High-performance Computing for Economists

Lukas Mann[1]

[1]Adapted from notes by R. Cioffi, J. Fernández-Villaverde, P. Guerrón, and D. Zarruk

May 22, 2023

Basics

Computation in economics I

- Computing has become a central tool in economics:

  1. Macro $\rightarrow$ solution and estimation of dynamic equilibrium models with heterogeneous agents, policy evaluation and forecast, ...
  2. Micro $\rightarrow$ computation of games, labor/life-cycle models, models of industry dynamics, study of networks, bounded rationality and agent-based models, ...
  3. Econometrics $\rightarrow$ non-standard estimators, simulation-based estimators, large datasets, ...
  4. International/spatial economics $\rightarrow$ models with heterogeneous firms and countries, dynamic models of international trade, spatial models, economic consequences of climate change and environmental policies, ...
  5. Finance $\rightarrow$ asset pricing, non-arbitrage conditions, VaR, ...

Computation in economics II

- Widespread movement across all scientific and engineering fields: *On Computing* by Paul S. Rosenbloom.
- Economics has been slow to embrace computation but is catching up.
- Nowadays, computation in economics is also becoming key in:
  1. Policy making institutions.
  2. Regulatory agencies.
  3. Industry.

Consequences for students

- This means that you will spend a substantial share of your professional career:

  1. Coding.
  2. Dealing with coauthors and research assistants that code.
  3. Reading and evaluating computational papers.
  4. Supervising/regulating people using computational methods.

High-performance computing

- High-performance computing (HPC) deals with scientific problems that require substantial computational power.

- Even simple problems in economics generate HPC challenges:

  1. Dynamic programing with several state variables.
  2. Highly non-linear DSGE models with many shocks.
  3. Problems with occasionally binding constraints.
  4. Complex asset pricing.
  5. Structural estimation.
  6. Frontier estimators without closed form solutions.
  7. Handling large datasets.

Parallel processing

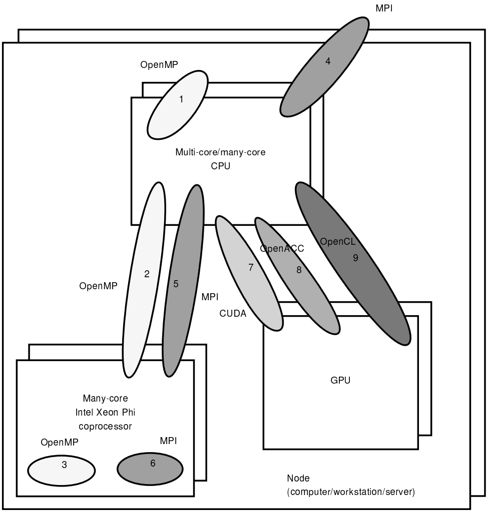
- Usually, but not always, HPC involves the use of several processors:

  1. Multi-core/many-core CPUs (in a single machine or networked).
  2. Many-core coprocessors.
  3. GPUs (graphics processing units).
  4. TPUs (tensor processing units).
  5. FPGAs (field-programmable gate arrays).

- Most of these machines are available to all researchers at low prices.

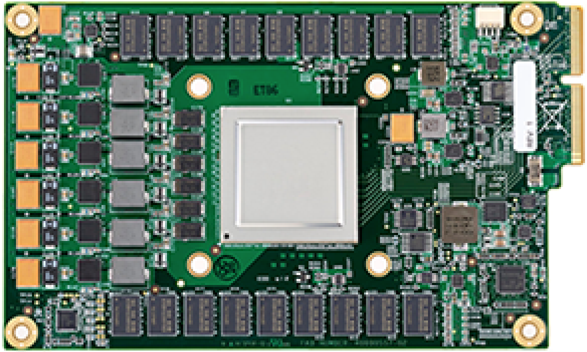
- Nevertheless, we will also think about how to produce efficient serial code (although, following most recent developments, we will not emphasize much vectorization).

Parallel paradigms



GPUs

TPUs



Total time

- Often HPC is framed regarding running time.
- In practice, coding and debugging time is often equally relevant.
- Why does running time matter?
  - Determines what kind of models you can solve
  - Determines the time spent working on a project
- Why does proper coding style matter?
  - Determines the time spent working on a project
  - Determines susceptibility to errors
- We will spend considerable effort in discussing proper coding.

Some resources

- HPC carpentry: `https://hpc-carpentry.github.io/`.
- JFV's homepage: `https://www.sas.upenn.edu/~jesusfv/teaching.html`.
- The Art of HPC (Victor Eijkhout): `https://theartofhpc.com/index.html`.
- Livermore documentation and tutorials: `https://hpc.llnl.gov/training/`.
- HPC Wire: `https://www.hpcwire.com/`.
- Inside HPC: `https://insidehpc.com/`.
- *High Performance Computing: Modern Systems and Practices* by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz.
- *Introduction to High Performance Computing for Scientists and Engineers* by Georg Hager and Gerhard Wellein.

Software Engineering

Randall Hyde

“Hackers are born, software engineers are made, and great programmers are a bit of both”

Motivation

- You are taking a class on computational methods.
- Even if only because you need to complete your homework, you just became a software engineer (and not just a simple coder/developer!).
- Coding is, in part, an art ($\tau\acute{\varepsilon}\chi\nu\eta$).
- But, in an even larger part, coding is about having good knowledge ($\grave{\epsilon}\pi\iota\sigma\tau\acute{\eta}\mu\eta$) of proven procedures.
- You can and should learn and use these procedures.
- Don't reinvent the wheel!

The goal I

- To produce code that is:

1. *Correct*: We are scientist and we pursue correct answers.
2. *Efficient*: you want to get your Ph.D., to get tenure, to become an influential research economist in FINITE time.
   2.1 Coding + Running time must be minimized.
   2.2 Trade-off between coding and running time.
3. *Maintainable*: revise and resubmits, extensions of existing papers.

The goal II

4. *Reproducible*: other researchers (and your future selves; beware of `bit-rot`!) must be able to replicate your results.

5. *Documented*: other researchers (and your future selves) must be able to understand how it works.

6. *Scalable*: code that can be used by you and by other researchers as a base for further development.

7. *Portable*: code that can work across a reasonable range of machines.

The means

- Knowledge accumulated over decades in computational-intensive fields and by the industry.
- Standard part of a CS curriculum.

This class I

- We will cover some of the basics of software engineering (theory and tools).
- Adapted, though, to the requirements of an economist (at least, as determined by our own experience).
- For instance, you will probably not have different "releases" of a code, UML and design patterns will not be important, testing will be done in differently.
- At the same time, speed and reproducibility will be key.
- Also, we will cover material that it is taught in some basic courses on CS but that economists may be less familiar with (IDEs, Profilers, OOP,...).
- We will emphasize the idea that you want to use well-tested tools that give *you* as much control as possible within a reasonable cost.

This class II

- Brief introduction that cannot substitute:

  1. A real course on software engineering (and other techniques) in your local CS department.

  2. Standard books:

     - *Object-Oriented and Classical Software Engineering*, by Stephen Schach.

     - *The Mythical Man-Month: Essays on Software Engineering*, by Fred Brooks.

     - *Code Complete: A Practical Handbook of Software Construction, Second Edition (2nd ed.)* by Steve McConnell.

     - Other books we will mention throughout the lectures.

  3. Reading the technical documentation (RTFM).

This class III

- Additional resources:
  1. Own experience.
  2. Searching the internet (GIYF).
  3. Stack Overflow: `http://stackoverflow.com/`
  4. ChatGPT
  5. Github Co-Pilot
  6. `Youtube`
  7. Software carpentry: `http://software-carpentry.org/index.html`.

Some final comments

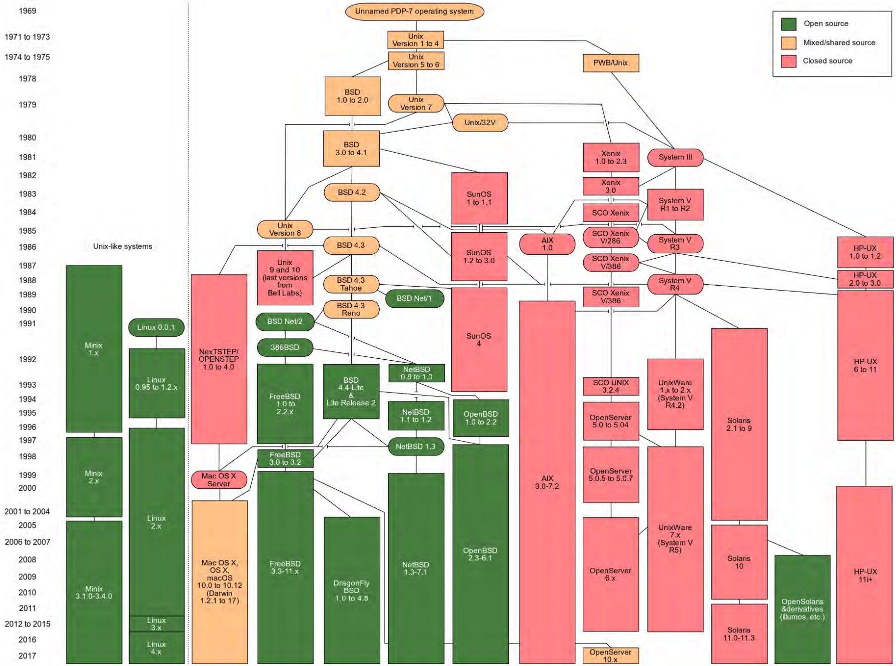
- None of the contents of this class is a substitute for common sense, self-discipline, and hard work.
- Moreover, experience is more important than anything else.
- There is no silver bullet out there.
- Beware of the temptation of: “If I just update my OS/computer/app everything would be fine.”

Swimming without water

Operating Systems

Operating systems

- If you are going to undertake some serious computation, you want to become a skilled user of your `OS`.
- Two main families of OS:
  1. `Unix` and `Unix-like` family (Ken Thompson and collaborators at Bell Labs):
     1.1 Commercial versions: `AIX`, `HP-UX`, `Solaris`, ...
     1.2 Open source: `OpenBSD`, `Linux`, ...
     1.3 `macOS`.
  2. `Windows` family.



1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2015
2016
2017
Open source
Mixed/shared source
Closed source
Unix-like systems
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
SunOS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
BSD 4.3
AIX 1.0
System V R3
HP-UX 1.0 to 1.2
Unix 9 and 10 (last versions from Bell Labs)
SunOS 1.2 to 3.0
SCO Xenix V/386
HP-UX 2.0 to 3.0
BSD 4.3 Tahoe
System V R4
BSD Net/1
SCO Xenix V/386
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1
BSD Net/2
SunOS 4
386BSD
NexTSTEP/ OPENSTEP 1.0 to 4.0
Linux 0.95 to 1.2.x
NetBSD 0.8 to 1.0
HP-UX 6 to 11
SCO UNIX 3.2.4
FreeBSD 1.0 to 2.2.x
BSD 4.4-Lite & Lite Release 2
UnixWare 1.x to 2.x (System V R4.2)
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
Solaris 2.1 to 9
NetBSD 1.3
Minix 2.x
FreeBSD 3.0 to 3.2
Mac OS X Server
AIX 3.0-7.2
OpenServer 5.0.5 to 5.0.7
Linux 2.x
UnixWare 7.x (System V R5)
OpenBSD 2.3-6.1
Solaris 10
Mac OS X, OS X, macOS 10.0 to 10.12 (Darwin 1.2.1 to 17)
FreeBSD 3.3-11.x
NetBSD 1.3-7.1
OpenServer 6.x
DragonFly BSD 1.0 to 4.8
HP-UX 11i+
Minix 3.1.0-3.4.0
OpenSolaris &derivatives (illumos, etc.)
Linux 3.x
Solaris 11.0-11.3
Linux 4.x
OpenServer 10.x

Why Unix/Linux? I

- Industry-tested for four decades: powerful beyond your imagination.
- Standard OS for scientific computation and high-performance computing $\rightarrow$ as of November 2022, **all** the Top 500 supercomputers in the world run on `Linux`.
- Particularly important for:
  1. Access to servers.
  2. Web services such as AWS.
  3. Parallelization
- It will be around forever: if you learned to use `Unix` in 1973, you can open a Mac today and use its terminal without problems.
- Watch `https://youtu.be/tc4ROCJYbm0`.

Why Unix/Linux? II

- Many (`Linux`) open source implementations. For example, `Ubuntu` and `Fedora`. You can check `https://www.distrowatch.com/`
- Much more robust: small kernel.
- Much safer: Sandboxing and rich file permission system.
- Easier to port code.
- Plenty of tools.
- For instance, a default `macOS` installation comes with `Emacs`,`VI`, `SHH`, `GCC`, `Python`, `Perl`,....
- Existence of `Windows` emulators such as `VMWare` or `Parallels`.
- Windows now allows Linux subsystems (`WSL`) - good compromise if you do not want to switch fully

Philosophy of Unix/Linux

- "Building blocks"+"glue" (pipes and filters) to build new tools:
    1. Building blocks: programs that do only one thing, but they do it well.
    2. Glue: you can easily combine them.
- Ability to handle Generalized Regular Expressions:

Ken Thompson

A regular expression is a pattern which specifies a set of strings of characters; it is said to match certain strings.

Interaction

- Both GUIs and command lines.
- The command line works through a shell.

Shell

Shell

- Different shells: **`bash`** (Bourne-again shell, by Brian Fox), **`bourne`**, **`ksh`**,...

- For instance, if you type

```
In [1]: echo $0
```

  on a Mac Terminal, you will probably get:

```
Out[1]: -bash
```

- Easy to change shells.

- Most of them offer similar capabilities, but **`bash`** is the most popular.

- Basic tutorial: **`http://swcarpentry.github.io/shell-novice/`**

Some basic instructions I

To check present working directory:

$ **pwd**

To list directories and files:

$ ls

To list all directories and files, including hidden ones:

$ ls -all

To navigate into directory `myDirectory`:

$ **cd** myDirectory

To go back:

$ **cd** ..

Some basic instructions II

To create a directory

$ mkdir myDirectory

To remove a directory

$ rmdir myDirectory

To copy `myFile`

$ cp myFile

To move `myFile` to `yourFile`:

$ mv myFile yourFile

To remove `myFile`:

$ rm myFile

Some basic instructions III

To find `myFile`:

$ find myFile

To concatenate and print `myFile`:

$ **cat** myFile

Wild card:

$ ls myF*

Manual entries

$ man

Bang

$!!

Some basic instructions IV

To check permissions on `myFile`:

$ ls -l myFile

To change permissions (mode) on `myFile`:

$ chmod 744 myFile

Interpretation digit:

- 4: read access.
- 2: write access.
- 1: execute access.

Interpretation position:

- first: *user* access.
- second: *group* access.
- third: *other* access.

Advanced shell interaction

- Customization: `.bash_profile`, `.bash_logout`, and `.bashrc` files.
- Shell programming:
  1. Automatization.
  2. Aliases.

```
$ alias myproject = '~/dropbox/figures'
```

Some more information

- Good references (among many):

  1. *Unix in a Nutshell (4th Edition)*, by Arnold Robbins.
  2. *Learning Unix for OS X: Going Deep With the Terminal and Shell (2nd Edition)*, by Dave Taylor.
  3. *A Practical Guide to Linux Commands, Editors, and Shell Programming (4th Edition)*, by Mark G. Sobell.
  4. *Learning the bash Shell: Unix Shell Programming (3rd Edition)*, by Cameron Newham and Bill Rosenblatt.

Editors

Editors

- By default, you should try to use plain, open files:

  1. Text files (`READMEs`, `HOWTOs`, ...).
  2. `CSV` files for data (also as text files).

- `.docx` and `.xlsx` files change over time and may not be portable.
- A good editor is an excellent way to write text files.
- A good editor will also help you write source code (with syntax highlight) and tex files.

Alternatives I

Harry J. Paarsch

Choose your editor with more care than you would your spouse because you will spend more time with your editor, even after the spouse is gone.

- Classics:

  1. `Emacs`, originally by Richard Stallman.
  2. `VI`.
  3. `Textwrangler`.
  4. `Notepad++`.
  5. `JEdit.`
  6. `Nano/Pico` (simplest).

Alternatives II

- New generation:

  1. `VS Code`

     - Highly recommended, particularly for development in `Julia` where it has become the standard, replacing `JuliaPro`
     - With extensions, it becomes as powerful as an IDE
     - Multiple extensions for various languages available

  2. `Sublime`.
  3. `Neovim`.

IDEs

IDEs I

- Integrated Developer Environment: tools to write, compile, debug, run, and version control code.
- Advantages and disadvantages.
- Standard choices:
  1. `JetBrains (CLion, PyCharm,...)`.
  2. `Xcode`.
  3. `VisualStudio`.
  4. `Eclipse` (with `Parallel Application Developers package`).
  5. `NetBeans`.

IDEs II

- Specific languages:
  1. `Spyder`.
  2. `RStudio`.
  3. `Matlab IDE`.
  4. `Wolfram Workbench`.

Dynamic notebooks

- Why?
- `Jupyter`: `http://jupyter.org/`. Also, `JupyterLab`.
- `Markdown`: `https://www.markdownguide.org/`.
- If you work in `R`:
  1. Knitr package: `https://yihui.name/knitr/`.
  2. *Dynamic Documents with R and knitr (2nd ed.)* by Yihui Xie.
- `Pandoc`: `http://pandoc.org/`

Build Automation

Build automation

- A build tool automatizes the linking and compilation of code.
- This includes latex and pdf codes!
- Why?
    1. Avoid repetitive task.
    2. Get all the complicated linking and compiling options right (and, if text, graphs, options, etc.).
    3. Avoid errors.
    4. Reproducibility.
- `GNU Make` and `CMake`.

Why Make?

- Programmed by Stuart Feldman, when he was a summer intern!
- Open source.
- Well documented.
- Close to Unix.
- Additional tools: `etags`, `cscope`, `ctree`.

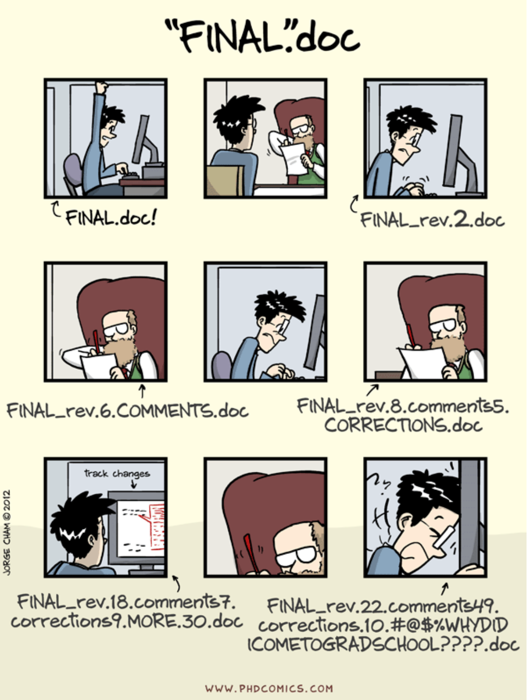
Basic idea

- You build a make file: script file with:
    1. Instructions to make a file.
    2. Update dependencies.
    3. Clean old files.
- Daily builds. Continuous integration proposes even more.
- *Managing Projects with GNU Make (3rd Edition)* by Robert Mecklenburg, `http://oreilly.com/catalog/make3/book/`.

Containers

- A container is stand-alone, executable package of some software.
- It should include everything needed to run it: code, system tools, system libraries, settings, ...
- Why? Keep all your environment together and allow for multi-platform development and team coding.
- Easier alternative to VMs.
- Most popular: Docker `https://www.docker.com/`.
- Built around dockerfiles and layers.

Version Control


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Challenge

- Projects nearly always end up involving many versions of code (even of your `tex` files).
- Version control is the management of changes to your code or documents.
- This is important:
  1. When you are working yourself, to keep track of changes, and to be able to return to previous versions.
  2. When you are working with coauthors, to coordinate task and ensure that all authors have the right version of the file.
- Hard to emphasize how important this is in real life: when, why, and how you did it?

Simple solution

- Possible (but usually suboptimal) solutions:

  1. Indexing files by version (mytextfile_1, mytextfile_July212018), with major and minor patches (x.y.z), e.g., 0.1.7
  2. Having a VCS folder (for **`Version Control System)`**.
  3. Dropbox or similar services.
  4. Automatic back-up software (**`Time Machine`** for **`Mac`**, **`fwbackups`** for **`Linux`**).

- While 1-4 are useful, they are not good enough to handle complex projects.

- Nevertheless, **SET UP** automatic backups.

Version control software

- Alternative? version control software (open source):
  1. First generation: **`RCS`**.
  2. Second generation: **`CVS`**, **`Subversion`**.
  3. Third generation: **`Mercurial`**, **`GIT`**.
- Components:
  1. Repository: place where the files are stored.
  2. Checking out: getting a file from the repository.
  3. Checking in or committing: placing a file back into the repository.

Workflow

- Standard procedure:

  1. You check out a file from the repository.
  2. You work on it.
  3. You put it back with (optional) some comments about your changes.
  4. The software keeps track of the changes (including different branches) and allows you to recover old versions.

- Version control software is only as good as your own self-discipline.

Git I

- Modern, distributed version control system.
- Developed by Linus Torvalds and Junio Hamano.
- Simple and lightweight, yet extremely powerful.
- Easy to learn basic skills.
- Originally designed for command line instructions, but now several good GUIs: `Sourcetree, GitHub Desktop, GitKraken`.

Git II

- Very popular: `http://www.github.com`
- Also, `https://about.gitlab.com/`
- A good reference: *Pro Git* by Scott Chacon, `http://git-scm.com/book`.
- Best git practices: `https://sethrobertson.github.io/GitBestPractices/`
- Many tutorials online.
- Integrated with `VS Code` and `RStudio` and can easily integrate with most standard IDEs.

Basics of Root Finding and Numerical Optimization

Introduction

- Numerical optimization and root-finding are central to solving many models numerically
- Will see: Two sides of the same coin
- Every scientific programming language has good optimization packages - use them!
- But: Good to know what happens ”under the hood” for algorithm choice, debugging, etc.

Root finding

- Why is root finding important?
  - Often: Can write model solution as solution to a fixed point problem
  - Fixed point problem = Root finding problem

$$M(x|\theta) = 0$$

- Given θ, find x s.t. equation is satisfied.

- Several options for root finding but we will focus on three main ones:
  - Bisection
  - Newton-Raphson (gold standard)
  - Quasi-Newton methods

Bisection

- Simplest root-finding algorithm
- Only works if x is one-dimensional
- Very robust (guaranteed convergence)
  - Start with a, b s.t. $M(a|\theta) < 0, M(b|\theta) > 0$
  - If $\frac{a+b}{2} > 0$, set $a := \frac{a+b}{2}$, otherwise $b := \frac{a+b}{2}$
  - Repeat
- Convergence with rate 2^{-n}.

Newton-Raphson

- Based on Taylor-approximation of function:

$$0 = \underbrace{y + \mathcal{J}(x)(x' - x)}_{\approx f(x')}$$

$$\implies x' = x - \mathcal{J}(x)^{-1} y$$

- Algorithm:
  - Given x, find x' using the formula above
  - Set $x := x'$
  - Repeat

- Potential problem: Might be costly to calculate $\mathcal{J}(x)$ (N evaluations each iteration).

- Solution: Quasi-Newton methods

Quasi-Newton Methods: Broyden

- Trick: Avoid re-computing $\mathcal{J}$ at every iteration n
- Instead, the Broyden method uses the following approximation:

$$J_n(x_n - x_{n-1}) \approx (f(x_n) - f(x_{n-1}))$$

- Secant instead of tangent
- If dimension > 1, this is indeterminate, so minimize $||J_n - J_{n-1}||_F$
- Formula:

$$J_n = J_{n-1} + \frac{\Delta f_n - J_{n-1}\Delta x_n}{||\Delta x_n||^2}\Delta x_n^T$$

Sobol sequences

- In some situations, we might need to search $\mathbb{R}^N$ for a good starting guess (or an approximate root)
- Can transform problem into search along hypercube $[0, 1]^N$
- Sobol sequences are an efficient way to equally distribute points in the hypercube
- Most programming languages will have packages to generate Sobol sequences (e.g. ”Sobol.jl”)

Optimization: Gradient-based

- In many applications, need to numerically optimize objective, e.g.:
    - Agents optimizing
    - Calibrating over-identified models
    - Maximum likelihood estimation
    - Neural networks
- Want to maximize $f : \mathbb{R}^N \mapsto \mathbb{R}$
- Can take FOC:

$$\nabla f(x) = 0$$

$\rightarrow$ Root finding problem! Root-finding methods apply.

Optimization: Newton

- Same as in root finding case, but now $\mathcal{J}_{\nabla f} = \mathcal{H}_f$ (Hessian is Jacobian of ∇f)

$$x_{n+1} = x_n - \mathcal{H}(x_n)^{-1} \nabla f(x_n)$$

- Optimization: More information - $\mathcal{H}$ is symmetric (and might be negative/positive definite)!
- Can use this info for constructing alternative Quasi-Newton methods

Optimization: Quasi-Newton

- Several different options:

Method	$B_{k+1} =$	$H_{k+1} = B_{k+1}^{-1} =$
BFGS	$B_k + \frac{y_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k} - \frac{B_k \Delta x_k (B_k \Delta x_k)^{\mathrm{T}}}{\Delta x_k^{\mathrm{T}} B_k \, \Delta x_k}$	$\left(I - \frac{\Delta x_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) H_k \left(I - \frac{y_k \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) + \frac{\Delta x_k \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \, \Delta x_k}$
Broyden	$B_k + \frac{y_k - B_k \Delta x_k}{\Delta x_k^{\mathrm{T}} \, \Delta x_k} \, \Delta x_k^{\mathrm{T}}$	$H_k + \frac{(\Delta x_k - H_k y_k) \Delta x_k^{\mathrm{T}} H_k}{\Delta x_k^{\mathrm{T}} H_k \, y_k}$
Broyden family	$(1 - \varphi_k) B_{k+1}^{\mathrm{BFGS}} + \varphi_k B_{k+1}^{\mathrm{DFP}}, \quad \varphi \in [0, 1]$	
DFP	$\left(I - \frac{y_k \, \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \, \Delta x_k}\right) B_k \left(I - \frac{\Delta x_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \, \Delta x_k}\right) + \frac{y_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \, \Delta x_k}$	$H_k + \frac{\Delta x_k \Delta x_k^{\mathrm{T}}}{\Delta x_k^{\mathrm{T}} \, y_k} - \frac{H_k y_k y_k^{\mathrm{T}} H_k}{y_k^{\mathrm{T}} H_k y_k}$
SR1	$B_k + \frac{(y_k - B_k \, \Delta x_k)(y_k - B_k \, \Delta x_k)^{\mathrm{T}}}{(y_k - B_k \, \Delta x_k)^{\mathrm{T}} \, \Delta x_k}$	$H_k + \frac{(\Delta x_k - H_k y_k)(\Delta x_k - H_k y_k)^{\mathrm{T}}}{(\Delta x_k - H_k y_k)^{\mathrm{T}} y_k}$

Source: `https://en.wikipedia.org/wiki/Quasi-Newton_method`

- Standard for Quasi-Newton: (L-)BFGS ("(Limited memory) Broyden–Fletcher–Goldfarb–Shanno")

Gradient Descent

- Another option: $\mathcal{J}_n = \alpha_n I$

$$x_{n+1} = x_n - \alpha_n \nabla f(x_n)$$

- Line search algorithm determines α_n
- Always step in the direction of steepest descent (”Walking into a valley”)
- Very popular in ML (big N)

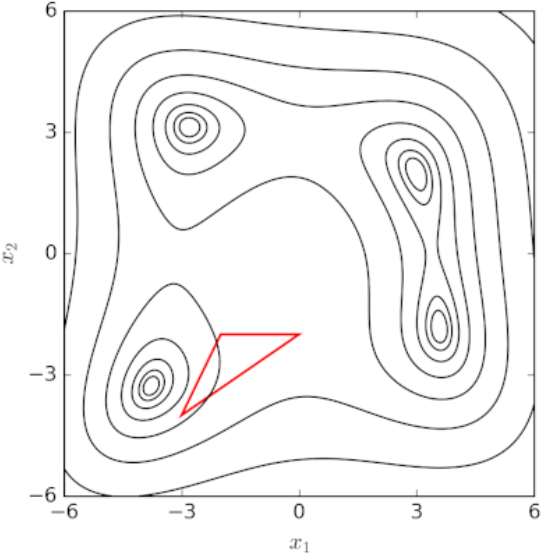
Non-gradient methods

- In some problems, gradient-based methods are hard to implement, e.g. because
  - it is hard to find good initial guesses
  - f is not well-behaved
  - Gradients and Hessians are hard to compute (e.g. because of simulation error)
- Solution: Non-gradient-based methods

Nelder-Mead

- Most robust non-gradient-based method
- Simplex ($N+1$ points) that "wanders" over the surface
- At any point, have collection of points $(x_1, \ldots, x_{n+1})$ where WLOG $f(x_1) \leq \ldots \leq f(x_{n+1})$
- Each algorithm step replaces x_{n+1} with a "better" point
- Terminate when all $(x_1, \ldots, x_{n+1})$ and all $f(x_1) \leq \ldots \leq f(x_{n+1})$ are close enough.

Nelder-Mead



Algorithm Choice

- For root-finding, Newton algorithm is standard
- For optimization, many options but rule of thumb:
  - If problem is very well behaved (small dimensionality, analytic gradient): Newton w/ trust region
  - For larger problems / without analytic gradient: Quasi-Newton (e.g. (L)BFGS)
  - If problem is not well-behaved (e.g. kinky objective): Nelder-Mead