47-805: Computational Methods
Introduction & General issues
Judd Chapters 1-2

David Childers (thanks to Y. Kryukov, K. Judd, and U. Doraszelski)

CMU, Tepper School of Business

March 13, 2023

Plan for today

1. Computational methods: why and how to use them
2. General points on:
    1. Number representation
    2. Problem solving
    3. Coding

Next time: Solving systems of linear equations (Chapter 3)

What are computational methods?

- Economics is full of problems without closed-form solutions:
  - Macro: dynamic problems
  - Econometrics: estimators solve optimization or integration problems
  - General equilibrium is a system of nonlinear eq-s
  - Finance: stochastic calculus & diff. eq's
  - Applied Micro: Games, especially dynamic ones
- Pencil & paper methods limit us to special cases
- Realistic applications lead to analytically intractable problems
- Numerical methods let us substantially relax constraint on tractability

Computing power & its limits

- Moore's law: computing power doubles every 18 months
  - Faster CPUs, more CPUs in the same box
  - Server farms / Supercomputers: 100's of CPUs/GPUs
- But: CPU is a complement to human brain, not a substitute
  - Brute force can take too long, or fail to solve the problem.
    $\Rightarrow$ choice of model form, solution method, starting value, etc.
  - Inappropriate methods can produce unreasonable results
    $\Rightarrow$ economic interpretation & verification of results.
- Numerical methods is a field dedicated to solving problems
  - Economics is about finding interesting problem to solve, and interpreting the solution; **not** the solution methods
  - But: need to know strength and weaknesses of each method

What can numerical methods do?

- Solve intractable problems:
  - Realistic models in both macro and micro
  - Specialized estimation problems (GMM, MLE, Bayes)
  - Policy experiments on estimated model
- Generate hypothesis or counterexamples
  - Can say: X might happen, maybe even likely/unlikely
  - Cannot say: X will always/never happen
- Check derivations, quantify & visualize effects
- Still, can only solve one parameterization at a time:
  - Large number of examples instead of general results

Criticism of numerical methods, & response

- Limited to a finite number of parameterizations
    - Theory is usually limited to a finite set of functional forms
    - We can approximate smooth functions
    - Can combine w/ symbolic/logical methods for non-local results
- Introduces errors
    - We can bound or measure them: compute and verify
- Is a "Black box": what are economic forces behind the result?
    - Do comparative statics, e.g. try turning effects off
    - Take analytical derivation as far as you can
- Computational methods extend analytical ones, not replace them

Computational research stages

1. Pick an economic issue, represent as mathematical problem
2. Pick a numerical solution method
3. Code it up, solve, experiment with it
4. Interpret the results:
    1. Note behavior consistent with existing theory
    2. Find economic reasons behind unexpected behavior, make sure it is not caused by numeric issues
    3. Solve trivial cases (e.g. discount factor = 0), where you know how the solution should look like
    4. Robustness checks: different numeric method(s) & settings, different functional forms, different model
5. Use data to quantify model features, test structure
6. 4.1-2 or 5 core of theory, empirical paper, respectively, 4.4 = brief section at the end + details in online appendix

Course goals

- Solve dynamic problems (e.g. discrete time):

$$\begin{aligned} & \max \sum \beta^t u\left(f\left(k_t\right) - k_{t+1}\right) \\ \Rightarrow \quad & V\left(k\right) = \max_{k'} u\left(f\left(k\right) - k'\right) + \beta V\left(k'\right) \end{aligned}$$

- Maximization $\Rightarrow$ nonlinear equations $\Rightarrow$ linear equations
    - Also: constraints, integration $\left(\beta \mathbf{E}\left[V\left(k'\right)|k, c\right]\right)$, etc.
- Unknown function $V\left(\cdot\right)$: polynomials or splines
- Solving the dynamic problem:
    - Dynamic programming (iterate on Bellman)
    - Projection: find $V\left(\cdot\right)$ and policy functions directly

Model taxonomy

- Time:
  - Discrete: $V(k) = \max_c u(k,c) + \beta V(f(k,c))$
  - Continuous: $\rho V(k) = \max_c u(k,c) + V'(k)\, g(k,c)$
- State:
  - Discrete ($k \in \{k_i\}$)
  - Continuous ($k \geq 0$)
- State transitions:
  - Deterministic: ... $+\beta V(f(k,c))$
  - Stochastic: ... $+\beta \mathbf{E}_{k'}[V(k')\,|k,c]$
- We can solve any combination of the above
  - Pick features that match your economic story
  - "Simplest" case (discrete time & state, deterministic transitions) often leads to unreasonable results, and can be hard to compute.

Course structure

1. Solving for an unknown number (or vector):
   - Solving linear and nonlinear equations
   - Optimization
   - Integration and simulation
2. Solving for an unknown function:
   - Polynomial, spline, & other approximations
   - Differential equations
   - Projection methods
3. Applications to dynamic problems in Econ:
   - Dynamic programming
   - Optimal control, Euler equations
   - Other advanced topics (suggestions?)

Tradeoffs

- Time vs. precision
- Time vs. robustness
- Machine time vs. programming time

Course logistics

- Canvas: lectures, homeworks, announcements
- **Homeworks** - your best chance to review the material, learn to code & receive feedback
  - Submit meaningful write-up & discussion; upload code as ZIP
  - Teams up to 3 people; HW 1 should be done alone.
  - Can't use outside help, e.g.: other students, last year solutions, code from other sources
  - Can use Internet for reference purposes (with citation)
  - Lowest scoring homework will be dropped (given honest effort)
- No midterm
- Final exam: 48 hour take-home, date TBD

Number representation & machine zero

- At lowest (binary) level, computer can only deal with integers
- Real numbers are stored as $\pm m * 2^{\pm n}$
  - $m =$ mantissa, $n =$ exponent
- **Double precision** storage format: m and n share the same 64 bits $\implies$ both are bounded
- Bound on exponent n determines lowest and largest values
- **Machine zero**: smallest positive number that machine can represent
  - Matlab: `realmin`/Julia `nextfloat(-Inf)`/Numpy `np.finfo(np.float64).min` $\approx$ -2e-308
  - Largest possible number: `realmax`/`prevfloat(Inf)`/`np.finfo(np.float64).max` $\approx$ 2e+308 $\approx e^{710}$

Machine epsilon

- Real numbers are stored as $\pm m * 2^{\pm n}$

Bound on mantissa m limits the number of digits:

- **Machine epsilon**: smallest difference from 1.0 that the machine can represent
- Given Matlab/Julia/Python 64-bit precision: `eps()` $\approx$ 2e-16
  - If you have $A \approx 1$, and `A-B` $\approx 1e\text{-}16$, then you can assume that $A = B$.
  - Example: type "`0.6-0.2-0.2-0.2`" into Matlab/Julia/Python
- Machine zero $<$ machine epsilon:
  - m and n share the same 64 bits
  - many digits in m leave less room for n
  - change units to keep your values in (0.01, 100)

Error propagation

- Mathematical operations propagate rounding error
- Solving: $x^2 - 26x + 1 = 0 \implies x = 13 - \sqrt{168} = 0.0385186$
- Machine that stores 5 decimal digits would compute:
  $13.000 - 12.961 = 0.039$
- Answer has precision reduced from 5 meaningful digits to 2
    - Small difference between large numbers will be imprecise
- If A and B are on the scale of 10^6,
  you cannot expect `A-B` to go below `1e-10`

Computation speed

- Basic arithmetic ($+$, –, *, $/$) is fast
- Everything else is approximated through those
- Functions with no closed form take a lot longer
- Exponent takes several times longer than multiplication:

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

- $\Rightarrow$ pre-compute & store expressions that will not change during iterations
- $\Rightarrow$ transform the formula, e.g. Horner's method:
  - $a_0 + a_1 x + a_2 x^2 + a_3 x^3$, vs.
  - $a_0 + x\left(a_1 + x\left(a_2 + a_3 x\right)\right)$

Solution methods

Direct methods (e.g. $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$):

- Result in an "exact" solution
- Takes finite (but potentially long) time

Iterative methods:

- Each iteration k generates a new guess at the solution: $x_{k+1} = g\left(x_k, x_{k-1}, ...\right)$.
- E.g. $x_k = \left[e^a\right]_k = \sum_{n=0}^{k} \frac{a^n}{n!} = x_{k-1} + \frac{a^k}{k!}$
- Each guess is (hopefully) closer to the true solution
    - I.e. sequence (hopefully) converges, to the solution
- Stopping criterion & max. number of iterations trade off computation time vs. precision.

Order notation

- Let x_k be the k^{th} element of a sequence and $f(k)$ a nonnegative sequence
- x_k is $O(f(k))$ (say "x_k is big O $f(k)$") if

$$\limsup_{k\to\infty}\left|\frac{x_k}{f(k)}\right|<\infty$$

- x_k is $o(f(k))$ (say "x_k is little o $f(k)$") if

$$\lim_{k\to\infty}\left|\frac{x_k}{f(k)}\right|=0$$

Computational Complexity: Some minimal theory

- Goal of computation is to take some (finite) input x (a *problem*) and produce the correct output y (the "result") by means of a function $f(.)$ (a *program*) that applies a series of steps which can be performed by a machine
- Standard to study case where $y \in \{True, False\}$
- Computational theory asks several questions about $f()$
  1. Computability: $\exists f()$ which produces y in finite time?
     - Fact: $\exists$ problems for which answer is *no*:
     - These problems are called "noncomputable"
  2. Supposing answer to (1) is *yes*, how many resources does it take to compute?
     - Time: # of operations
     - Space: # of bits of memory
     - Other resources as needed

Computational Resources

- Answers to resource cost allowed can depend on size n of input
  - If # ops (bits) $O(n^k)$, $k < \infty$ problem is polynomial time (space)
  - If # ops (bits) $O(2^{n^k})$ problem is exponential time (space)
- Biggest unsolved problem in compsci is, roughly, what kinds of problems are in each class
- Numerics looks at problems where y is a number, which allows consideration of other resources
  - Precision: For $\epsilon > 0$, let $f(x)$ produce $\tilde{y}$ such that $\|y - \tilde{y}\| < \epsilon$
  - Probability: For $\delta > 0$ allow $\tilde{y}$ random such that $Pr(\|y - \tilde{y}\| < \epsilon) > 1 - \delta$

Approach to resources in this class

- Goal: find programs for problems x in some class which are fast, small, precise, reliable.
  - i.e. $ops < O(q(\epsilon, \delta, n))$ with optimal dependency of $q()$ on ϵ, δ, n
  - Equivalently, $\epsilon < O(r(ops, \delta, n))$ for optimal $r()$
- We will not systematically probe optimal production possibilities frontier for any class of problem
- Instead, we will describe reasonable assortment of useful methods, and say a little bit about properties and tradeoffs

Rates of convergence (geometric convention)

- x_k converges to x^* at rate q if:

$$\lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^q} < \infty$$

- x_k converges to x^* **linearly** at rate β if:

$$\lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} = \beta < 1$$

- Convergence at rate $q \implies \|x_k - x^*\| = O(\exp(-q^k))$
- Linear convergence at rate $\beta \implies \|x_k - x^*\| = O(\beta^k)$
- Informally: Linear convergence improves precision by 1 digit each fixed # of iterations $\left(\frac{-1}{\log_{10}(\beta)}\right)$
- Iterative algorithms usually have geometric convergence
- Will use this convention in most of class

Stopping rules

- Want to stop when x_k is close to x^*.
- We do not know $x^* \Rightarrow$ stop when x_k is close to x_{k+1}, i.e. when $f\left(\|x_{k+1} - x_k\|\right) < \delta$
- Using absolute difference $\|x_{k+1} - x_k\|$ is a bad idea if x's are large, due to limited precision.
- Relative difference $\|x_{k+1} - x_k\| \,/\, \|x_k\|$ – better but might "blow up" if x_k is close to zero
- Hybrid rule: $\|x_{k+1} - x_k\| \,/\, [1 + \|x_k\|]$
- Potential problem: we might stop far from x^*
    - E.g. $x_k = \sum_{n=1}^{k} \frac{1}{n} \to \infty$, but $|x_k - x_{k+1}| = \frac{1}{k+1} \to 0$

Testing the solution: compute and verify

- Solving $f(x) = 0$: true solution is x^* is unknown, computed numeric solution is $\hat{x}$
- Forward error analysis (compare x^* to $\hat{x}$) is infeasible
- Backward error analysis: compare $f(\cdot)$ to similar function $\hat{f}(\cdot)$ that has $\hat{f}(\hat{x}) = 0$
    - $\hat{f}(\cdot)$ can be tricky to make meaningful
- Compute and verify: compare $f(\hat{x})$ to 0, normalized for scale if possible
    - GE: $E(p^*) \equiv D(p^*) - S(p^*) = 0$ (absolute error)
    - Relative error: compare $E(\hat{p})/D(\hat{p})$ to 0

Matlab hints

- Every variable is a matrix (or N-dimensional array), of double-precision reals.
- Capitalization matters
- Interpreter language: check variables after error
  - *F12* = break point; `error('Message')` stops code
- Use help:
  - *Menu/Help/Product help*, then maybe *Index*
  - Select command and press F1
  - Type: `help <command>`
- Matrix operations instead of `for` loops:
  - Elementwise operations: `.+ .- .* ./ .^`
  - `Log()`, `exp()` and most functions work on matrices
  - indexing: `lag_X=[NaN X(1:end-1)];`

Readable code

- Header comment %% explaining what this code does
- Set all parameters as variables at the beginning of the code
- Use descriptive comments % every few lines
    - *Ctrl-R* (*command* / on Mac) and *Ctrl-T* = (un)comment lines
- Use offsets with control structures (`for, while, if`)
    - *Ctrl-[* and *Ctrl-]* = offset selected lines
- Cell mode (=paragraphs): "`%%`" starts a new cell
    - Ctrl-Enter runs the current cell

Writing the code

- One does not simply type 500 lines of code
  - Mistakes will happen
- Try to test every "block" of code
  - F9 runs selected code
  - Use Matlab's cells (%% Headers, Ctrl-Enter to run)
  - Give it trivial inputs, so you can know the correct output
  - Verify output using manual calculation or alternative formula
- Provide meaningful output on screen
  - E.g. iteration report: x, difference from stopping criterion.
  - Look up `fprintf` for formatted output

Julia hints

- Every variable has a *type* (Float64, Int64, Array, etc) and every function must be compatible with the type of the input.
- Programming with types improves speed and readability by specializing method to input structure.
- Capitalization matters
- Use help:
  - `https://docs.julialang.org/` and `https://julia.quantecon.org` invaluable sources
  - Type: `?` <`command`>
- `for` loops: not slow like Matlab, but *broadcasting* still useful
  - Elementwise operations: `.+` `.-` `.*` `./` `.^`
  - `log.()`, `exp.()` etc elementwise using . symbol, to whole array without.

Readable code

- Code in Jupyter notebooks mixes text cells and code cells
  - Provide useful header plus context and explanations
- Use notebooks for exploratory model building, collections .jl scripts (in terminal or IDE like VSCode) for large projects
- Code in scripts should have header and comments
- Use descriptive comments # every few lines
  - *Ctrl-/* (*command* / on Mac) in VSCode IDE
- Set all parameters as variables at the beginning of the code
- Use offsets with control structures (`for`, `while`, `if`)

Writing the code

- One does not simply type 500 lines of code
  - Mistakes will happen
- Try to test every "block" of code
  - Run code in REPL (interactively) to see output
  - Encapsulate discrete behaviors in functions: prevents repetition, improves speed, allows repurposing code when model changes
  - `using Test` package allows setting formal tests
  - Verify output using manual calculation or alternative formula
- Provide meaningful output on screen
  - E.g. iteration report: x, difference from stopping criterion.
  - Look up `println` for formatted output

Python hints

- Spaces and capitalization matter, indices start at 0 instead of 1
- Use help:
  - `https://python.quantecon.org` invaluable source
  - Colab and VSCode have docs as popup
- Python is *interpreted* and *dynamically typed*:
  - Commands make a guess at type of data at runtime rather than requiring explicit declaration
  - Good for convenience, interactivity, bad for reliability and speed
  - *Libraries* like `numpy`, `scipy` have fast precompiled functions
  - `numba` allows `@jit` (just-in-time) compilation to optimize own functions

Readable code

- Code in Jupyter notebooks mixes text cells and code cells
    - Provide useful header plus context and explanations
- Use notebooks for exploratory model building, collections of scripts (in terminal or IDE like VSCode) for large projects
- Code in scripts should have header and comments
- Use descriptive comments # every few lines
    - *Ctrl-/* (*command /* on Mac) in VSCode IDE
- Set all parameters as variables at the beginning of the code
- Use offsets with control structures (`for`, `while`, `if`)

Writing the code

- One does not simply type 500 lines of code
  - Mistakes will happen
- Try to test every "block" of code
  - Run code in interactively to see output
  - Encapsulate discrete behaviors in functions and classes: prevents repetition, improves speed, allows repurposing code when model changes
  - Verify output using manual calculation or alternative formula
- Provide meaningful output on screen
  - E.g. iteration report: x, difference from stopping criterion.
  - `print(f"Text {variable}")` for formatted output