Topic 1
Appendix B: the Need for Speed

Adam Hal Spencer

The University of Nottingham

Applied Computational Economics

Roadmap

Outline

- Value function iteration with a basic grid search is super simple and reliable.
- It can be slow though.
- Keep increasing the size of your state space: the curse of dimensionality.
- How can we speed things up?

Outline

- Here are three things you can do.
- They vary in the amount of time it takes to learn/implement.

(1) Quick fix acceleration methods.

(2) Longer-term fix: learn a faster programming language.

(3) Longer-longer-term fix: supercomputers.

Roadmap

Howard's Improvement Algorithm

- What is it that makes VFI slow?
- Always optimising!
- Can we optimise less and leverage the contraction mapping property more?

Howard's Improvement Algorithm

- Idea: we only update the policy functions (optimise) occasionally.
- Just plug our current estimate of the policy function into the Bellman equation and let it contract! Keep iterating!
- Say that we're trying to solve the problem

$$v(x) = \max_{x' \in X} \; u(x - x') + \beta v(x')$$

Howard's Improvement Algorithm

- Make an initial guess for the value function as usual $v_0(x)$.

(a) Discretise the state space X into $\{x_i\}_{i=1}^N$ gridpoints.

(b) Solve the Bellman equation for one run as you would in grid search

$$v_1(x_i) = \max_{x' \in \{x_i\}_{i=1}^N} u(x_i - x') + \beta v_0(x')$$

where you'll do this for each $i = 1, ..., N$. Denote the optimal choice function $x' = g(x_i)$, which you have computed numerically.

Howard's Improvement Algorithm

(c) Iterate K times now without optimising. Run:

$$\tilde{v}_k(x_i) = u(x_i - g(x_i)) + \beta \tilde{v}_{k-1}(x')$$

for $\tilde{v}_1(x') = v_1(x')$ and $k = 2, ..., K$. Then re-optimise

$$v_2(x_i) = \max_{x' \in \{x_i\}_{i=1}^N} u(x_i - x') + \beta \tilde{v}_K(x')$$

and repeat step **(c)** again with $\tilde{v}_1(x') = v_2(x')$.

- $K \in \mathbb{N}$ is something you need to choose.
- I.e. after how many runs on the contraction do I re-update the policy function?
- Time between re-optimises of the policy function is longer, but fewer re-updates.

Exploiting Monotonicity

- Under certain conditions, (that are usually met in these models), the optimal policy function $x'(x)$ is monotonic.
- Means we can restrict our set of possible choices!
- E.g. what's the policy choice $x'(x_i)$: the optimal choice for current state x_i?
- Search over the set $\{x_i, x_{i+1}, x_{i+2}, ..., x_N\}$ rather than $\{x_1, x_2, ..., x_i, x_{i+1}, ..., x_N\}$.

Roadmap

Other Languages

- I was instructed to teach you Matlab or Julia here in this course.
- They're (probably) most widely-used languages amongst economists.
- That's fine, but when you have a problem with large dimensionality, you switch to Fortran.

Fortran

- Fortran stands for Formula Translation.
- It's super old: created in 1957.
- It's a fu$*ing beast though when it comes to numerical computing.

Fortran

- My advisor at UW-Madison, during the first lecture of our Computational Methods class said:

  > *Learn Fortran this weekend. It'll be the the worst weekend of your life. But it'll make the rest of your life infinitely easier (Corbae, 2014).*

Fortran

- What's the problem?
- This language is super old, so debugging is absolutely terrifying.
- It can take hours to track-down one simple mistake.
- Much harder to de-bug than Matlab.
- But check this out...

Fortran

- Aruoba and Fernandez-Villaverde (2018) solved the neoclassical growth dynamic programming problem several times in several different programming languages...

Fortran

Table 1: Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
C++	GCC-7.3.0	1.60	1.00
	Intel C++ 18.0.2	1.67	1.04
	Clang 5.1	1.64	1.03
Fortran	GCC-7.3.0	1.61	1.01
	Intel Fortran 18.0.2	1.74	1.09
Java	9.04	3.20	2.00
Julia	0.7.0	2.35	1.47
	0.7.0, fast	2.14	1.34
Matlab	2018a	4.80	3.00
Python	CPython 2.7.14	145.27	90.79
	CPython 3.6.4	166.75	104.22
R	3.4.3	57.06	35.66
Mathematica	11.3.0, base	1634.94	1021.84
Matlab, Mex	2018a	2.01	1.26
Rcpp	3.4.3	6.60	4.13
Python	Numba 0.37.9	2.31	1.44
	Cython	2.13	1.33
Mathematica	11.3.0, idiomatic	4.42	2.76

Fortran

- Julia is good, but Fortran destroys almost everything but C++.
- But it's hard to get the hang of 100%.
- You should only invest in Fortran if you want to do macro or other stuff that involves lots of state variables.
- For most people, Matlab is fine.

Roadmap

Supercomputers

- There are two types of supercomputing:
  - High **performance** computing.
  - High **throughput** computing.

Supercomputers

- The type you should use depends on the type of speed gains you need. Ask the following question:

  Can I break my problem-up into a thousand processes, without the need for any of the processes to speak to each other at all?

- If the answer is yes, you use high-throughput computing (and you're lucky for that matter).

- If the answer is no, you use high performance computing.

Supercomputers

- Examples:
  - My job market paper: I had three endogenous state variables. I had 60 cores that could divide-up my state space for the value function iteration. The cores could optimise certain parts of the state space then pool all their information at the end of each iteration. HPC.
  - Say you have a small problem and you want to calibrate your model. You can divide-up your parameter space into 1000 pieces, evaluate each set and calculate your SMM criterion function each time. Pick the smallest criterion function set of parameters as your starting point for the formal SMM procedure. HTC.

Supercomputers

- HPC basically takes a bunch of cores and puts them together as if they were one computer. All cores work on exactly the same job.
- HTC in contrast takes the same job with some different parameters (that you tell the submit node) and then they operate without speaking to each other.

Supercomputers

- Parellelising a problem in Matlab can be straightforward, (e.g. replace the word "for" in your loops with "parfor": parallel for).
- But since it's commercial software, there will usually be a limit to the number of cores you can use at once.
- Again, Fortran is the king here.
- Plus, since it's so old, it is very easy to use on a cluster with OpenMP, (or MPI).
- There's usually no limit on cores.

Supercomputers

- Also: just to mention, more cores for HPC doesn't always mean faster.
- E.g. 60 cores v.s. 2000. It takes time for them to pool all their information: so you might lose-out in moving a lot.

Supercomputers

- The time cost to using supercomputers comes from (1) it takes time to get access to one, (I've never used the one here at Nottingham so you'd need to look into it).
- (2) These supercomputers all use command-line operating systems, so you need to get used to that. Plus again, debugging is hard.

GPU Processing

- I have played with this before.
- I got it to work, but didn't really see much in the way of speed gains, (I guess I coded it inefficiently).
- I've heard the gains are massive. But you need special graphics cards to do it. Compatibility between software/graphics cards is again an issue.
- I believe C is the language most compatible with GPU processing, but I don't know this language.
- I spent like two weeks straight on this and didn't make much progress: there's not much in the way of documentation online, (or at least in 2017 there wasn't).

Roadmap

Conclusion

- Start with the basic single-core grid search, if it's too slow, there are options.
- Varying degree of fixed costs to learn them though.