Numerical Methods
Lecture 1: Basics

Zachary R. Stangebye

University of Notre Dame

Fall 2017

Traditional Trade-off: Language Choice

1. Ease of development: **Dynamic Languages**
   - Examples: Matlab, R, Python, Stata, etc.
   - Line-by-line execution in real-time
   - No variable declarations
   - User-friendly features built-in (packages, plotting, etc.)
2. Speed: **Static Languages**
   - Examples: Fortran, C, C++, Java
   - Packages (compiles) entire source code *before* executing
   - Painful user interface (gives .exe file)
   - Long development time

Recent Improvements

- **Julia** is making pretty successful attempts at eliminating trade-off
  - Novel Feature: *Just-in-Time (JIT) Compilation*
  - Runs dynamically *but compiles as it goes*
  - First time execution slow, but subsequent executions very fast
- Easy to develop like a dynamic language, but fast like a static one
- We'll use Julia in this course
  - Increasingly popular in profession
  - Still relatively new (constant updates)

Getting Started in Julia: 3 Ways

1. Julia terminal/shell/REPL
   - Most basic implementation: Runs on your machine line-by-line
   - "Read-Eval-Print-Loop"
2. JuliaBox portal
   - Runs like terminal, but in browser
   - On a cloud instead of your machine
3. Juno IDE (Integrated Development Environment)
   - Similar to Matlab
     - 3.1 Text/script editor
     - 3.2 Command line interface: REPL
     - 3.3 Workspace/plotting function/documentation/etc.
   - We'll use this most often

Storing Information

- Information stored in *variables*
- Variables have many different **types**
  1. Real numbers ("Float": 64-bit and 32-bit)
  2. Integers ("Int": 64-bit and 32-bit)
  3. Characters
  4. Strings (sequences of characters)
  5. Vectors
  6. Arrays
  7. Functions
  8. ...
- Assign variables with an equal sign e.g. $x = 2$
- Static languages: Must state variable type when declaring variable
- Dynamic languages: Figures it out on its own (Julia)

Storing Information: Arrays

- Be a bit careful. Array assignment initially treated as shared memory (pointers)
- More efficient, but be aware of it
- Let A be an array...
  - $B = A$ means they share memory. Changes in assigned values of A translate to B
    - Not true for mathematical operations on A
  - To avoid this, use $B = copy(A)$, which gives values but not memory
- Not an issue with scalars (Float64 and Int64)

Storing Information: Functions

- Defining functions: Several ways to do it in Julia
  1. In-line: $myfunc(x) = x^2$
     - $\rightarrow myfunc(2) = 4$
  2. In a block:
     function $myfunc(x)$
         return x^2
     end
     - $\rightarrow myfunc(2) = 4$

- JIT-compilation: After initial definition of a function
  - Julia compiles a new function each time you pass it different argument types
  - Alternative: Pre-specify argument types to boost speed, but can cause errors

Subroutines

- Julia functions in some instances can be interpreted as *subroutines*
  - Subroutines *modify* their arguments; functions leave them alone
  - Subroutines tend to be more memory-efficient, especially with larger arrays (don't create new variables)
- Julia functions will naturally modify their *array*-type arguments *if told to do so*
  - Won't modify scalars (Ints or Floats)
- Julia Nomenclature: Denote subroutines with a '!' after the function name and have it *return nothing* e.g.

function *myfunc*!(y, x)
    $y[1] = x^2$
    return *nothing*
end

Conditional Statements and Executions

- Important variable type: *Boolean*
  - Takes one of two types: *true* or *false*
  - Used to evaluate conditionals
- Two ways of defining them
  - Directly: $myTrueBoolean = true$
  - Condition: $myFalseBoolean = 1 == 2$
    - Sets *myFalseBoolean* to false, since 1 and 2 are not equal
- Multiple conditions
  - Use 'elseif' after first conditional
  - In-line: Use *and* i.e. & or &&
  - In-line: Use *or* i.e. | or ||

Conditional Statements and Executions

- 'If' statement used to run code sections based on boolean values
- Example:
  if *myTrueBoolean*
      println("It is true")
  else
      println("It is false")
  end
- Can also feed conditionals directly e.g. if $x <= 2$

Loops

- Often need to perform the same operation many times or access many different elements of an array
- Done in loops: Two kinds
  1. Definite: *for* loops
  2. Indefinite: *while* loops
- For loops execute an operation a pre-specified number of times
  - Can nest loops, use alternative indexing schemes, count backward, etc.
  - Tremendous flexibility
- Unlike Matlab, Julia does not punish you for writing in loops!
  - In fact, Julia runs loops *slightly faster* than array operations, since they tend to require less memory

While loops

- while loops execute indefinitely until a pre-specified condition is satisfied
- Format
  while *boolean condition*
      Do stuff...
      eventually boolean condition set to false...
  end
- Exercise caution: while loops most common way of stalling programs
  - If condition is never met, it will go on forever!

First Problem

- We have enough tools to start some application
- Problem 1: Market clearing in general equilibrium
  - Two consumers (i), two goods (j)
  - Normalize $p_1 = p$, $p_2 = (1 - p)$
  - Endowments: $e_j^i > 0$ for all agents
  - Consumption: $c_j^i \geq 0$ for all agents
- Perfect competition: Agents take prices as given and maximize utility
- Market clearing determines prices:

$$c_j^1 + c_j^2 = e_j^1 + e_j^2$$

for $j = 1, 2$

Demand Problem

- Agent i solves

$$\max_{c_1^i, c_2^i \geq 0} \; u_i(c_1^i, c_2^i)$$

$$\text{s.t. } pc_1^i + (1-p)c_2^i \leq pe_1^i + (1-p)e_2^i$$

- Solution implies demand function: $c_j^i(p)$
- Equilibrium price: $p^\star$ such that

$$c_j^1(p^\star) + c_j^2(p^\star) = e_j^1 + e_j^2$$

  for $j = 1, 2$
- Walras' law: If $p^\star$ clears market for good 1, then market for good 2 clears automatically

Equilibrium Price

- Suppose that preferences are Cobb-Douglas i.e.

$$u_i(c_1^i, c_2^i) = \beta_1^i \log(c_1^i) + \beta_2^i \log(c_2^i)$$

- Demand function is known analytically (Cobb-Douglas shares $\implies$ Expenditure shares)

$$c_j^i(p) = \frac{\beta_j^i}{\beta_1^i + \beta_2^i} \times \frac{pe_1^i + (1-p)e_2^i}{p_j}$$

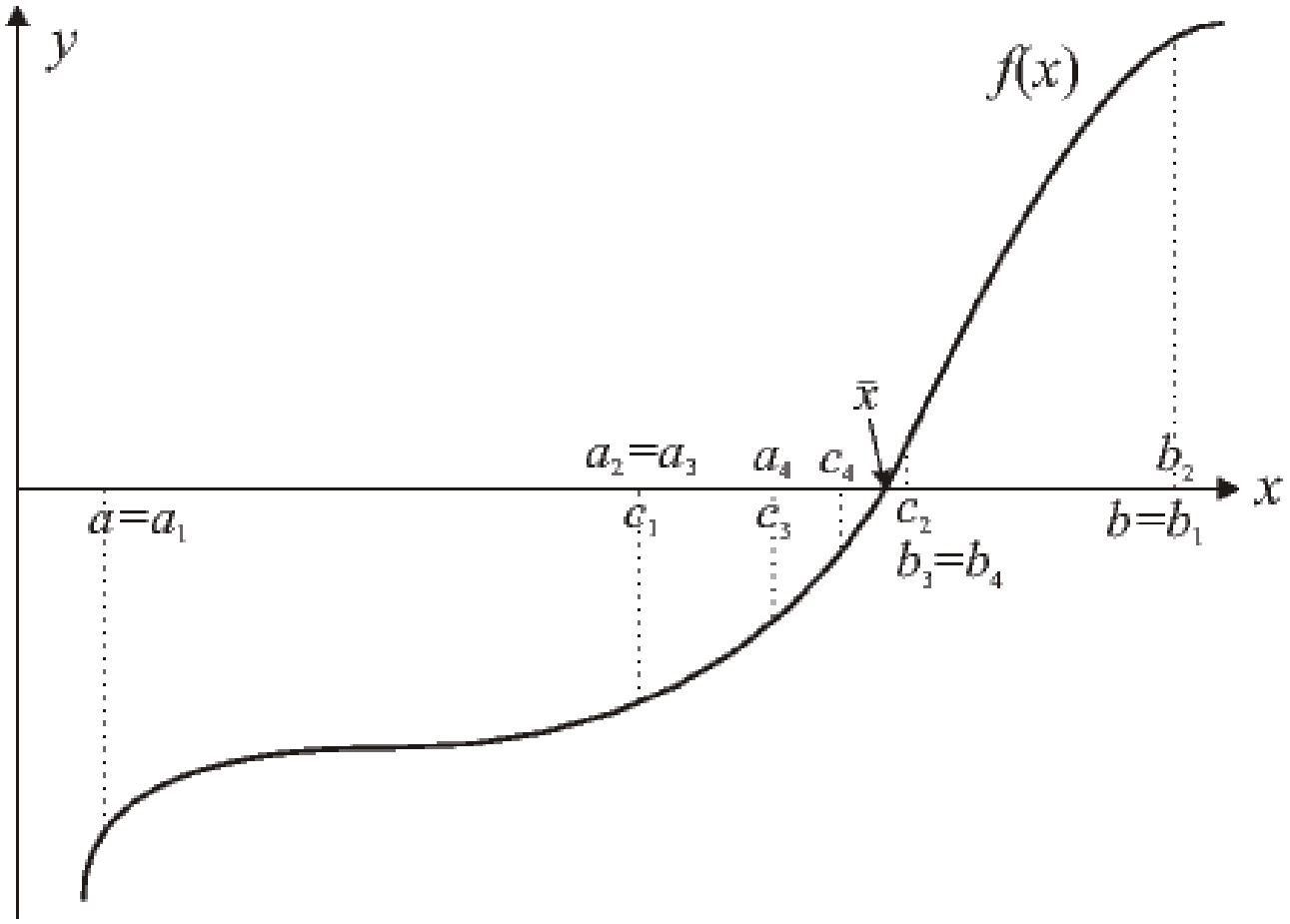
- Need to find eq'm price such that

$$c_2^1(p^\star) + c_2^2(p^\star) - (e_2^1 + e_2^2) = 0$$

Strategy: Interval Bisection

- Need to find the *zero* of a non-trivial, one-dimensional, continuous function e.g. $f(x) = 0$
- Suppose we know $\exists x_L, x_H$ such that $f(x_L) < 0$ and $f(x_H) > 0$
  - IVT tells us that $\exists x \in (x_L, x_H)$ such that $f(x) = 0$
- To find x, we iteratively split $[x_L, x_H]$ into smaller and smaller intervals until we find it
  1. Compute $y_{new} = f\left(\frac{x_L + x_H}{2}\right)$
  2. If $y_{new} > 0$, set $x_H = \frac{x_L + x_H}{2}$
  3. If $y_{new} < 0$, set $x_L = \frac{x_L + x_H}{2}$
  4. Repeat until either $|y_{new}|$ or $|x_H - x_L|$ sufficiently small
- Adjust algorithm slightly if function is decreasing instead (flip 2 and 3)

Interval Bisection in Practice



Implementation

- We can use interval bisection to get our market clearing price
- $f(p) = c_2^1(p^\star) + c_2^2(p^\star) - (e_2^1 + e_2^2)$
- Demand theory tells us
  1. $f(0) = \frac{\beta_2^1}{\beta_1^1+\beta_2^1} e_2^1 + \frac{\beta_2^2}{\beta_1^2+\beta_2^2} e_2^2 - (e_2^1 + e_2^2) < 0$
  2. $f(1) = \infty + \infty - (e_2^1 + e_2^2) > 0$ (this good is free)
- Set $p_L = 0$ and $p_H = 1$ and we can find eq'm price

Measuring Time

- Can use in-built clock to measure time
- Two lines sandwich code:
  1. start = time()
  2. elapsed = time()-start
- Delivers execution time between (1) and (2)

Storing Information: Memory

- Some variables do not need to be used all the time. Each variable has a **scope** of use
  - **Global** variables can be accessed by any entity, anywhere in your program
  - **Local** variables can only be accessed by the local environment (function, loop, etc.)
- Speed and memory tradeoffs in choosing variable types
  - Local variables preferred for each (sometimes vastly), but leads to more decentralized code
  - Local variables greatly simplify *machine level code*
- Dynamic languages generally assume everything is global
  - Some exceptions e.g. Matlab's separate-script functions
- Static languages generally assume everything is local
- Julia straddles a (sometimes uncomfortable) middle-ground

Variable Scope: Julia

- Where does Julia fall? JIT compilation $\implies$ in the middle
  - All variables defined in 'body' of a file are considered global
  - All variables defined in a structure e.g. function, loop, are considered local
- Global variables can be seen in the workspace following execution
- Can undo this with the variable types: 'local' and 'global'
  - Instead of $x = 3.0$, set *global* $x = 3.0$
- More details on scopes: https://docs.julialang.org/en/v1/manual/variables-and-scoping/index.html

Maximization Techniques

- Many problems don't have closed-form solutions
- Want to let the computer do the maximization (minimization) for you
- Consider simple problem from before
  - Know the solution: Can compare speed and accuracy of different methods

$$\max_{c_1^i, c_2^i \geq 0} \beta_1^i \log(c_1^i) + \beta_2^i \log(c_2^i)$$

$$\text{s.t. } p c_1^i + (1-p) c_2^i \leq p e_1^i + (1-p) e_2^i$$

$$\implies c_j^i(p) = \frac{\beta_j^i}{\beta_1^i + \beta_2^i} \times \frac{p e_1^i + (1-p) e_2^i}{p_j}$$

Approach 1: Grid-Search

1. Discretize options into a grid on feasible space
2. Evaluate objective at each point
3. Find largest value

- Start with substitution to get rid of constraint

$$U(c_1^i) = \beta_1^i \log(c_1^i) + \beta_2^i \log\left(\frac{p(e_1^i - c_1^i) + (1-p)e_2^i}{1-p}\right)$$

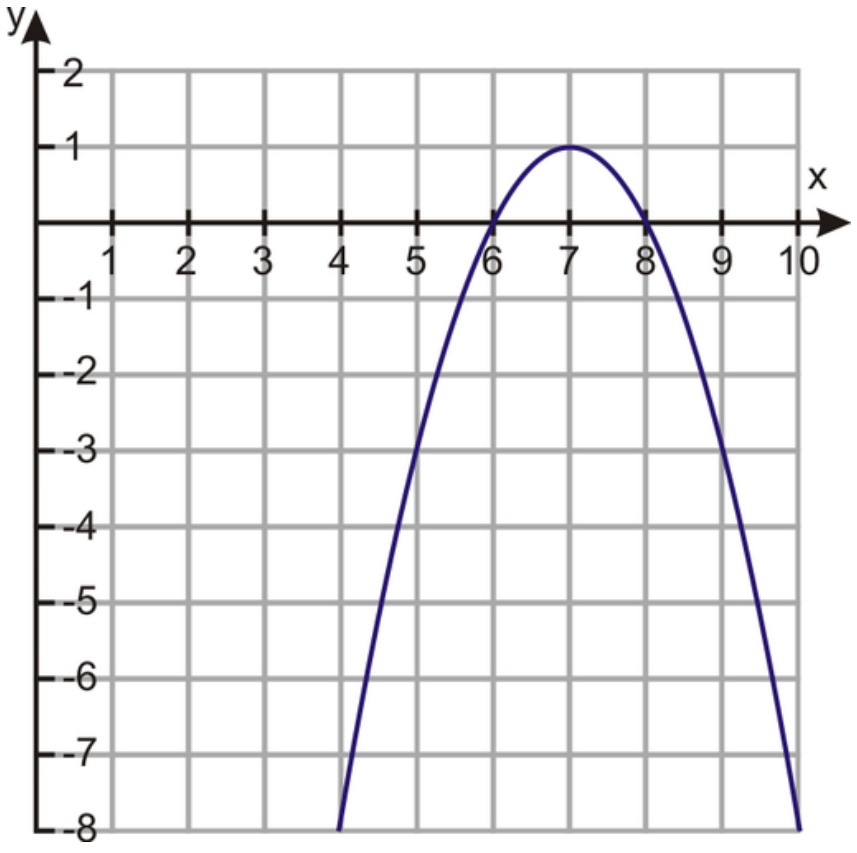
- Bounds: $c_1^i \in \left[0, \frac{pe_1^i + (1-p)e_2^i}{p}\right]$
- Break into N, equidistant gridpoints
  - Requires N evaluations of the function
  - Accuracy: Within $(\textit{upper bound} - \textit{lower bound})/(N-1)$

Variant on Approach 1

- Many benefits of grid-search
  - Finds a global max (regardless of curvature)
  - Super-simple/intuitive
- Drawbacks
  - Pretty slow: Time scales exponentially with dimensions (really bad)
  - Must evaluate function at every point
- If function *concave*: Alternative
  1. Evaluate gridpoints in ascending order
  2. Stop when objective begins to decrease (must be max)

  - Necessarily uses fewer evaluations than grid-search: Often substantial
  - In small applications, not always better than built-in max functions...

Modified Grid-Search



Need only evaluate points on grid less than 7

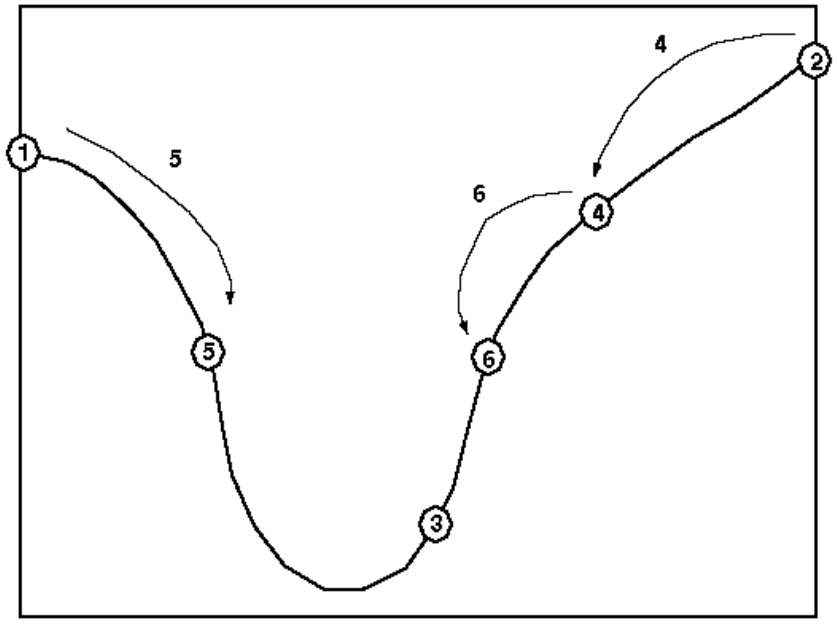
Approach 2: Golden-Search

- Alternative approach: Similar to interval bisection
- Idea: Want a max in between two points, $[a, b]$
  1. Check two *other* points in that range, $x_1 < x_2$
  2. If $f(x_1) < f(x_2)$, must be a max in range $[x_1, b]$
  3. If $f(x_2) \leq f(x_2)$, must be a max in range $[a, x_2]$
  4. Adjust range and pick two new points
  5. Continue until interval shrinks small enough
- How to pick the points? Two criteria. Want
  1. To use an old point. Each new iteration takes only *1* new function evaluation
  2. Distance to shrink by same fraction each time
- Unique set of points does this: Chosen by *Golden Ratio*

$$x_i = a + \alpha_i(b - a)$$

where $\alpha_1 = \frac{3-\sqrt{5}}{2}$ and $\alpha_2 = \frac{\sqrt{5}-1}{2}$

Golden-Search Example (Minimization)



Golden-Search

- Advantages
  - Pretty fast and accurate
  - If tolerance is $\epsilon > 0$, converges in

$$N = int\left(\frac{\log(\epsilon) - \log(\alpha_1\alpha_2[b-a])}{\log(\alpha_2)}\right)$$

  - Example: If $a = 0$, $b = 1$, and $\epsilon = 1e-4$, then $N = 17$
    - Can increase precision by 100x ($\epsilon = 1e-6$) with only 9 more evaluations i.e. $N = 26$
- Drawbacks
  - Only finds *local* maxima. Could miss global optimum

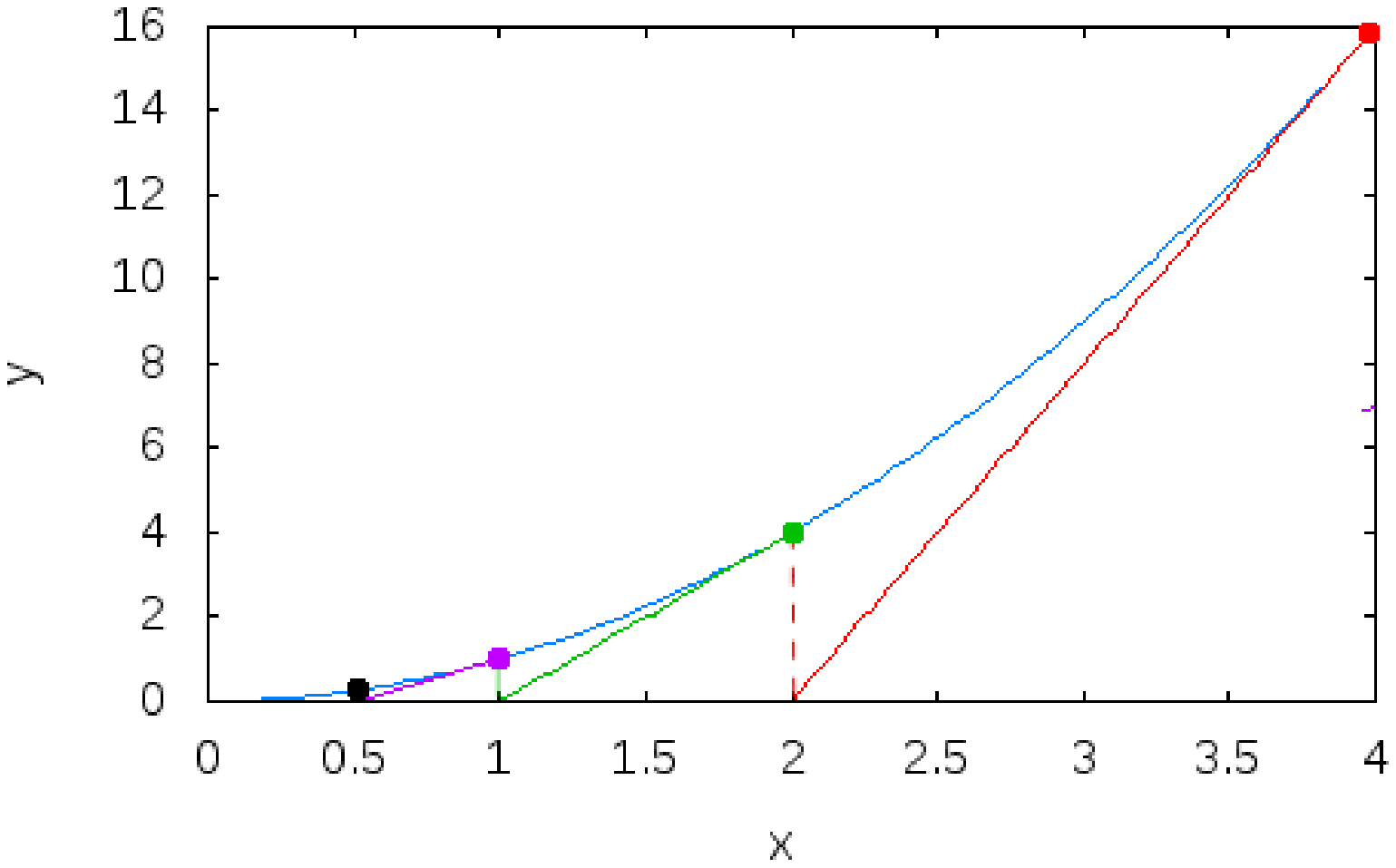
Digression: Plotting in Julia

- Several different *packages* in Julia that allow you to see plots in real time or save them as PDFs/PNGs
- Packages give Julia extra functionality and are easy to install:
  1. ']' + add PackageName [done only once in terminal, ever]
  2. using PackageName [done every time you run your code (include in script)]
- Package "Plots" enables basic plot functionality
  - Basic Syntax: plot(x, y, option1=option1set, option2=option2set, ...)
  - Details/specifics in lecture code
- We'll use a bunch of extra packages throughout the course

Approach 3: Newton-Raphson Method

- Based on Newton's root-finding approach: $y = f(x)$
- Tangent line at x_k: $y = f'(x_k)(x - x_k) + f(x_k)$
  - Idea: Each update x_{k+1} is zero of the current tangent line
- To find $f(x) = 0$...
  1. Evaluate $f(x_k)$; stop if close to zero
  2. If not, evaluate $f'(x_k)$
  3. Updated guess: $x_{k+1} = x_k - f(x_k)/f'(x_k)$
- Tends to work pretty well on well-behaved problems (though there are pathological cases)
- Converges quadratically in the neighborhood of the true zero

Newton's Method



Newton-Raphson Method

- Useful on its own (rootfinding sometimes necessary in numerical analysis)
- Can be used for optimization: Set $f'(x) = 0$
- The Newton-Raphson Method
  1. Evaluate $f'(x_k)$; stop if close to zero
  2. If not, evaluate $f''(x_k)$
  3. Updated guess: $x_{k+1} = x_k - f'(x_k)/f''(x_k)$
- Pros:
  - Converges pretty quickly (locally quadratically)
  - Works well in practice
- Cons:
  - Doesn't always handle broad sets of constraints well
  - Only finds extrema: Could be local max (or a local min!)
  - Often numerical derivatives needed
  - Sensitive to initial conditions

Newton-Raphson in Practice

- Recall our objective:

$$U(c_1^i) = \beta_1^i \log(c_1^i) + \beta_2^i \log\left(\frac{p(e_1^i - c_1^i) + (1-p)e_2^i}{1-p}\right)$$

- First derivative:

$$U'(c_1^i) = \frac{\beta_1^i}{c_1^i} - \frac{\beta_2^i \frac{p}{1-p}}{\frac{p(e_1^i - c_1^i) + (1-p)e_2^i}{1-p}}$$

- Second derivative:

$$U''(c_1^i) = -\frac{\beta_1^i}{(c_1^i)^2} - \frac{\beta_2^i \left(\frac{p}{1-p}\right)^2}{\left(\frac{p(e_1^i - c_1^i) + (1-p)e_2^i}{1-p}\right)^2}$$

Alternative First-Order Approach

- Use interval bisection to find the root of $f'(x^\star) = 0$
- Requires $f'(x_L) > 0$ and $f'(x_H) < 0$ (or vice versa)
  - Need not evaluate second derivative, so less error-prone
  - Generally does not converge as quickly
  - Suffers from same key flaw as Newton-Raphson
    - Finds only extrema (could be a local min!)
  - Handles bounds better (naturally)

Numerical Differentiation

- What if you cannot express $f'(\cdot)$ analytically?
- Approximate it numerically: Choose a small step-size, $h > 0$
- Several ways
  1. Forward difference

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

  2. Reverse difference

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

  3. Average

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

- Average tends to work best for most applications

Numerical Second Derivatives

- Second derivatives a little trickier...several valid ways but most versatile is generally the following average

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

- Second derivatives notoriously bad for complicated problems...
  - Example: Numerical Hessians generally not symmetric (mathematically they must be)
  - Error introduced by approximation large enough to throw off Hessian value

Implementation

- Step size choice? Tricky...depends on your problem
  - Standard is usually a little less than your optimization tolerance e.g. 1e-4 when tolerance is 1e-6
    - Too large and your approximations are no good (away from Taylor neighborhood)
    - Too small and limits of computer memory become an issue

Autodifferentiation in Julia

- JIT compilation works really well (for technical reasons) for *automatic*, *analytic* differentiation
- The ForwardDiff package (among others) implements this
- Can compute analytically for compositions of built-in functions
  1. Any order of derivative
  2. Gradient
  3. Hessian
- Extremely helpful!
  1. Many non-linear solvers/algorithms require gradient estimates
  2. Numerical derivatives (especially Hessians) often unreliable
  3. Gradients by hand for large problems both slow and error-prone
- Limitations: Can handle arrays (and even splines), but not loops
  - Limited functionality for complicated expectations, GMM, etc.

Saving your Work

- Sometimes you will want to save your work for later analysis (or analysis in another program)
- Best way to do this: JLD package (JuliaData)
  1. Pkg.add("JLD")
  2. using JLD
- Save variables: @save "filename.jld" x y z ...
- Load variables: @load "filename.jld"

Adding Constraints

- Substitute as much as possible to reduce dimension space
- In one dimension, pretty easy
- Only one type: Bounds
  - Built in to many of our algorithms
  - Newton's method doesn't always do so well with bounds
- Several intervals: Solve for each interval and take max across them
- Several dimensions: Other methods required

Built-in Packages

- Julia has some nice built-in optimization packages
- Three in particular are useful
  1. Roots package
     - Finds zeros: $soln = fzero(function, LB, UB)$
  2. Optim package (minimizer)
     - $result = Optim.optimize(function, LB, UB)$
     - Really fast
     - Basic algorithms (can choose as an option)
     - Only basic, box-constraints
  3. JuMP package (with the NLOpt solver)
     - A bit slower
     - More advanced algorithms
     - Can handle any constraints
- Generally faster than writing yourself: Exploit advanced machine-level code

Multivariate Optimization

- Things get more complicated in higher dimensions...
  - Grid-search suffers from 'curse of dimensionality'
    - K gridpoints along each of N dimensions $\rightarrow K^N$ gridpoints
    - Gets really big really fast!
- Variations on Newton-Raphson and Nelder-Mead tend to be most popular approaches
  1. Newton-Raphson $\rightarrow$ LFBG-S/Line-Search
  2. Nelder-Mead/PatternSearch (more on specifics later)
- Other popular option: *Genetic* and *Simulated Annealing* Algorithms
  - Both exploit stochastic search to find global optima
- Will not go into algorithmic details (yet)

Nonlinear Programming with JuMP and NLOpt

1. Construct a 'model' i.e. a constrained maximization problem
    - Specify the solver (NLOpt) and/or the algorithm
2. Add choice variables (and initial values)
3. Add constraints (as many as you want)
4. Add an objective function
    - Dictate whether min or max
5. "Solve" model
6. Pull out results

- Alternative solvers can be used if you have them

Nonlinear Programming with JuMP and NLOpt

- NLOpt on its own (without JuMP) has a similar functionality
- Useful Algorithms
  1. LD_MMA
     - Simple, gradient-based multivariate optimization with constraints
     - Uses autodifferentiation; be careful
  2. LN_COBYLA
     - Gradient-free ("PatternSearch"), but a bit slower
     - Details later in course
- More available at http://ab-initio.mit.edu/wiki/index.php/NLopt_Algorithms

Random Variables

- Random variables pop up all the time in economics
- Most useful ones do not have closed-form PDFs...
- Distributions package provides those (and lots of other stuff)
- Can define random variables *as an object*
  1. Get PDF/CDF at any point
  2. Draw from it
  3. Compute moments, quantiles, correlations, entropy, likelihood, etc.
  4. Multivariate distributions allowed
- Can truncate with the Truncate package
- Can even do MLE estimation to attain parameters for a distribution type

RVs and Computers

- Computers get along great with discrete distributions
  - e.g. Easy to compute moments with the pmf

$$E[m(\tilde{X})] = \sum_{i=1}^{N} m(X_i)p(X_i)$$

- Often need to tweak things a bit for continuous...
  - Cannot evaluate (even if pdf is analytic)

$$E[m(\tilde{X})] = \int_{x_L}^{x_H} m(X)f(X)dX$$

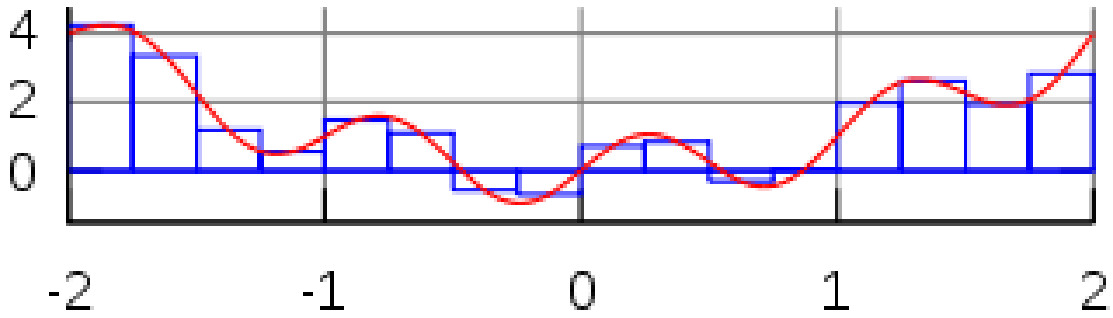
- This is one of the *the* most common operations in most models
  - Need a way to approximate it

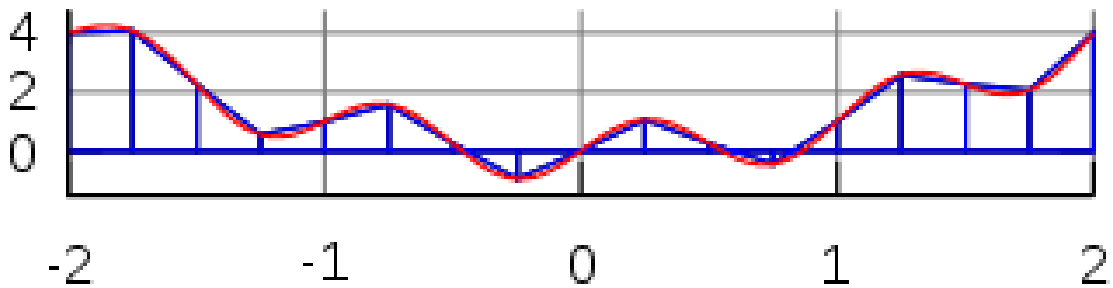
Expectations Method 1: Quadrature

- A **Quadrature** is a set of discrete weights meant to approximate a continuous pdf
- Idea: $\int_{X_L}^{X_H} m(X)f(X)dX \approx \sum_{i=1}^{N} w_i m(x_i)f(x_i)$
  - w_i is a weight and x_i is some sort of representative point
- Basic Calculus Method:
  - Split into N intervals $\{X_0, X_1, \ldots, X_N\}$ and define $g(x) = m(x)f(x)$

1. Rectangle Rule:
   $\int_{X_L}^{X_H} m(x)f(X)dX \approx \sum_{i=1}^{N} (X_i - X_{i-1})\, g\left(\frac{X_i + X_{i-1}}{2}\right)$
2. Trapezoidal Rule:
   $\int_{X_L}^{X_H} m(x)f(X)dX \approx \sum_{i=1}^{N} (X_i - X_{i-1}) \left(\frac{g(X_i) + g(X_{i-1})}{2}\right)$

Expectations Method 1: Quadrature





Newton-Cotes Quadrature

- Newton-Cotes: Approximate $g(x) = m(x)f(x)$ with Lagrange polynomials
  - Integrate over the polynomials (integral known)
  - Pick n evenly spaced points over interval $[x_1, x_n]$: $\{x_1, x_2, \ldots, x_n\}$: Yields approximating polynomial

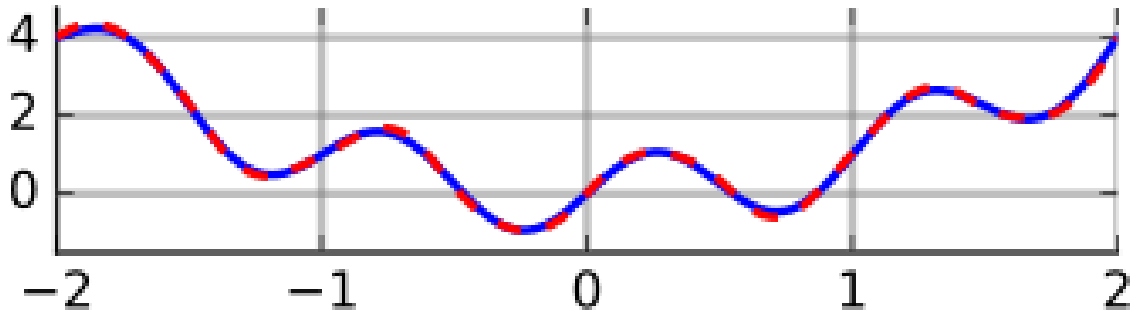
$$h_N(x) = \sum_{i=1}^{N} g(x_i) \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

- Evaluating integral yields closed-form approximations e.g. $N = 5$ and $t = x_i - x_{i-1}$

$$\int_{x_1}^{x_5} g(x)dx \approx \frac{2}{45} t(7g(x_1)+32g(x_2)+12g(x_3)+32g(x_4)+7g(x_5))$$

Newton-Cotes Quadrature

- Can subdivide into intervals *before we begin*
  - Tends to provide more accurate estimates
  - e.g. subdivided 4-point Cubic Quadrature aka "Simpson's 3/8 Rule"



- Some known approximation error: Shrinks with N
- Long list of approximation formulas (and errors) at http://mathworld.wolfram.com/Newton-CotesFormulas.html

Gaussian Quadrature

- Most accurate numerical integral approximation
- Similar to Newton-Cotes, but x_i endogenous
  - m-point approximation fits *exactly* any polynomial up to degree $2m-1$
- Start on interval $[-1,1]$: Idea, find weights/points such that

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

  then

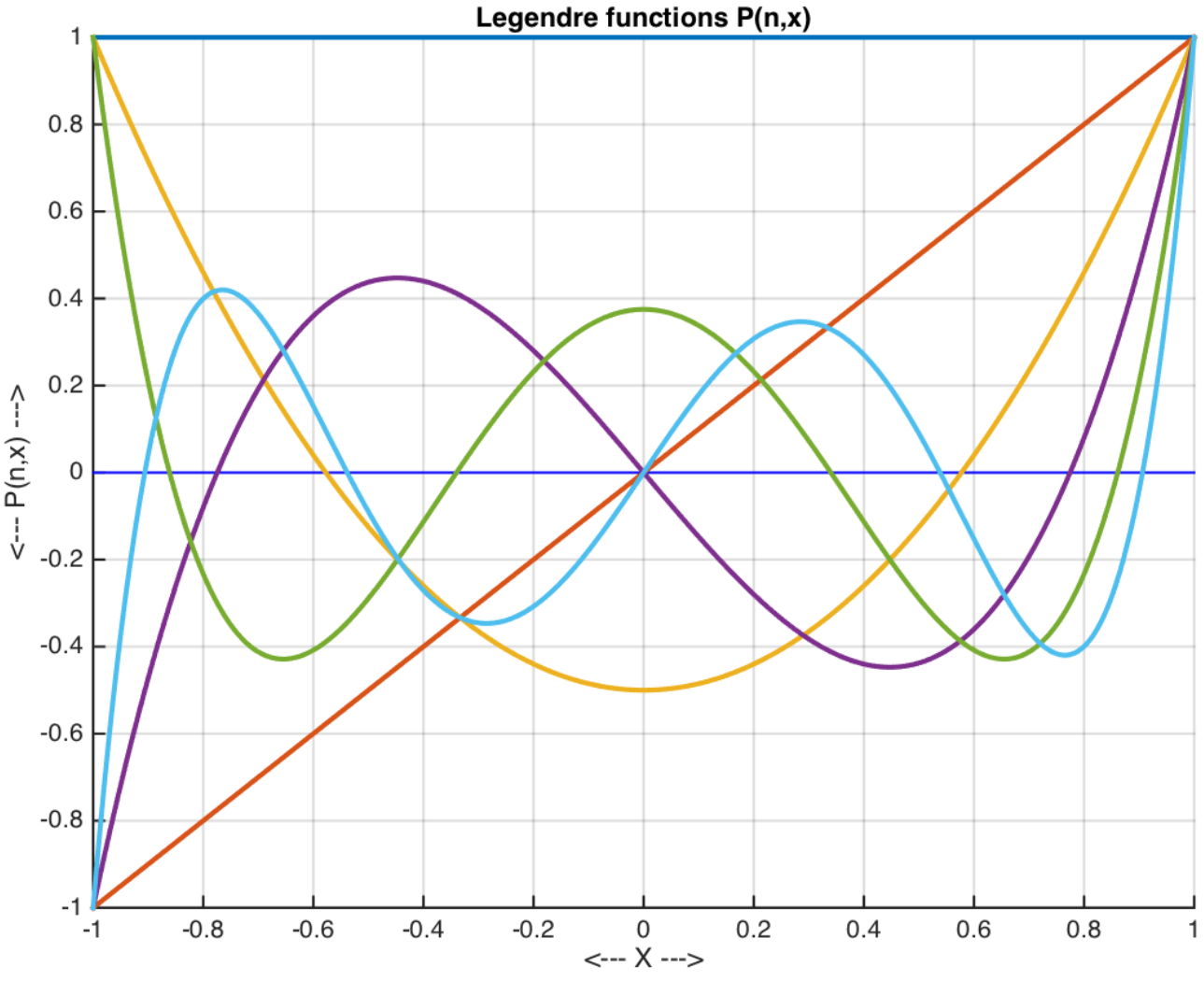
$$\int_{-1}^{1} g(x)w(x)dx \approx \sum_{i=1}^{N} w_i g(x_i)$$

- $w(x) = 1 \implies x_i$ are roots of *Legendre Polynomials*

Gaussian Quadrature: Under the Hood

- Constructs a (relatively) low-order polynomial approximation of your function
- Linear combination of an orthogonal class of polynomials
- Polynomial class determined by weighting function
  - $w(x) = 1 \implies$ Legendre
  - $w(x) = \frac{1}{\sqrt{1-x^2}} \implies$ Chebyshev
  - $w(x) = (1-x)^{\alpha}(1+x)^{\beta}, \alpha, \beta > -1 \implies$ Jacobi
  - $w(x) = e^{-x^2} \implies$ Hermite
  - ...
- Evaluates integral of approximate polynomial exactly

Legendre Polynomials



Gauss-Legendre Quadrature

- When $w(x) = 1$, this implies

N	x_i	w_i
2	$+/-\sqrt{1/3}$	(1, 1)
3	$0, +/-\sqrt{3/5}$	(8/9, 5/9, 5/9)
4	$+/-\frac{1}{35}\sqrt{525-70\sqrt{30}}$	$\frac{1}{36}(18+\sqrt{30})$
	$+/-\frac{1}{35}\sqrt{525+70\sqrt{30}}$	$\frac{1}{36}(18-\sqrt{30})$
...	...	...

- Fun fact: Weights always sum to 2 for any N

More weights at
http://mathworld.wolfram.com/Legendre-GaussQuadrature.html

Alternative Weights

- If we want to use a different set of weights than $w(x) = 1$
  1. Change of variables for function

$$\int_{-1}^{1} f(x)dx = \int_{-1}^{1} \hat{f}(x)w(x)dx$$

     where $\hat{f}(x) = f(x)/w(x)$
  2. Apply method to $\hat{f}$
- Some weighting schemes are more efficient that Gauss-Legendre

Gaussian Quadrature: Translation

- Most problems don't lie in $[-1, 1]$
- Shift problem and work on translated one

$$\int_a^b g(x)dx = \frac{b-a}{2} \int_{-1}^{1} g\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) dx$$

- Alternative polynomial sets and weights:
  1. $w(x) = \frac{1}{\sqrt{1-x^2}} \implies x_i$ roots of Chebyshev Polynomials
  2. $w(x) = (1-x)^\alpha (1+x)^\beta, \alpha, \beta > -1 \implies x_i$ roots of Jacobi Polynomials
  3. $w(x) = e^{-x^2} \implies x_i$ roots of Hermite Polynomials

Approach 2: Monte Carlo Methods

- Alternative approach: Exploit law of large numbers (LLN)
- Suppose we want to compute $E_f[g(\tilde{x})] = \int_{x_L}^{x_H} g(x)f(x)dx$
  1. iid sample from $f(\cdot)$: $\{x_1, x_2, \ldots, x_N\}$
  2. Law of large numbers says (under regularity conditions):

$$plim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} g(x_i) = E_f[g(\tilde{x})]$$

- Can set the seed with *srand* for consistency
- Advantages
  - Tends to be fast, closer to truth when N large
  - More effective when $Var(\tilde{x})$ is low (Central Limit Theorem)
- Drawbacks
  - Random answers. Can fix this with a fixed seed
  - Computers don't always generate random numbers perfectly...
  - **Can take up a lot of memory if many expectations required**

General Integrals with MC Methods

- What if it's not an expectation? Twist it until it is
- If it's bounded...make it look like a transformation of a uniform

$$\int_{x_L}^{x_H} g(x)dx = [x_H - x_L] \int_{x_L}^{x_H} g(x) \frac{1}{x_H - x_L} dx$$

  1. Draw a sample of size N from a uniform random on $[x_L, x_H]$
  2. Estimate $= [x_H - x_L] \frac{1}{N} \sum_{i=1}^{N} g(x_i) \rightarrow \int_{x_L}^{x_H} g(x)dx$

- If it's unbounded, can set high bounds
  - Drawback: Large bounds $\Longrightarrow$ High variance $\Longrightarrow$ Slow convergence

Markov Chains

- One more method very common in economics, but specific to *Markov Chains*
- A **Markov Chain** is a sequence of random variables $\{X_t\}_{t=-\infty}^{\infty}$ such that
  1. $Pr(X_t|X_{t-1}, X_{t-2}, \dots)$ is time-invariant (stationary)
  2. $Pr(X_t|X_{t-1}, X_{t-2}, \dots) = Pr(X_t|X_{t-1}, X_{t-2}, \dots, X_{t-n})$ for some $n < \infty$
- Canonical example: AR(1) process

$$z_t = \rho_z z_{t-1} + \epsilon_t$$

- Markov Chains appear *all the time* in dynamic stochastic models

Discrete Markov Chains

- In a discrete Markov Chain, $X_t \in \mathcal{X}$, where $|\mathcal{X}| = N < \infty$
- If $Pr(X_t|X_{t-1}, X_{t-2}, \dots) = Pr(X_t|X_{t-1})$, we can construct an $N \times N$ **Transition Matrix**, P
  - $P_{ij} = Pr(X_t = x_j|X_{t-1} = x_i)$
  - Notice $sum(P_{i:}) = 1$ for any row i
- Transition matrix *completely characterizes* the Markov Chain
  - Sometimes called a **Stochastic Matrix** as well
- Useful: Very easy to put into a computer!

Discrete Markov Chains: Marginal Distributions

- Current marginal distribution of X_t: $\psi_t \in \Delta_N$ i.e.
  1. $\psi_t \in \mathcal{R}_N^+$: $\psi_t(i) = Pr(X_t = x_i)$
  2. $\sum_{i=1}^{N} \psi_t(i) = 1$
- Can use transition matrix to compute ψ_{t+1}

$$\psi_{t+1}(j) = Pr(X_{t+1} = x_j) = \sum_{i=1}^{N} Pr(X_{t+1} = x_j | X_t = x_i) Pr(X_t = x_i)$$

$$\implies \psi_{t+1}(j) = \underbrace{\psi_t'}_{1\times N} \times \underbrace{P_{:j}}_{N\times 1}$$

$$\implies \psi_{t+1}' = \psi_t' \times P$$

$$\implies \psi_{t+1} = P'\psi_t$$

Discrete Markov Chains: Limiting Properties

- Repeated iteration can give distant marginals

$$\psi_{t+m} = (P')^m \psi_t$$

- Two states, $x, y \in \mathcal{X}$ are said to **communicate** if there exist positive integers k and j such that

$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

  i.e. there is some non-zero chance of eventually transitioning back and forth
- A Markov Chain is **irreducible** if all states communicate
- A Markov Chain is **aperiodic** if it does not cycle predictably

Discrete Markov Chains: Stationary Distributions

- A **Stationary Distribution** is a marginal distribution, $\psi^\star \in \Delta_N$, such that

$$\psi^\star = P'\psi^\star$$

Proposition

Every transition matrix implies a stationary distribution.

Proposition

If P is both aperiodic and irreducible, then

1. *$\psi^\star$ is unique*
2. *From any $\psi_0 \in \Delta_N$, $\lim_{t\to\infty} ||(P')^t\psi_0 - \psi^\star|| = 0$*

Computing the Stationary Distribution

- Suppose we have P. How can we compute $\psi^\star$? Two ways

1. Repeated application (approximate solution)
   - $\psi^\star \approx (P')^k \psi_0$ for any $\psi_0 \in \Delta_N$ and large k
2. Eigenvector approach (exact)
   - An **eigenvector** of a matrix A is a vector, v such that $Av = \lambda v$ for some scalar λ (which is called the corresponding eigenvalue)
   - We know $[I - P']\psi^\star = 0$
   - Compute the eigenvector corresponding to the zero eigenvalue of $I - P'$
   - Scale it such that it's in Δ_N

Discrete Markov Chains: Ergodicity

- In a Markov Chain, **Ergodicity** is the equivalent of the law of large numbers

Proposition

If P is irreducible, then for any $x_i \in \mathcal{X}$,
$plim_{N\to\infty} \frac{1}{N} \sum_{t=1}^{N} \mathbf{1}\{X_t = x_i\} = \psi^\star(i)$

- i.e. over a long simulation, the fraction of time spent in state x_i is the same as its weight in the stationary distribution.

Discrete Markov Chains: Application

- Suppose we have a hand-to-mouth consumer, $C_t = Y_t$
  - $Y_t \in \mathcal{Y}$, where $|\mathcal{Y}| = N < \infty$
  - Y_t transition matrix, P, is both aperiodic and irreducible
  - Time-discount-rate $\beta < 1$ and flow utility, $u(\cdot)$
- Want to compute expected utility:
  $U(y) = E_0[\sum_{t=0}^{\infty} \beta^t u(\tilde{C}_t) | Y_0 = y_i]$
  1. Define a vector, $u = u(\mathcal{Y}) \in \mathcal{R}^N$
  2. Know $\psi_0(y_i)$: $\psi_0(y_i)_i = 1$ and $\psi_0(y_i)_j = 0$ for all $j \neq i$
  3. Rewrite: $U(y_i) = \psi_0(y_i)'u + \beta\psi_1(y_i)'u + \beta^2\psi_2(y_i)'u + \ldots$

$$= \psi_0(y_i)'[I + \beta P + (\beta P)^2 + (\beta P)^3 + \ldots]u$$

$$\implies U(y_i) = \psi_0(y_i)'[I - \beta P]^{-1}u$$

- Last line follows since geometric series works for matrices too!

Continuous Markov Chains

- Similar idea, but transition probabilities no longer a matrix
  - $X_t \in \mathcal{S}$, which is infinite-dimensional
  - A **Stochastic Kernel** is a function, $p : S \times S \to \mathcal{R}^+$, such that for any $x \in \mathcal{S}$, $\int p(x|y)dy = 1$
  - Stationary distribution: Marginal density $\psi^\star$ such that

$$\psi^\star(y) = \int p(x|y)\psi^\star(x)dx \quad \forall y \in \mathcal{S}$$

- Examples
  1. AR(1) process

$$z_t = \rho_z z_{t-1} + \epsilon_t$$

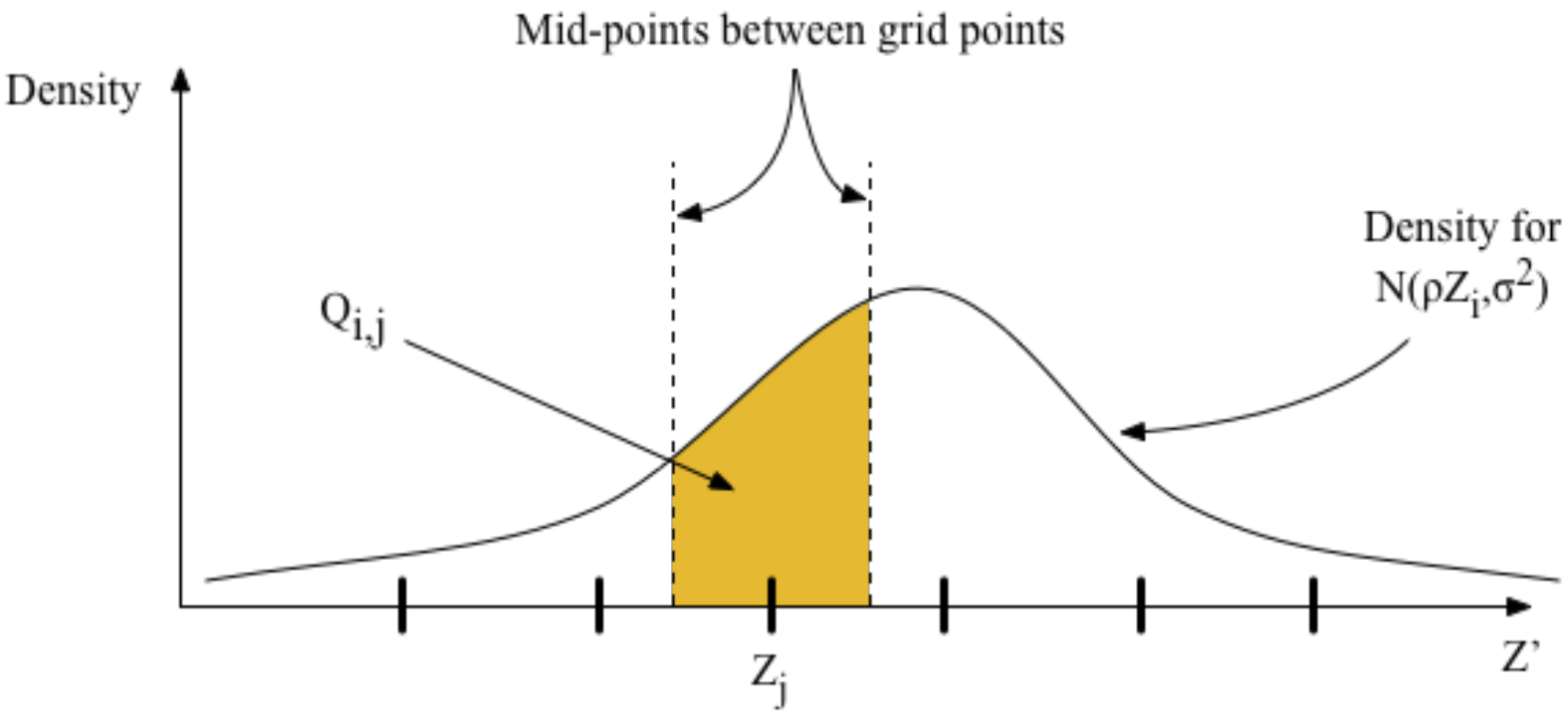
  2. Solow model with production shocks (A_{t+1})

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t$$

Numerical Integration: Approach 3

- Continuous Markov Chains most common type of numerical integration problem
- Can easily use our former strategies, but there's a new one too
- Intuitive Strategy: **Tauchenize** the process i.e. approximate it with a discrete Markov Chain of size N
  - Attributed to Tauchen (1986)
- Vast majority of numerical work continues to Tauchenize (even though not the most efficient way)
  - Rouwenhurst method also works and tends to be more efficient (see Kopecky and Suen [2010] for details)

Tauchenizing: Intuitively



Tauchenize an AR(1)

$$z_t = (1-\rho_z)\mu_z + \rho_z z_{t-1} + \sigma_\epsilon \epsilon_t$$

- ϵ_t is iid standard normal; μ_z is unconditional/ergodic mean
- Denote ergodic stdev by $\sigma_z = \sigma_\epsilon / \sqrt{1-\rho_z^2}$

1. Define an N-sized grid over z: $z^1 = \mu_z - m\sigma_z$, $z^N = \mu_z + m\sigma_z$; space other points evenly in between
2. Define a transition matrix, P, as follows for every row i
   - 2.1 $P_{i1} = \Phi\left(\frac{z_1 + w/2 - (1-\rho_z)\mu_z - \rho_z z_i}{\sigma_\epsilon}\right)$
   - 2.2 For $j \in \{2, \ldots, N-1\}$, $P_{i1} = \Phi\left(\frac{z_j + w/2 - (1-\rho_z)\mu_z - \rho_z z_i}{\sigma_\epsilon}\right) - \Phi\left(\frac{z_j - w/2 - (1-\rho_z)\mu_z - \rho_z z_i}{\sigma_\epsilon}\right)$
   - 2.3 $P_{iN} = 1 - \Phi\left(\frac{z_N - w/2 - (1-\rho_z)\mu_z - \rho_z z_i}{\sigma_\epsilon}\right)$

Tauchenizing an AR(1)

- We now have a matrix, P, that is a stochastic matrix by construction
- Together with the grid, we have a discrete Markov chain!
- Approximation works pretty well for first moments with only $7 - 11$ points
- Drawbacks
  1. Does not always capture higher moments very well
  2. Big flaw: Underestimates large transition probabilities for high ρ_z
     - If ρ_z really close to one, some states can even become absorbing! Lose irreducibility
     - Rouwenhurst viable alternative discretization for these cases
- Extends easily to higher dimensions (VAR)
  - See Tauchen (1986) for details