Lecture Notes 1: Introduction to Computation

Ivan Rudik

ECON 509: Computational Methods

August 29, 2017

1 Why do we need computational methods?

Thus far, every economic problem you have faced has likely had a closed-form, analytic solution, e.g., Cournot competition with linear demand and constant marginal costs. However, not all economic models have closed-form solutions, nor can they be simplified so that they possess analytic solutions without losing critical economic content. The former is true for a large set of models, and the latter is particularly true for dynamic models. Let's look at a few examples (Miranda and Fackler, 2002):

Q: Suppose we have a constant-elasticity demand function: $q = p^{-0.2}$ and a quantity demanded of $q = 2$ in equilibrium. What market price p clears the market?

This has a simple solution where we just invert the quintic root and arrive at a closed-form solution of $p = q^{-5}$. Your calculator can do the rest. Now what if demand exhibits slightly more complex behavior?

Q: Suppose we have a constant-elasticity demand function: $q = 0.5p^{-0.2} + 0.5p^{-0.5}$ and a quantity demanded of $q = 2$ in equilibrium. What market price p clears the market?

Does a solution exist? *Yes.* The demand function is continuous, takes a value larger than 2 at $p = 0.1$ and a value smaller than 2 at $p = 0.2$. By the Intermediate Value Theorem it *must* take a

value of 2 somewhere in the interval $(0.1, 0.2)$. But what is this value?

If we substitute in $t = p^{-0.1}$, we have $q = 0.5t^2 + 0.5t^5$. The demand function is actually a fifth-order polynomial in disguise. Can we solve for t now? No, inverses of fifth-order polynomials are not guaranteed to exist! Therefore we have no closed-form solution for the price p that clears the market. Now a second, more interesting example:

Q: Consider a two period model of an agricultural commodity market. Acreage decisions must be made before knowing the realization of per-acre yield and the price at harvest. The farmer has an expectation of the harvest price and makes decisions as a function of the expectation: $a = 0.5 + 0.5\,E[p]$. After planting, the random yield per-acre, $\hat{y}$ is realized, producing a quantity $q = a\hat{y}$ of the crop. Demand for the crop is given by the inverse demand function $p = 3 - 2q$. Suppose $\hat{y}$ is exogenous and has mean 1 and variance 0.1. How much acreage does the farmer plant?

If we substitute the second and third equations into the first and then take the expectation, we have that in equilibrium the farmer plants $a = 1$ acre.

Q: Suppose the government implements a price floor on the crop of $p = 1$. What is the farmer's optimal level of planted acreage with the price floor?

The farmer's acreage planting decision changes to $a = 0.5 + 0.5\,E[\max(1, p)]$. With the information we have, we can't solve for a analytically because we cannot pass the expectation through a nonlinear function, i.e., $E[\max(1, p)] \neq \max(1, E[p])$. Is the efficient level of acreage greater than, less than, or equal to one? Think about Jensen's inequality. The maximum operator is a convex function, and expectations of convex functions are higher than the convex function at the expectation of the random variable. So with the price floor the expected price is higher and we should see the farmer plant more acreage than before.

Both of these problems are solvable numerically with simple root-finding algorithms like Newton's methods. Returning to the first problem, we can solve it with the following code:

```
# Plot the function
using PyPlot

# We know solution is between .1 and .2
x = linspace(.1,.2,1000)
z = 2*ones(length(x))
```

```
# Price function minus equilibrium quantity of 2
price(p) = 0.5*p.^(-0.2) + 0.5*p.^(-0.5)

# Get corresponding quantity values at these prices
y = price(x)

# Plot
plot(x, y, color="red", linewidth=2.0)
plot(x, z, color="blue", linewidth=0.5, linestyle="--")
title("Where's the root?")



# Initial price guess
p = 0.20

# Initialize stepsize
deltap = 1.0e10

# Demand function
demand_function(p) = (.5*p^(-.2)+.5*p^( -.5) -2)
# Gradient of demand function
demand_function_gradient(p) = (.1*p^(-1.2) +.25*p^(-1.5))

# Loop through Newton's method until tolerance is met
while abs(deltap) > 1e-8
    deltap = demand_function(p)/demand_function_gradient(p)
    p += deltap
end

println("The equilibrium price is \$$(round(p*1000)/1000).")
```

The resulting equilibrium price is p=0.154 (rounded to three digits), which is in the interval $(0.1, 0.2)$ mentioned above. The idea behind Newton's method is to begin with a guess of the solution to the problem: $p = 0.25$, which is almost surely not correct. However, from this initial guess we can construct a tangent line to the demand function at $p = 0.25$ and find where this tangent line intersects the x-axis. This tends to get us closer to the actual solution! We take the root of the

tangent line as our new guess for the root of the demand function and repeat this process until we are "close enough," in the sense that the difference between our new guess and previous guess for the root are small. When the difference is small, this implies we are close to a root. There exist proofs which demonstrate convergence of Newton's method under certain conditions. We will go more in depth into Newton's method and more complex optimization routines in future classes.

For the second problem, we can solve for equilibrium acreage using the following code:

```
using CompEcon
# Create quadrature
y, w = qnwnorm(10, 1, 0.1)

# Initial guess for acreage
a = 1.
p = 0. # Need to define price outside the loop to have correct scope
diff = 100

# Loop through solution algorithm until tolerance is met
while diff > 1e-8
    # Save old acreage
    aold = a
    # Get price at all the quadrature nodes y
    p = 3.-2.*a*y
    # Compute expected price with price floor
    expectation = w'*max(p,1)
    # Get new acreage planted given new price
    a = 0.5 + 0.5*expectation[1]
    # Get difference between old and new acreage
    diff = abs(a-aold)
end

println("The optimal number of acres planted is $(round(a*1000)/1000).")
println("The expected price is $(round((w'*max(p,1))[1]*1000)/1000).")
```

The first line imports the package for taking numerical expectations. The second line constructs a set of 10 *quadrature* nodes y and weights w for a normally distributed random variable with mean 1 and variance 0.1. Quadrature is a way to approximate the density of a continuous random

variable and is a convenient technique when taking expectations numerically. This code follows very closely to Newton's method but is a technique called function iteration. We begin with a guess of the acreage planted, in this case we begin with $a = 1$ to test if the price floor has any effect on equilibrium outcomes. We then approximate the potential distribution of prices using the 10 y quadrature nodes, conditional on our guess that the farmer plants 1 acre. This gives us a quadrature of prices at 10 nodes. We then calculate how many acres the farmer plants conditional on this approximated price distribution. We then see if how many acres the farmer plants is "close enough" to our guess. If not, we store the new value as our guess and continue iterating on the process. Eventually, the acreage planted will converge: the difference between successive iterations will become very small. This results in equilibrium acreage planted of 1.10 acres (rounded to three digits).

Q: Suppose we want to solve a 100, 1,000, or 10,000 equation system of linear equations (perhaps asymmetric firm first-order conditions and market clearing conditions). How do we solve for the equilibrium controls of this system?

We can represent a system of linear equations by $Ax = b$ where A is $n \times n$, x is the $n \times 1$ matrix of variables and b is $nx1$. We can solve for the equilibrium by inverting A,

$$x = A^{-1}b.$$

You will find out that matrix inversion is very costly: in big-O notation it's computational complexity is ($O(n^3)$) to invert an $n \times n$ matrix. This means that the time to invert the matrix increases *cubically* in the dimensionality of the matrix.[1] We can observe this by running the following code:

```
# Initialize matrices
A100 = rand(100,100)
A1000 = rand(1000,1000)
A10000 = rand(10000,10000)

@time invA100 = inv(A100)
@time invA1000 = inv(A1000)
@time invA10000 = inv(A10000)
```

The time necessary to invert the matrix increases rapidly as the number of arithmetic operations that must be performed grows cubically in the dimension of the matrix. Selecting an efficient al-

[1] Value function iteration, which you may have seen in a macro course, is even more costly and has exponential computational complexity: $O(m^n)$ for an n dimensional state space. We will discuss computational complexity later in this class.

gorithm is critical for large systems. The cubic computational complexity is actually a feature of the common Gauss-Jordan elimination algorithm. But with a different algorithm, Coppersmith-Winograd, this computational complexity can be reduced to $O(n^{2.376})$.

Q: Solve the following system of equations:

$$\mathbf{400\,x_1 - 201\,x_2 = 200}$$
$$\mathbf{-800\,x_1 + 401\,x_2 = -200}$$

This can be represented in matrix notation as a 2x2 matrix of coefficients, the 2x1 vector of unknowns, and the 2x1 vector of constants. By doing a simple matrix inversion we can solve the system and obtain $x_1 = -100, x_2 = -200$. But suppose we make a small change of the first equation so that it is instead: $\mathbf{401\,x_1 - 201\,x_2 = 200}$. Although we only slightly changed one coefficient, the solution to the system is now $x_1 = 40000, x_2 = 79800$! When small changes to the parameters of a system lead to drastic changes in the solution we call the system *ill-conditioned*. This is of large concern in numerical settings due to necessary rounding of numbers for storage in finite memory.

1.1 Big-O Notation

We will be using Big-O notation throughout the course to quantify the speed or accuracy of numerical methods.

1.1.1 Accuracy

Big-O notation allows us a convenient way to capture the precision of our approximations. You may be familiar with Big-O notation for Taylor expansions:

$$sin(x) \approx x - x^3/6 + x^5/120 + O(x^7)$$

for x near zero. $O(x^7)$ means that inside a neighborhood about zero, $|sin(x) - (x - x^3/6 + x^5/120)| \leq C\,x^7$ where C is some positive constant. The left hand side is your error in approximating $sin(x)$ with a Taylor expansion. So $O(x^7)$ means that there are additional terms (this is an infinite sum) of order higher than 7 that we are not capturing in our approximation to $sin(x)$. Since these terms are higher order than 7, and x is near zero, the other terms are small and $O(x^7)$ can tell us the *upper bound* on the growth rate of the error in our approximation as we move away from $x = 0$. So as we move from $x = 0$ to another point in the neighborhood of 0, the error in our sin Taylor approximation is growing no larger than x to the seventh power. For small x, having higher order

terms in the big-O implies a better approximation since the error will grow slower. You can test this out yourself. Clearly $sin(0) = 0$, and the Taylor approximation is also zero. If we move to $x = .001$, the fifth-order Taylor approximation is accurate to over 20 decimal places, but the third order taylor expansion is accurate to only 17. If we move to $x = 0.1$, then the fifth-order Taylor approximation is only accurate to one decimal place. As we move further away from 0, the error grows.

1.1.2 Speed

We also use Big-O notation to tell us how the upper bound of computation time increases as we input more data to the algorithm. For example, a function that takes an input, but always returns π regardless of what the inputs are is $O(1)$ since the algorithm always executes in the same amount of time regardless the size of the input data. Accessing a location in an array (the array is the input) is also $O(1)$ since it takes roughly the same amount of time to access a location no matter the size of the array. We say these algorithms run in *constant time*.

An function that scales linearly in data in $O(N)$. An example of this would be inserting an element into an arbitrary location of a vector. The larger the vector is, the more elements you have shift around in memory to accomodate the new element and the more operations are needed. Another example would be finding the largest element in a vector. We need to check every element, so doubling the size of the vector doubles the number of elements to check. This should double the amount of time required to complete the process. These algorithms run in *linear time*. $O(N^2)$ represents algorithmic performance that is proportional to the square of the size of the data. Looping over the elements of an array and applying an operation is $O(N)$, but doing it in a nested loop is $O(N^2)$. These algorithms run in *quadratic or polynomial time*. $O(N^3)$ denotes cubic time. Matrix multiplication and inversion algorithms can take cubic (or less) time. $2^{O(N)}$ denotes exponential growth in compute time given additional data. In future classes we will see that value function iteration with Chebyshev nodes is an algorithm that runs in *exponential time*. The travelling salesman problem is also an exponential time algorithm.

1.2 What can we compute?

From these examples, there clearly exist economic models which cannot be solved analytically. Many types of models used widely in research are computed in practice, including:

- Computable general equilibrium
- Linear quadratic models
- Real business cycles

- Stochastic rational expectations
- Dynamic programming for non-linear models
- Dynamic games (structural IO)
- Agent-based modeling

We can use computation alongside theory to answer *quantitative* questions. Standard theory is inherently qualitative, answering questions of whether policy is efficient or not. However theory does not always tell us how efficient: what are the dollar gains to policy. Perhaps theory shows us that carbon tax is more efficient than an intensity standard on electricity generation. But it does not tell us if the gains from switching to a carbon tax from an intensity standard are economically meaningful.

Theory often must rely on strong assumptions such as log utility, zero transactions costs, or perfect information. The real world does not necessarily display these features and it's unclear what we lose in terms of real world efficiency by employing strict assumptions. Strictly concave objective functions with linear transitions are often used in economic models to guarantee uniqueness of equilibrium. But if, for example, we were to model the climate-economy system with the objective of cost-efficiently limiting global warming, this assumption may not hold. Cost-minimization problems require a convex objective and convex Hamiltonian (dynamic equivalent of the Lagrangian). But temperature is a concave function of emissions which contradicts the usual assumptions used in economic theory. Using computation we can circumvent these issues and find quantitative results.

2 Computer arithmetic

2.1 Storage of numbers

Computers have finite memory and hard disk space and therefore have limitations on the storage of numbers. Rational numbers are stored as $\pm m\, 2^{\pm n}$, where m is called the mantissa or significand, and n is the exponent. The size of the numbers that can be represented by a computer are limited by how much space is allocated for a real number, typically 64 bits. Usually 53 are for the significand and 11 are for the exponent. Increasing the size of m allows for greater precision in representing a number while larger n allows for larger range of magnitudes. Irrational numbers such as π or $\sqrt{2}$ are also represented in this way and are therefore just approximations.

Limitations on storage suggest three facts. First, there exists a *machine epsilon* which denotes the smallest relative quantity that is representable by a computer. This is the smallest ϵ that a machine can distinguish $1 + \epsilon > 1 > 1 - \epsilon$.[2] Second there is *machine zero*, any quantity that is

[2] In Julia the machine epsilon is given by *eps*().

equivalent to zero on the computer. Third, there is *machine infinity*, the largest number that a computer can represent.[3] Because of these limitations, computer programs can run into overflow or underflow problems where an operation takes machine representable numbers and produces numbers that exceed machine infinity or are less than machine zero. These limitations are extremely important in computational methods and recognizing them can help avoid severe problems in coding.[4]

2.1.1 Error

Error comes into play through rounding and truncation. Computers are built to do integer arithmetic so when we want to represent non-integers, like irrational numbers, they must be rounded to the nearest number that is representable by the machine. This introduces errors into the system. The way to mitigate this error is to allocate more bits to representing numbers so that they can be more precise, larger, or smaller. Truncation occurs when a computer must represent an infinite sum or the limit of an infinite process. Consider the exponential function,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

This is an infinite sum of fractions of integers, but computers do not have infinite space to hold each fraction. Therefore computers must truncate the sum at some upper index $N < \infty$.

Rounding and truncation leads to errors in calculations which can propagate through the model and reduce accuracy of the final solution. Consider a quadratic equation,

$$x^2 - 26x + 1 - 0.$$

The solution to this is $x^* = 13 - \sqrt{168} = 0.0385186$. However $\sqrt{168}$ is an irrational number and therefore must be approximated by the computer. If the computer approximates it to five digits, 12.961 then the computer returns a solution of 0.039. This results in a relative error of over 1 percent between the true value and the approximated value by the five digit machine.[5] In general, avoid adding or subtracting numbers that different greatly in magnitude. If the magnitudes differ by more than the size of the exponent then the smaller number is treated as zero due to rounding. In our example, if $\sqrt{168}$ was replaced by -0.000001, the result may be 13.000001, but on the machine it gets rounded to five digits: 13.000.

In another example, consider two alternative statements:

[3] *realmin*() and *realmax*() give these two numbers in Julia.
[4] Consider situations where the optimal level of a control is less than machine zero due to the scale of the problem.
[5] For ways to deal with error propagation like in this example see Judd (1998, Chapter 1).

1. x = (1e-20 + 1) - 1
2. x = 1e-20 + (1-1)

The two statements are mathematically equivalent. However for the first statement, the expression inside the parentheses gets rounded to 1, resulting (incorrectly) in $x = 0$. However, the second statement results in the correct answer of $x = 1e{-}20$.

In a final example let us try to solve $x = 100000.2 - 100000.1$. The solution is clearly $x = 0.1$. However, evaluating this statement in Julia leads to a result of $x = 0.09999999999126885$. Why? This is due to how each number is represented by the machine. Neither is actually able to be precisely represented by the computer:

$$100000.1 \approx 8589935450993459 \times 2^{-33} = 100000.09999999997671693563461 30$$
$$100000.2 \approx 8589936309986918 \times 2^{-33} = 100000.19999999995343387126922 61$$

3 Differentiation, integration, and linear algebra

3.1 Differentiation

Derivatives are pervasive in economic analysis and critical to finding equilibrium. Analytically, the derivative of $f(x)$ is given by,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

If we replace h with $\frac{1}{t}$ this re-frames the derivative as an infinite limit in t, which as stated above, computers cannot handle.[6] We can approximate the derivative using *finite difference* methods which, instead of taking the limit to zero, replace h with a small number. The *one-sided finite difference* formula is given by:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

where h is some number we select. We have two opposing forces in selecting h. We want h to be as small as possible to best approximate the limit, but we cannot have h be too small without running into truncation issues in the numerator. Consider the following Julia code to calculate the derivative of $f(x) = x^2$ at $x = 2$:

```
# Finite-difference as a function of the difference
x_sq_deriv(h,x) = ((x+h)^2 - x^2)/h
```

[6] Often times optimization routines can be fed an analytic expression for a derivative or gradient which can significantly reduce runtime.

```
# Display several finite differences
println(x_sq_deriv(1.e-8,2.))
println(x_sq_deriv(1.e-12,2.))
println(x_sq_deriv(1.e-30,2.))
println(x_sq_deriv(1.e-1,2.))
```

The first one-sided finite difference yields the correct answer, $f'(2) = 4$ up to the 8th decimal point. However, the second finite difference returns a value of $f'(2) = 4.0004$, substantially more error because h is too small. In fact, if is sufficiently small as in the third finite difference, it actually returns a value of zero ($x + h \approx x$) due to truncation. Finally, the fourth finite difference demonstrates the result when h is too large and returns a value of $f'(2) = 4.1$. Miranda and Fackler (2002) suggest selecting $h = \max\{|x|, 1\}\sqrt{\epsilon}$ for one-sided approximations, where ϵ is machine epsilon. We can obtain an error bound on this approximation using first-order Taylor expansions:

$$f(x + h) = f(x) + f'(x)h + O(h^2),$$

where $O(h^2)$ means that all omitted terms in the expression can be expressed in terms of second or higher powers of h. We can re-arrange this to obtain,

$$f'(x) = \frac{f(x + h) - f(x)}{h} + O(h),$$

using the fact that $O(h^2)/h = O(h)$, yielding an approximation error of $O(h)$. In the limit as $h \to 0$, this is exact. Each time we halve h, we increase accuracy of our approximation by a factor of two.

An improvement on the one-sided finite difference is the *central finite difference*:

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}.$$

Using the central difference can often buy us additional significant digits in the resulting derivative approximation. This can be seen by taking two second-order Taylor expansions,

$$f(x + h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + O(h^3)$$
$$f(x - h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} + O(h^3).$$

Subtracting the two and dividing by $2h$ we arrive at,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

This is one more order accurate ($O(h^2)$ versus $O(h)$) than the one-sided finite difference since the error includes strictly higher order terms in the Taylor expansion compared to the one-sided finite difference. If we halve h we increase accuracy by a factor of four. Miranda and Fackler (2002) suggest selecting $h = \max\{|x|, 1\}\sqrt[3]{\epsilon}$ for central approximations. We can also use central finite difference methods to approximate higher order derivatives. First take third-order Taylor expansions,

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + O(h^4)$$
$$f(x-h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + O(h^4).$$

If we add the two expressions and then divide by h^2 we arrive at,

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).$$

Second derivatives are particularly important in checking for maxima or minima, and calculating the Hessian is one way to obtain standard errors when performing maximum likelihood estimation.[7] Generally the higher order derivatives are more difficult to approximately accurately because h gets raised to higher powers and truncation problems arise.

3.1.1 Automatic Differentiation

A problem with finite-differences is that we are being put between two opposing forcing. We want h to be small to have a more accurate derivative, but a smaller h is more likely to lead to computational error due to truncation and rounding (i.e. $x + h \to x$). An obvious alternative is to try to supply the analytic derivatives if possible. This yields an exact derivative, but for complicated problems this is highly likely to lead to programmer error. This may make it seem like there's no good solution to taking derivatives. However, intuitively, all computer programs are built upon arithmetic. Everything is just adding, subtracting, multiplying, dividing, exponentiating, etc. Applying the chain rule to each of these individual arithmetic operations is easy and simple. In fact, it turns out there exist methods to essentially represent your entire computer program as a giant chain rule, and then take the derivative of it with another computer package. This is called

[7] The CompEcon toolbox has scripts which compute Jacobians and Hessians of arbitrary functions.

automatic differentiation. This gets rid of the two drawbacks of finite differences and analytic derivatives. Automatic differentation yields an exact derivative, and it exploits the structure of computers and programming so it can do the derivative for you.

3.2 Integration

Numerical integration is a common problem in economics. Any time we must take expectations due to stochasticity in the system, or due to epistemic uncertainty, we must take integrals of the form $\int_D f(x)dx$ where $f : \mathbb{R}^n \to \mathbb{R}$ and $D \subset \mathbb{R}^n$. Why is integration problematic? Because, in a slight abuse of notation, integrals are effectively infinite sums. In a one dimensional example integrating $f(x)$ over the interval $[a, b]$ is essentially,

$$\lim_{dx \to 0} \sum_{i=0}^{(a-b)/dx_i} f(x_i) \times dx_i,$$

where dx_i is now some subset of $[a, b]$ and x_i is some evaluation point, e.g. the midpoint of dx_i.

Just like derivatives, we have trouble with infinity (here an infinite sum). To approximate integrals we will use *quadrature* formulas. These formulas (and all formulas) take a finite number of evaluations of the integrand and use a weighted sum of the evaluations to approximate the integral. We will look at several different quadrature methods varying in complexity and approximation quality.[8]

3.2.1 Newton-Cotes

Newton-Cotes quadrature methods are ones you have likely encountered before: *the trapezoid rule and Simpson's rule.* They are essentially just piecewise polynomials. Suppose we wish to integrate a unidimensional function over the interval $[a, b]$. The trapezoid rule splits the interval into subintervals of equal length and approximates the function with linear interpolation. In other words, it constructs trapezoids below the curve. The integral is approximated by summing the areas of the trapezoids.

Formally, define $x_i = a + (i - 1)h$ for $i = 1, 2, .., n$ where $h = \frac{b-a}{n-1}$. The x_i are the *nodes* of our approximation scheme and divide the $[a, b]$ interval into $n - 1$ subintervals of length h. For some subinterval $[x_i, x_{i+1}]$, we can approximate $f(x)$ by a line passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$. The area under this line is our approximation to the integral of $f(x)$ over this

[8] We will leave Monte Carlo methods until later in the class. But note that quadrature is generally quicker and more accurate than Monte Carlo.

subinterval:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

If we sum up the approximate areas for each subinterval we have the trapezoid rule:

$$\int_a^b \approx \sum_i^n w_i f(x_i)$$

where $w_1 = w_n = h/2$ and $w_i = h$ otherwise.

How accurate is the trapezoid rule? It is *first-order exact* because it can exactly compute any first-order polynomial (i.e. a line). Therefore, for a smooth function f, the trapezoid rule has error $O(h^2)$: the error shrinks quadratically with the size of the subintervals.

We can improve on the approximation quality of the trapezoid rule by using piecewise quadratic functions instead of piecewise linear functions, this is called *Simpson's rule*. Let x_i and h be the same as before, and let n be odd. For a *pair* of subintervals, $[x_{2j-1}, x_{2j}]$ and $[x_{2j}, x_{2j+1}]$, we can approximate $f(x)$ by the quadratic function that is uniquely identified by the three points $(x_{2j-1}, f(x_{2j-1})), (x_{2j}, f(x_{2j})), (x_{2j+1}, f(x_{2j+1}))$. The area under this quadratic function is our approximation to the integral of $f(x)$ over this pair of subintervals:

$$\int_{x_{2j-1}}^{x_{2j+1}} f(x)dx \approx \frac{h}{3}[f(x_{2j-1}) + 4f(x_{2j}) + f(x_{2j+1})]$$

Summing over all pairs of subintervals we have Simpson's rule:

$$\int_a^b \approx \sum_i^n w_i f(x_i)$$

where $w_1 = w_n = h/3$, otherwise $w_i = 4h/3$ if i is even and $w_i = 2h/3$ if i is odd. Simpson's rule is substantially more accurate than the trapezoid rule: it's third-order exact meaning it can perfectly approximate any cubic function.[9] This may seem strange but Simpson's rule is only quadratic *locally*. This means that the approximation error is $O(h^4)$, which falls two orders faster than the trapezoid rule.

[9] Why? Note that the local quadratic approximation is equal to the underlying, for now assumed to be cubic function, at the end points and mid points. The difference between the quadratic approximating function and the true cubic function over the interval defined by the end points is then another cubic function that goes to zero at the end points and mid point. It turns out that this new cubic function is odd with respect to the midpoint so integrating it over the first half of the interval is canceled out by the second half and the third-order error goes to zero.

3.2.2 Gaussian Rules

Newton-Cotes formulas choose the x_i quadrature nodes arbitrarily, typically such that they are evenly spaced. On the other hand, Gaussian quadrature rules select these nodes more efficiently while also relying on *weight functions* $w(x)$. Gaussian rules generally try to *exactly integrate* some finite dimensional collection of functions (i.e. polynomials up to some degree). For a given order of approximation n, the weights $w_1, ..., w_n$ and nodes $x_1, ..., x_n$ are chosen to satisfy $2n$ *moment-matching* conditions:

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

where I is the interval on the real line over which we are integrating, and $w(x)$ is a given weight function. Using the nodes and weights pinned down by the moment matching conditions, we can approximate the integral by using a weighted sum of the function at the prescribed nodes:

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

This algorithm is order $2n-1$ exact: it can exactly compute the integral of any polynomial of order $2n-1$. But what do we select for the weighting function? To integrate some function $f(x)$ we can use the identity weighting function $w(x) = 1$. This results in *Gauss-Legendre* quadrature. Note that Gauss-Legendre quadrature can approximate the integral of any continuous function arbitrarily well by increasing the number of nodes n. Often times in economics we want to compute exponentially discounted sums so we may wish to evaluate $\int_I f(x)e^{-x}dx$. Using the weighting function $w(x) = e^{-x}$ is *Gauss-Laguerre* quadrature. There exists several other forms of Gaussian quadrature for different weighting functions (normal pdfs, Chebyshev polynomials). The correct nodes and associated weights for these integral approximation schemes are easily available.

Most applicable to economics is the use of Gaussian quadrature for taking expectations:

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

Indeed, if the weight function is the probability density function of a random variable X, we can take expectations of $f(x)$. The way Gaussian quadrature works is by effectively discretizing X into mass points (nodes) and probabilities (weights), where the discrete distribution is for some other random variable $\bar{X}$. Given an approximation with n mass points, X and $\bar{X}$ have identical moments up to order $2n$. As $n \to \infty$, we have a continuum of mass points and recover the continuous pdf

for X.

3.3 Linear Algebra

Many computational economics problems break down into systems of linear equations. In addition, many non-linear economic models can be linearized. But computers can only store numbers with limited precision, which introduces error in solving these systems. Moreover, if other algorithms depend on solutions from numerical systems, the error can propagate and grow in size as the solution routine steps through the model. There are several common solution methods that typically yield accurate and quick results.

3.3.1 L-U Factorization

If matrix A in our system of linear equations $Ax = b$ satisfies certain conditions, we can easily solve the system. For example, if A is either lower or upper triangular, we can solve for x recursively using forward and backwards substitution techniques, respectively. Consider some lower triangular matrix. The first row is only non-zero on the diagonal so x_1 is easy to solve for. Row 2 is then a function of x_2 and the already solved for x_1 so we can solve for x_2. The algorithm continues proceeding in this way. For a lower triangular matrix, forward substitution yields solutions:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \, x_j \right), \quad \forall i$$

However most systems of equations will not be lower or upper triangular. L-U factorization is an algorithm to decompose A into the product of lower (row permutable) and upper triangular matrices so that we can use the substitution solutions. L-U involves two steps. First we must factor A into lower L, and upper U, triangular matrices using Gaussian Elimination. This is possible for any non-singular square matrix. Once we have obtained L and U, we have $(LU)x = b$. Using forward substitution we can solve the following system for y: $Ly = b$. Once we have obtained y, we know that $Ux = y$ can be solved using backwards substitution.

Why should we prefer L-U factorization over numerically inverting a matrix or just using Cramer's rule? It's much faster. L-U factorization requires $\frac{n^3}{3+n^2}$ operations to solve an nxn linear system. Computing the inverse of the matrix takes $n^3 + n^2$. Cramer's rule takes $(n+1)!$. Are these differences important? Yes! To solve a 10x10 system of equations, L-U factorization takes only 430 operations compared to 1100 and 40 million for matrix inversion and Cramer's rule. The difference in computing time will only grow as the system of equations grows larger. Use the following code to test the difference in time between the two. Writing solving the system by using the backslash

defaults to L-U factorization, while the *inv* command uses matrix inverses. Cramer's rule requires one to write their own code (increasing programmer time) and will be clearly slower than the other two methods since looping through each element x will be necessary.

```
A = rand(100,100)
# Regular inversion
@time inv1 = inv(A)
# L–U Factorization
@time inv2 = A\eye(100,100)
println("The maximum relative error is $(maximum((inv1-inv2)./inv1)).")
```

How do we perform the first step of L-U factorization? We use *Gaussian Elimination* (also known as row reduction) which is based on simple row operations: (1) swap two rows, (2) multiply a row by a non-zero scalar, (3) add a scalar multiple of a row to another. With these tools, we begin by initializing matrix (LU) as (IA). We then perform row operations until U is upper triangular which will leave L as a row permutable lower triangular matrix.[10] We can then use forward and backward substitution in the second step of L-U to solve for x.

3.3.2 Ill-Conditioned Systems

As demonstrated before, small perturbations in the parameters of the system, potentially from rounding error, can lead to wildly inaccurate results. We can demonstrate this analytically with a two equation system,

$$\begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where M is a large number. We can solve this system with one row operation: subtract $-M$ times the first row from the second. This yields the L-U factorization,

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -M & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 0 & M+1 \end{bmatrix}$$

Using the substitution methods we obtain that,

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} M/(M+1) \\ (M+2)/(M+1) \end{bmatrix}$$

[10] Row operations effectively preserve our system of equations while column operations do not.

Both variables are approximately 1 for large M. But remember, adding small numbers to large numbers cause problems numerically. Backwards substitution (when solving $Ux = y$ after we have solved $Ly = b$) first calculates x_2 since it is a *recursive* method. But if $M = 10000000000000000000$, the computer will return x_1 is equal to precisely 1.[11] This is clearly incorrect. When we then perform the second step of backwards induction, we solve for $x_1 = -M(1 - x_2)$. This equals **zero**, a relative error of 100%! Large errors like this often occur because diagonal elements are very small. *Pivoting*, the interchanging of rows so that diagonal elements are as large as possible, is a method for mitigating this error.

Unfortunately some systems are *ill-conditioned* and pivoting cannot save them. A matrix A is said to be ill-conditioned if a small perturbation in b yields a large change in x. One way to measure ill-conditioning in a matrix is the elasticity of the solution with respect to b,

$$\sup_{||\delta b||>0} \frac{||\delta x||/||x||}{||\delta b||/||b||}$$

which yields the percent change in x given a percentage point change in the magnitude of b. If this elasticity is large, then computer representations of the system of equations can lead to large errors due to rounding. We approximate the elasticity in practice by computing the *condition number*,[12]

$$\kappa = ||A|| \cdot ||A^{-1}||$$

which gives the least upper bound of the elasticity. κ is always larger than one and a rule of thumb is that for every order of magnitude, a significant digit is lost in the computation of x.

References

Judd, Kenneth L. (1998) *Numerical Methods in Economics*, Cambridge, MA: MIT Press.

Miranda, Mario J. and Paul L. Fackler (2002) *Applied Computational Economics and Finance*, Cambridge, MA: MIT Press.

[11] Try it yourself.

[12] Later we will use condition numbers to help us improve the conditioning of value function approximations.