# Advanced Programming: Numerical Function Optimization and Parallel Programming

Erica Moszkowski

April 17, 2020

Numeric Function Optimization

A Brief Intro to Parallelization

## Numeric Function Optimization

## Problem description

We are all very familiar with standard constrained optimization problems:

$$\begin{aligned} \min_x f(x) & \qquad x \in \mathbb{R}^n \\ \text{s.t. } g_i(x) = 0 & \qquad i \in 1, \dots, I \text{ (equality constraints)} \\ \text{and } h_j(x) \geq 0 & \qquad j \in 1, \dots, J \text{ (inequality constraints)} \end{aligned}$$

You all know how to optimize functions when $f$ is continuous and differentiable. But there are a lot of cases where you might prefer to let a computer do the work for you. For example:

- $n$ is large (and it's hard to know what $f$ looks like)
- $f$ has too many modes and checking them all is a pain
- $f$ is not everywhere differentiable
- you will need to minimize $f$ a lot of times

## When should I use numerical optimization?

In economics, common uses include:

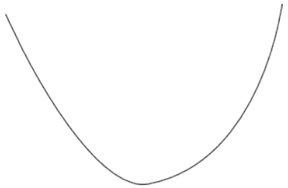
- Maximum likelihood
- Nonlinear GMM

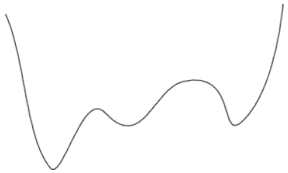
Numerical optimization can take a long time.

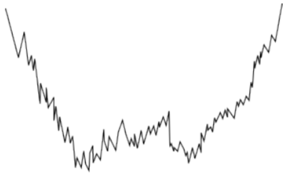
So you should probably **not** use numerical optimization when you can reasonably find an analytical solution, for example:

- Solving for demand/supply functions
- Linear GMM

## What does $f$ actually look like?


Convex problem


Multi-modal problem


Noisy problem

## How does numerical optimization work?

**Your computer does not stare at a picture of your objective function.**

- What does it do?
  - Test out a lot of different points in your parameter space, evaluates the objective function at each one.
  - Testing infinite points will take infinite time, so there is a stopping condition (usually, when $f$ falls by less than 1e-6, we say that the optimization routine has converged).
- Different algorithms explore the parameter space in different ways
- There are many, many optimization algorithms, but 3 main types:
  1. **Gradient descent methods:** well-adapted for convex objective functions
     - *e.g. Newtonian method, steepest-descent method*
  2. **Evolutionary methods:** adapted to multi-modal objective functions
     - *e.g. simulated annealing, particle swarms, Sequential/Hamiltonian Monte Carlo*
  3. **Pattern search methods**: adapted to noisy objective functions
     - *e.g. Nelder-Mead*

## Gradient descent

## Gradient descent idea

**Recall the definition of a minimum:**

$x^*$ is a minimum of $f : \mathbb{R}^n \to \mathbb{R}$ if and only if there exists a radius $\rho > 0$ such that:

- $f$ is defined on $\mathcal{B}(x^*, \rho)$
- $f(x^*) < f(y) \quad \forall \in \mathcal{B}(x^*, \rho)$ , $y \neq x^*$

This is not very helpful for building algorithm, because even a small ball has infinite points in it.

Instead, remember that a **sufficient condition for $x^*$ to be a minimum is**:

- $\nabla f\left(x^\star\right) = 0$
- $\nabla^2 f\left(x^\star\right) > 0$

If $f$ is twice differentiable, this is easy to check!

## Basic gradient descent algorithm

- Pick a starting point $x_0$
- For $k = 0, 1, \ldots$:
    - Evaluate the gradient $\nabla f(x^\star)$
    - Choose a direction to move in $d_k(\nabla f(x^\star))$
    - Choose how big of a step $\rho_k$ to take
    - Compute $x_{t+1} = x_k + \rho_k d_k$

## Choosing the search direction $d_k$

**Steepest Descent Method**: $d_k = -\nabla f(x_k)$

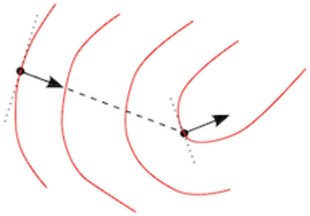


Illustration of steepest-descent path

- **Pro:** you're definitely moving down the objective function
  - $\nabla f(x_k) \cdot d_k = -\nabla f(x_k) \cdot \nabla f(x_k) < 0$
- **Con:** the path oscillates a lot
- **Convergence rate:** linear
  - $\lim_{k\to\infty} \frac{\|x_{k+1} - x^\star\|}{\|x_k - x^\star\|} = a > 0$

## Choosing the search direction $d_k$

**(Quasi-)Newtonian methods**: $d_k = -H_k^{-1} \cdot \nabla f(x_k)$

- $H_k = \nabla^2 f(x_k)$ is the Hessian matrix, or (for quasi-Newton methods) an approximation to it. $H$ should be symmetric and positive definite
- **Idea:** assume that around the optimium, the objective function can be locally approximated as a quadratic. Then use the first and second derivatives to find that optimum.
- **Examples:** BFGS, L-BFGS (low-memory version)
- **Pro:** pretty fast (super-linear!), if local quadratic assumption holds, because path towards optimum is smoother
- **Con:** assumption is hard to verify, and non-smoothness breaks the algorithm
- **Notes:** if you can analytically compute the gradient and the Hessian, you should!

## Choosing the step length $\rho_k$

- You could just choose a fixed step length, but that can lead to steps that are either too big or too small.
- Many algorithms choose a step size adaptively at every $x_k$. For example:
  - Choose some adjustment factor $\beta$
  - Start with $\rho_k^{(p)} = 1$ for $p = 0$
  - While $f(x - \nabla f(x)) > f(x) - \frac{\rho_k^{(1)}}{2} ||\nabla f(x_k)||^2$, update $\rho_k^{(p+1)} = \beta \rho_k^{(p)}$
  - Stop when $\rho_k$ converges.
- **Notice:** balance between computational cost and accuracy

## Supplying gradients and Hessians

Many gradient descent methods require you to supply a gradient or Hessian function. You have a few options here:

- Code up a quick numerical derivative using a fixed $h$ and hand that off
- Use **automatic differentiation (AD)**:
  - AD is a very precise way of calculating derivatives, based on applying the chain rule over and over again.
  - *Warning:* software dependencies on software dependencies. But more and more languages are starting to implement AD, so this should get easier.

## Evolutionary methods

## Darwinian inspiration

- A population of particles (starting $x$ values) is randomly generated
- The fittest $x$'s (i.e. lowest $f(x)$) survive and reproduce, and the others die off
- There are about a million evolutionary strategies you could (and people have) come up with.

## Simulated annealing

- I think of simulated annealing as a blind turtle: it is slow, and bumps into things, but it is tough and very endearing.
- For every $k = 0, \ldots, K$:
  - Start at $x_k $
  - Pick a random move $x'$
  - If the move improves the solution, $x_{k+1} = x'$
  - If the move does not improve the solution:
    - with probability $\exp(\frac{f(x') - f(x_k)}{T})$, reject $x'$ and try again
    - otherwise, set $x_{k+1} = x'$ anyway
- What is $T$?
  - $T$ is often called the "temperature". It normall starts high, and is gradually decreased according to an "annealing schedule".
  - The annealing schedule also determines when SA decides to stop.
- Notice: because SA essentially picks random points, it can take a long time. But because it sometimes accepts upward moves, it can get out of local minima.

## Pattern search methods

## Nelder-Mead

- Basic idea:
  - Start with a set of $n+1$ points $x_0, \ldots, x_n \in \mathbb{R}^n$ that are considered as the vertices of a simplex
  - Then transform each vertex $x_i$ to decrease $f(x_i)$
  - Stop when the simplex gets “small enough”
- Pros: no derivatives: good for non-smooth/discontinuous problems, fairly robust.
  - If you’re having trouble with other algorithms, try Nelder-Mead before giving up.
- Cons: Can take awhile.

## Implementation

## How to use optimizers in most programming languages

Most programming languages have built-in optimization packages for you to use. You usually call an optimization routine using syntax that looks something like this:

```
out = optimize(f, options)
```

- `f` is your objective function. You define it! It can be as simple as 1 arithmetic expression or as complicated as a nested fixed point.
- `optimize`: wrapper function that hands `f` off to an already-implemented optimization routine
- `options` could be a list, keyword args, or an object that lets you:
  - select the optimization routine
  - supply an analytical gradient or Hessian function
  - choose how small the differences in $f$ or $x$ should get before you decide to stop
- `out`: output of the optimization routine, including $x^*$, $f(x^*)$, and some metadata

## Implementation Tips and Warnings

- Most numerical optimizers only *minimize* functions.

  $\rightsquigarrow$ to find a maximum, just multiply your objective function by -1.

- For robustness, always try starting your optimization algorithm at a lot of different points in the parameter space. If they all converge to the same solution, you can be fairly confident you're at the global minimum. If they don't, you should be worried.

- Handling constraints:
  - many programming languages let you enter a matrix of constraints directly into the optimization function and then just do their thing
  - Sometimes I use the following cheat:
    - inside function `f` , check if a constraint is violated. if it is, return `Inf`

- Never forget: **YOUR OPTIMIZATION ALGORITHM IS DUMB AND BLIND.**

## A Brief Intro to Parallelization

## A motivating example: value function iteration

Standard neoclassical growth model: a social planner is maximizing expected discounted utility of a representative household:

$$V(k) = \max_{k'} \{u(c) + \beta V(k')\}$$
$$\text{s.t. } c = k^\alpha + (1-\delta)k - k'$$

- $k$ = current capital, $k'$ = next period capital, $c$ = consumption, $\delta$ = depreciation

Most econ PhD students have solved this problem in their first year courses. The usual algorithm is:

1. Make a grid of $j$ points, $k \in [k_1, k_2, \ldots, k_j]$
2. Initialize $V_0$ (a guess of $V$) for every point $k$ in the grid
3. Until $V^{n+1}(k) - V^n(k) < \epsilon$ for all $k$,

   3.1 compute $\max_{k'} \{u(c) + \beta V(k')\}$ s.t. $c = k^\alpha + (1-\delta)k - k'$

## Example, continued:

We have to compute

$$\max_{k'} \left\{ u(c) + \beta V\left(k'\right) \right\} \text{ s.t. } c = k^{\alpha} + (1-\delta)k - k' \qquad (\star)$$

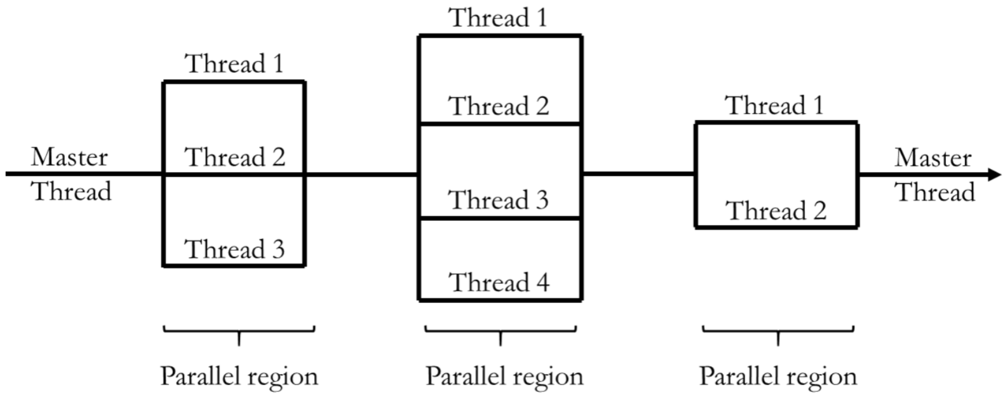
for every $k$ in our grid.

**Notice:** there's no reason we have to compute $(\star)$ for $k_1$, and then $k_2$, and then $k_3$: these computations don't depend on each other.

- If we had $j$ workers, we could compute step 3.1 simultaneously for every $j$.
- If we had $w$ workers, we could give each one $\frac{j}{w}$ points to compute

**Good news:** most computers have $> 1$ processor!

## Master-worker paradigm: a visualization



Source: https://www.sas.upenn.edu/~jesusfv/Guide_Parallel.pdf

## Setting up your code to run in parallel

Your parallelization script should follow these steps:

1. Request worker processes (aka initializing the “parallel pool”)
   - At this point you may need to load dependencies into memory for all worker processes
   - This can take awhile, but it’s a fixed cost
2. Farm out jobs to worker processes, either by:
   - utilizing a parallel for loop (best when each loop does a small amount of work)
   - follow a mapReduce paradigm (best when you have a function that you want to do a lot of work)
3. Release workers so that other people can use them

## Matlab parfor loops

Using the parallel toolbox:

```
# request 6 workers
parpool(6)

# parallel for loop
parfor i = 1:J
    ...
end
```

- Note: for nested loops, choose just 1 layer to parallelize. Otherwise you could end up with worker processes trying to farm out work to new worker processes, and eventually your computer will crash.

## Julia distributed for loops

```
using Distributed

# add processes
addprocs(6)

# Run a nested for loop, with the inner loop parallelized
for i = 1:10
    @sync @distributed for(j=1:J)
    ...
    end
end
```

- `@sync` ensures all workers finish their job before iterating to $i + 1$. This is important if each step depends on what was computed in the previous step.

## MapReduce

MapReduce is a programming paradigm that is extremely useful. It is composed of 2 steps:

1. Map: take input and generate a set of intermediate values to be farmed out
2. Reduce: turn the intermediate values into a single value and return it to the master process

Example: we want to find the average of 8 numbers.

- The average function is linear $\implies$ we could parallelize this computation by farming out 4 averages to 4 different workers, each of which computes an average of 2 numbers.
- Then split the 4 returned numbers into 2 groups of 2, and have workers 1 and 2 compute averages of those .
- Finally, the master process can compute the average of the 2 final averages.

But notice: this involves a lot of passing data back and forth, so it is probably not worth parallelizing.

## MapReduce paradigm in Julia

Suppose we have 10 different matrices and want to get their singular value decompositions:

```
using Distributed
workers = addprocs(10)

# make an array of 10 matrices
M = [rand(1000,1000) for i = 1:10]

# compute the singular values of each matrix
results = pmap(svdvals, M)

# remove workers
rmprocs(workers)
```

## Things to worry about

- **Moving data:** passing data between the master and worker processes creates a HUGE amount of overhead. Like, shockingly huge. This is a serious tradeoff.
- **Race conditions**: Basically, this is when multiple different threads of execution trip over each other. For example, this can happen if thread A requires data that thread B was supposed to provide before thread B has had a chance to compute it. This can also be very difficult to debug.

## General notes about parallel computing

- Whether your problem is easy to parallelize depends on how you set it up. So think carefully before you start programming.
- Your hardware does end up mattering a lot, so there is only so much general advice I can give you about how to speed up your code
- Experiment, experiment, experiment
  - start with small parallel problems, then scale up
  - try multiple ways of parallelizing something
- You will probably be running parallel code on a grid that is also used by others. Therefore:
  - Before you start a process, make sure you know how busy the machine is. Different grids often have special Bash commands that are helpful for this, so spend some time reading about your grid.
  - Leave plenty of processors available for other people.
  - Sharing is caring, but also sysadmins will shut you down.

## Terminology you will run into

- **process**: an instance of a computer program or code that is running
- **core/processor**: a CPU. The part of your computer that computes things.
- **scheduler**: a part of your computer's operating system that is in charge of deciding how to allocate computing power (cores) to processes. When you request workers, the scheduler is the thing that gives them to you.
- **thread**: the smallest sequence of programmed instructions that can be managed independently by a scheduler. Sometimes multiple theads are run on the same processor (this is called "multithreading", and is useful when you want the threads to share data). Sometimes they are run on different processors
- **machine/node**: a computer
- **grid**: a set of different machines that are networked together and can communicate (communication across node)