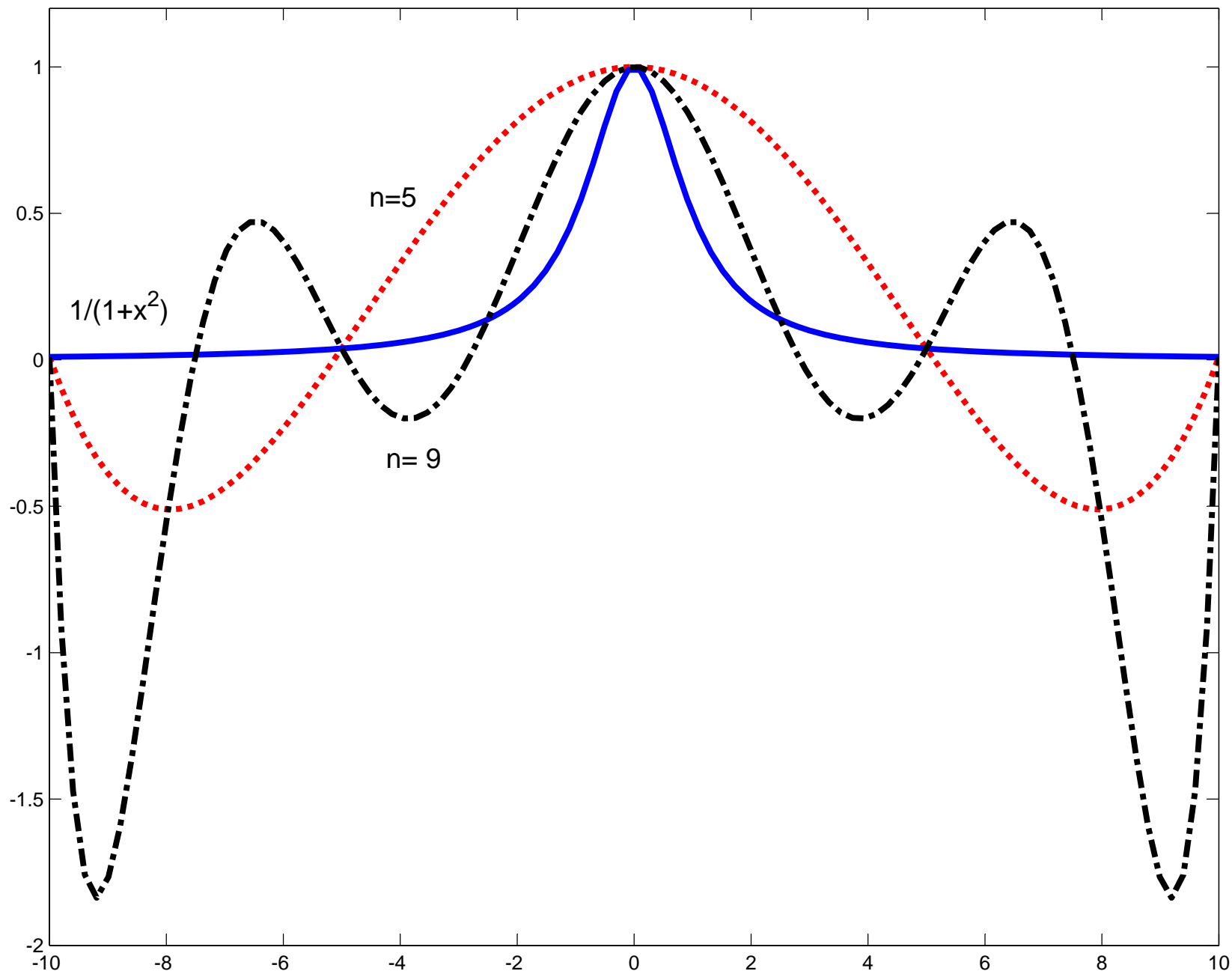


Figure 2: Monomial Basis Functions

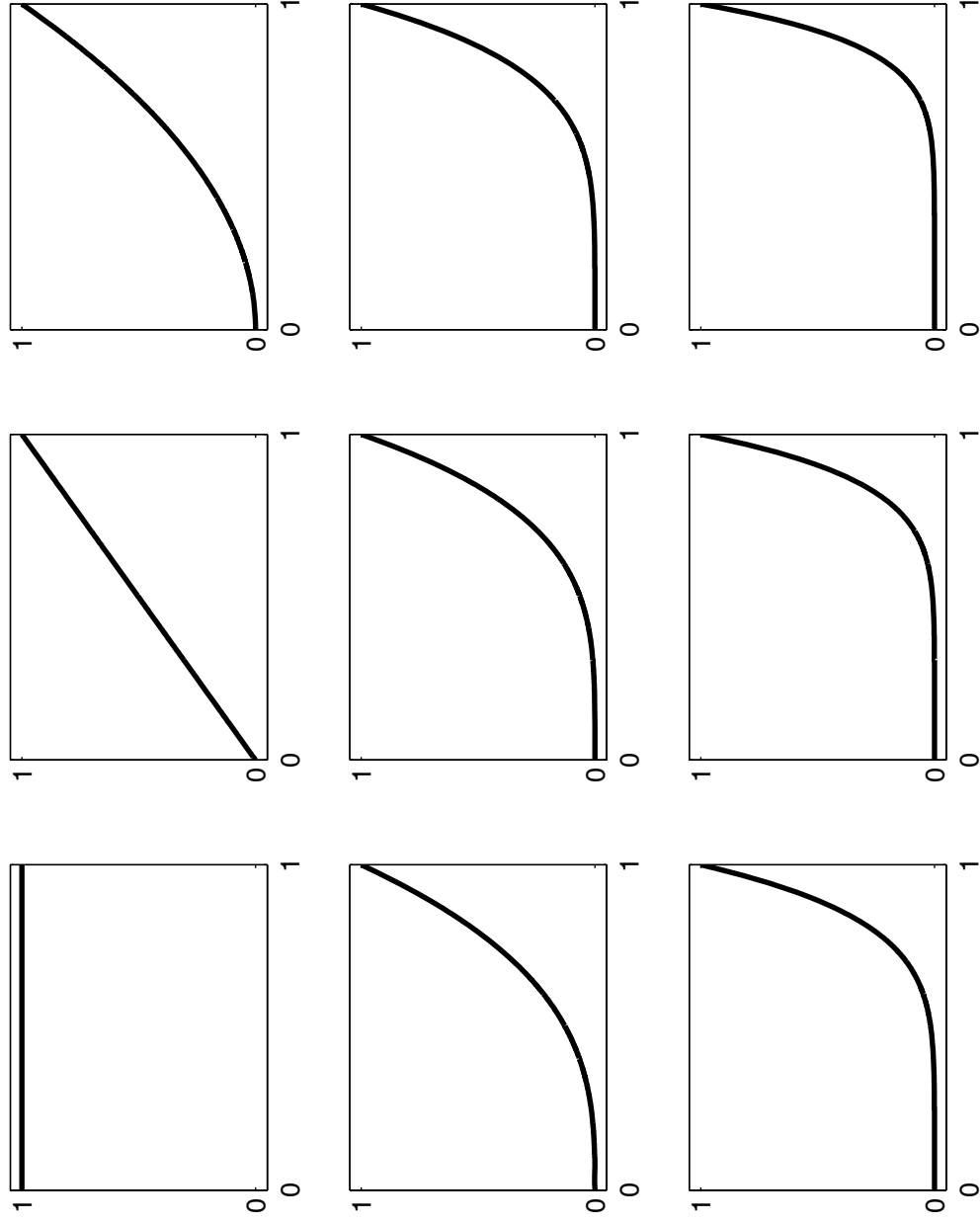


Figure 3: Chebyshev Polynomial Basis Functions

