Introduction and Tools 1

Chase Coleman

June 22, 2017

NYU Stern

Introduction

Welcome

Thanks for being here.

Introduce yourself:

- Name
- Year of PhD
- Programming langauge you plan to use
- Answer the question you wish I asked you here

Organizational Stuff

Next lecture will be Monday the 26th

All slides and practice questions will be online at `https://github.com/NYUEcon/CompEconWorkshop_2017`. We will try to provide answers to some of the practice questions we assign there as well.

Questions?

Writing Good Code

An Economics Crisis

I think we are in the middle of a simmering crisis in economics...I think the first-order issue with economists and programming languages is not (necessarily) performance, but (1) correctness and (2) software engineering...Have someone with a published numerical paper send you their code, and tell me if you can tell whether it implements the model or not...The software engineering problem in economics needs to be addressed with better training and helping people understand tools like version control, continuous integration, unit testing, the benefits of tested libraries, etc.

What is “Software Engineering?”

“The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.”

What does this mean?

[1]This can improve your citability

What is “Software Engineering?”

“The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.”

What does this mean?

Write code to minimize the probability of having a bug and to ensure that others can understand your code without excessive effort[1].

> *Always code as if the guy who ends up maintaining your code will be a violent pyschopath who knows where you live.*

[1] This can improve your citability

Simple Rules to Live By

These are in no way exhaustive, but are a start

- Don’t Repeat Yourself
  - Use functions rather than writing things repeatedly
  - Writing (c^(1 – gamma) – 1) / (1 – gamma) repeatedly makes it likely you will eventually make a mistake
  - It is a pain to make sure you change all instances if you decide to use a different utility function.
- Write test cases to ensure that various functions you use do what you think they do.
- Use whitespace wisely (ALWAYS HAVE SPACE AFTER COMMA).
- Comment and document your code carefully.

Understanding the Basics

Floating Point Numbers

Computer reads numbers differently than you.

A 16 bit floating point number: $\underbrace{0}_{\text{Sign}}\ \underbrace{01101}_{\text{Exponent}}\ \underbrace{0101010101}_{\text{Mantissa}}$

$$0011010101010101 \rightarrow -1^{\text{Sign}}\left(1+\sum_{n=1}^{10}\text{Mantissa}_n 2^{-n}\right)2^{\text{Exponent - Bias}}$$
$$\rightarrow 1(1.3330078125)2^{13-15} = 0.33325$$

Occasionally useful to understand this fact.

[2] See `http://www.walkingrandomly.com/?p=5380` for discussion

Floating Point Numbers

Computer reads numbers differently than you.

A 16 bit floating point number: $\underbrace{0}_{\text{Sign}}\ \underbrace{01101}_{\text{Exponent}}\ \underbrace{0101010101}_{\text{Mantissa}}$

$$0011010101010101 \to -1^{\text{Sign}}\left(1 + \sum_{n=1}^{10} \text{Mantissa}_n 2^{-n}\right) 2^{\text{Exponent - Bias}}$$
$$\to 1(1.3330078125)2^{13-15} = 0.33325$$

Occasionally useful to understand this fact.

For example, since computer arithmetic is not necessarily associative or commutative[2] for floating points (because of accuracy limitations) checking exact equality between floating points is a *bad* idea...

[2] See **http://www.walkingrandomly.com/?p=5380** for discussion

Ill Conditioned Matrices

Consider $A = \begin{bmatrix} 2 & 6 \\ 2 & 6.00001 \end{bmatrix}$, $b_1 = \begin{bmatrix} 8 \\ 8.00001 \end{bmatrix}$, and $b_2 = \begin{bmatrix} 8 \\ 8.00002 \end{bmatrix}$

- Solution to $Ax = b_1$ is $\begin{bmatrix} 1 & 1 \end{bmatrix}$
- Solution to $Ax = b_2$ is $\begin{bmatrix} 2 & -2 \end{bmatrix}$

An ill conditioned matrix is one that is close to singular. This causes, for the equation $Ax = b$, that small changes in b lead to large changes in x.

Dealing with Ill Conditioned Matrices

Can check whether a matrix is ill conditioned by looking at its condition number:

$$\kappa(A) = \sigma_{\mathsf{max}}(A)/\sigma_{\mathsf{min}}(A)$$

If a matrix is actually ill-conditioned ($\kappa(A)$ large), there is not much you can do except try to make sure b is calculated as precisely as possible.

Column vs Row Major

Matrices (and higher dimensional arrays) are stored as sequential elements of memory. The order in which they are stored determines whether a language is column or row major.

For example, consider the following matrix: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

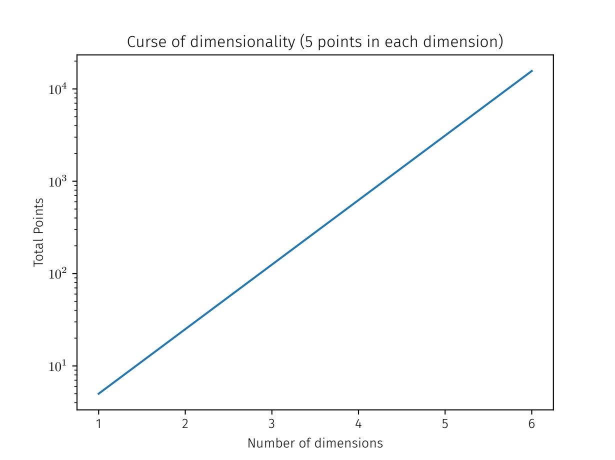
- A row major language would store this as: 1, 2, 3, 4, 5, 6
- A column major language would store this as: 1, 4, 2, 5, 3, 6

This has performance implications (More info here)

Curse of Dimensionality

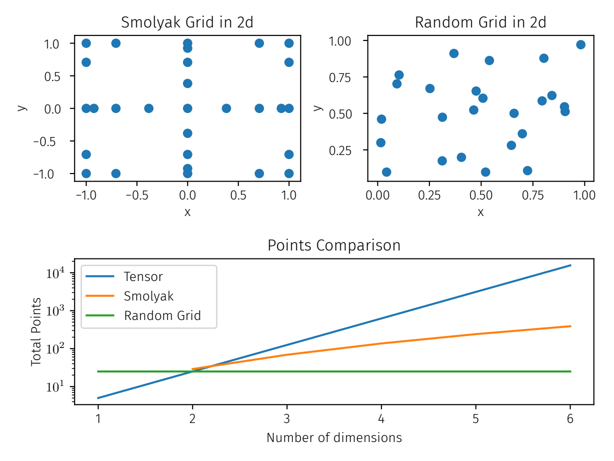
Often need to approximate functions $f : \mathbb{R}^d \to \mathbb{R}$

If want to use n points to approximate function in each dimension using a tensor grid of points then have a total of n^d points.



Curse of Dimensionality

Most obvious way to slow down the curse of dimensionality is to move away from tensor grids!



Vectorization

Sometimes useful to recognize that loops can be avoided by using vectorized operations.

For example, imagine a Markov chain, (S_t, P, π_0), for endowments.

$$V(s_t) = u(s_t) + \beta E\left[V(s_{t+1})|s_t\right]$$

Recognize that $\beta E[V(s_{t+1})|s_t]$ is just $\beta P\vec{V}$ thus $\vec{V} = (I - \beta P)u(\vec{s})$

Root Finding and Optimization

Goal of Root Finding

Given a function $f : \mathbb{R}^n \to \mathbb{R}^n$, find $x \in \mathbb{R}^n$ such that $f(x) = \vec{0}$

Whenever we search for prices in a general equilibrium model, we are seeking to clear markets — i.e. find prices such that the excess demand functions are zero.

In algorithms presented, the stopping point will be determined by how close x_{k+1} and x_k are, but the algorithm is only a success if ultimately $f(x_{k+1}) \approx 0$. For space constraints, I have left this function convergence check out of the algorithm descriptions.

Bisection: Simplest Root Finding Algorithm

Consider a continuous function $f : \mathbb{R} \to \mathbb{R}$ and $a, b \in \mathbb{R}$ such that $f(a)f(b) < 0$. Then Intermediate Value Theorem states, $\exists c \in (a, b)$ such that $f(c) = 0$.

Require: $f(a)f(b) < 0$
  $fc \leftarrow 10$
  while $|b - a| > tol$ **do**
    $c \leftarrow (a + b)/2$
    $fc \leftarrow f(c)$
    if $fa * fc < 0$ **then**
      $b \leftarrow c$
    else
      $a \leftarrow c$
    end if
  end while

Bisection

Simplest root finding algorithm

- Pros: Relatively fast, simple, guaranteed to find a solution given certain conditions
- Cons: Uses little info about function, not natural to extend to higher than 1 dimension

Newton's Method

Natural follow up to Bisection is Newton's Method which uses information about derivatives by linearizing and find the zero of the linearized version.

$$f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0$$

while $|x_{k+1} - x_k| > tol_x$ do
  $x_{k+1} \to x_k - f(x_k)/f'(x_k)$
end while

Newton's Method

- Pros: Even faster than bisection, simple, uses information about derivative
- Cons: Does not always find a solution, requires ability to compute derivative

Secant Method

Secand method is related to Newton's Method, but, rather than use the analytical derivative, it approximates $f'(x)$ by taking secant line between $f(x_{k-1})$ and $f(x_k)$

while $|x_{k+1} - x_k| > tol_x$ **do**

$x_{k+1} \to x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$

end while

Secant Method

- Pros: Similar in speed to Newton's method — The relative speed depends on how much time evaluating the derivative takes vs possibly slower convergence because not using actual derivative, sometimes useful not to need exact derivative
- Cons: Not guaranteed to converge, doesn't use analytical derivative

Multi-dimesional Newton

Use Jacobian rather than derivative

while $|x_{k+1} - x_k| > tol_x$ do

  $A_k = J(x^k)$

  Solve $A_k s_k = -f(x^k)$ for s_k

  $x_{k+1} \to x_k + s_k$

end while

Multi-dimesional Secant (aka Broyden)

This is a multi-dimensional version of Secant method. We must determine how to approximate Jacobian.

One way[3] : $A_{k+1} = A_k + \frac{(y_k - A_k s_k)s_k^T}{s_k^T s_k}$

$A_0 = I$
while $|x_{k+1} - x_k| > tol_x$ **do**
  Solve $A_k s_k = -f(x^k)$ for s_k
  $x_{k+1} \to x_k + s_k$
  $y_k = f(x^{k+1}) - f(x^k)$
  $A_{k+1} = A_k + \frac{(y_k - A_k s_k)s_k^T}{s_k^T s_k}$
end while

[3] This ensure that $A_{k+1}q = A_k q$ whenever $q^T s_k = 0$ (predicted change in directions orthogonal to s_k with new Jacobian matrix are equal to predicted change in that direction with old Jacobian matrix)

Gauss-Jacobi and Gauss-Seidel

Both of these algorithms simplify a n-dimensional problem to a 1-dimensional problem. Solve for x_i such that $f(x_i, x_{-i}) = 0$. However, the approach is slightly different.

- **Gauss-Jacobi**: Find x^{k+1} such that its elements satisfy following equations

$$f^1(x_1^{k+1}, x_2^k, \dots, x_n^k) = 0$$
$$f^2(x_1^k, x_2^{k+1}, \dots, x_n^k) = 0$$
$$\dots$$

- **Gauss-Seidel**: Find x^{k+1} such that its elements satisfy following equations

$$f^1(x_1^{k+1}, x_2^k, \dots, x_n^k) = 0$$
$$f^2(x_1^{k+1}, x_2^{k+1}, \dots, x_n^k) = 0$$
$$\dots$$

Goal of Optimization

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, find $x^* \in \mathbb{R}^n$ such that $x^* \in \arg\min_x f(x)$

Economic models typically have various agents who are each solving a maximization problem — For example, consumers choose consumption and savings in order to maximize their value today (flow + continuation).

WARNING: All methods shown today are about converging to local extrema…This is often not problematic in economic problems because of "nice" properties, but not universally true.

Grid Search

“Premature optimization is the root of all evil” —Donald Knuth

A very simple way to get an idea of where the minimum of a function is, is to simply compare on a discrete number of points.

This approach is often more effective than people give it credit for.

Golden-Section Search

Member of a class of methods known as bracketing methods.

Require: a, b

  $c \leftarrow b - (b-a)/\phi$

  $d \leftarrow a + (b-a)/\phi$

  while $|c-d| > tol_x$ **do**

    if $f(c) < f(d)$ **then**

      $(b, fb) \leftarrow (d, fd)$

    else

      $(a, fa) \leftarrow (c, fc)$

    end if

    $c \leftarrow b - (b-a)/\phi$

    $d \leftarrow a + (b-a)/\phi$

  end while

Nelder-Mead (Simplex Method)

Member of a class of methods known as polytope methods. The general idea[4] is that you build a simplex and expand, shrink, move, and rotate the simplex until the distance between the points of the simplex are very small.

[4]For a full description refer to `http://nbviewer.jupyter.org/github/QuantEcon/QuantEcon.notebooks/blob/master/chase_nelder_mead.ipynb`

Gradient Descent

The idea of gradient descent is to move in the steepest direction "down the hill." Very effective in first few iterations, but can be slow as it gets near the minimum…

while $|x^{k+1} - x^k| > tol_x$ do

$\quad \gamma^k = \frac{(x^k - x^{k-1})^T (\nabla f(x^k) - \nabla f(x^{k-1}))}{||\nabla f(x^k) - \nabla f(x^{k-1})||^2}$

$\quad x^{k+1} \leftarrow x^k - \gamma^k \nabla f(x^k)$

end while

Newton's Method (Optimization)

Like finding the root of $f'(x)$... Similar in idea to gradient descent, but requires/uses additional information (Hessian) which overcomes the problem of when gradient descent gets slow.

while $|x_{k+1} - x_k| > tol_x$ **do**
  Compute gradient, $\nabla f(x^k)$
  Compute Hessian, $H(x^k)$
  Solve for s^k in $H(x^k)s^k = -(\nabla f(x^k))^T$
  $x^{k+1} \leftarrow x^k + s^k$
end while

BFGS Method

Similar to the ideas to gradient descent and Newton's method, but want to get bonuses of having information about Hessian without actually computing it. Approximation of Hessian is B_k.

while $|x_{k+1} - x_k| > tol_x$ do
    p_k as solution of $B_k p_k = -\nabla f(x_k)$
    Perform line search to find good stepsize α_k
    $s_k \leftarrow \alpha_k p_k$
    $x^{k+1} \leftarrow x^k + s_k$
    $y_k = \nabla f(x^{k+1}) - \nabla f(x^k)$
    $B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$
end while

Goal of Constrained Optimization

Given functions $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^n \to \mathbb{R}^p$ solve

$$x^* = \arg\min_x f(x)$$

subject to

$$g(x^*) = 0$$
$$h(x^*) \leq 0$$

Constrained: Penalty Function Methods

One way to solve a constrained optimization problem is by just building the Lagrangian and using the theory in the Karush-Kuhn-Tucker conditions.

while $|x_{k+1} - x_k| > tol_x$ **do**

Solve a particular unconstrained problem to get x_{k+1}

$$x_{k+1} \leftarrow \arg\min_x f(x) + \frac{1}{2} P_k \left(\sum_i g_i(x)^2 + \sum_j \max\{0, h_j(x)\}^2 \right)$$

$P_{k+1} = \chi P_k$

end while

Why increase P over time? (If too large then ill-conditioning can hurt accuracy)

Constrained: Sequential Quadratic

Leverages the fact that problems with quadratic objectives and linear constraints are “easy” to solve. Instead of solving original problem, solve a sequence of Quadratic-Linear problems

$$\begin{aligned} &\min_s (x^k - s)^T \mathcal{L}_x x(x^k, \lambda^k, \mu^k)(x^k - s) \\ &\text{s.t. } g_x(x^k)(x^k - s) = 0 \\ &\qquad h_x(x)(x^k - s) \leq 0 \end{aligned}$$

Column vs Row Major: Supplement

Computers have different layers of memory storage. In order to apply operations, data must be moved to a small piece of memory called L1 memory. L1 memory lives directly on the processor and is very fast.

As computers have evolved, they have gotten better at “guessing” what data you will need next. Instead of moving just a single element of an array, the processor will retrieve a small block of consecutive elements.

As you iterate along an array, this means the computer will need to spend less time passing memory between RAM and L1 and can spend more time doing actual operations.

Additional info: online

Back to Column vs Row Major

References

- Numerical Methods in Economics by Kenneth L Judd