Minicourse on quantitative macro

Grey Gordon
FRB Richmond[1]

2022

[1]The views are my own and not necessarily those of the FRB of Richmond or the BoG.

Introduction

In this minicourse, I aim to show you three topics:

1. Solving Bellman equations:
   - How to discretize shocks
   - How to speed computation
   - How to ensure convergence
2. Calibrating models:
   - Checking for existence and identification
   - Algorithms for optimization
3. Coding strategies and parallelization:
   - Structuring code in flexible ways with OOP
   - Using parallel paradigms, SMP (OpenMP) and DMP (MPI)
   - Linux for accessing, compiling and running on super computers

Bellman equations

Consider the Bellman equation from the Aiyagari model:

$$V(a,e) = \max_{c,a'} u(c) + \beta \mathbb{E}_{e'|e} V(a',e')$$

$$\text{s.t. } c + a' = we + (1 + r - \delta)a$$

$$c \geq 0, a' \geq \underline{a}$$

$$\log e' = \rho_e \log_e + \sigma_e \varepsilon'$$

Solving the Bellman

The simplest way to solve the h.h. problem: discretization.

Two aspects: discretizing the shocks and discretizing assets.

Suppose $\log e = \rho \log e_{-1} + \sigma\varepsilon$.

There are many methods to convert this to a Markov chain $(e, F(e'|e))$, but I will focus on two

- Tauchen (1986)'s method
- Tanaka and Toda (2013,2015) / Farmer and Toda (2017)

(For a survey of methods, you can see my 2021 EL article.)

Discretizing an AR1 with Tauchen's method

Tauchen's (1986) method is the most common way to discretize

$$z = (1-\rho)\mu + \rho z_{-1} + \sigma\varepsilon, \quad \varepsilon \sim N(0,1).$$

Given a number of states n and a "coverage" κ, the method

- Sets $\mathcal{Z} = \{z_1, \ldots, z_n\}$ as n linearly spaced points from $\mu - \kappa\frac{\sigma}{\sqrt{1-\rho^2}}$ to $\mu + \kappa\frac{\sigma}{\sqrt{1-\rho^2}}$
- Computes the midpoints $m_{i,i+1} = \frac{z_i + z_{i+1}}{2}$
- Determines the transition probabilities for $z_i, z_j \in \mathcal{Z}$ as

$$F(z_i|z_j) = \begin{cases} \Phi(m_{1,2}; a_j, \sigma^2) & \text{if } i = 1 \\ \Phi(m_{i,i+1}; a_j, \sigma^2) - \Phi(m_{i-1,i}; a_j, \sigma^2) & \text{if } i = 2, ..., n-1 \\ 1 - \Phi(m_{n-1,n}; a_j, \sigma^2) & \text{if } i = n \end{cases}$$

where a_j is the conditional mean $a_j = (1-\rho)\mu + \rho z_j$ and $\Phi(x; a, b)$ is the normal cdf with mean a and variance b.

Refining with Tanaka and Toda

A very powerful and flexible method for refining discretizations was constructed by Tanaka and Toda (TT) and extended by Farmer and Toda (FT).

Given transition probabilities $\pi^o_{\mathbf{j}|\mathbf{i}}$, the method selects new probabilities $\pi_{\mathbf{j}|\mathbf{i}}$ that are as close as possible to the original (in the Kullback and Leibler, 1951, sense) by solving, for each $\mathbf{i}$,

$$\begin{aligned} &\min_{\pi_{\mathbf{j}|\mathbf{i}}} \sum_{\mathbf{j}} \pi_{\mathbf{j}|\mathbf{i}} \log\left(\pi_{\mathbf{j}|\mathbf{i}}/\pi^o_{\mathbf{j}|\mathbf{i}}\right) \\ \text{s.t. } &\sum_{\mathbf{j}} \pi_{\mathbf{j}|\mathbf{i}} T^k(z_{\mathbf{j}}) = \bar{T}^k_{\mathbf{i}},\ \sum_{\mathbf{j}} \pi_{\mathbf{j}|\mathbf{i}} = 1,\ \pi_{\mathbf{j}|\mathbf{i}} \geq 0. \end{aligned} \tag{1}$$

$\{\bar{T}^k_{\mathbf{i}}\}_k$ are moments to be matched and $\{T^k(\cdot)\}_k$ are functions such that $\sum_{\mathbf{j}} \pi_{\mathbf{j}|\mathbf{i}} T^k(z_{\mathbf{j}})$ gives the k-th discretized moment.

This is a difficult optimization problem, but TT show its dual simple. It is not always the case that the moments can be matched.

Discretizing VARs

Models often have many shocks or richer dynamics than an AR(1).

These shocks can often be put into VAR(1) form. For example,

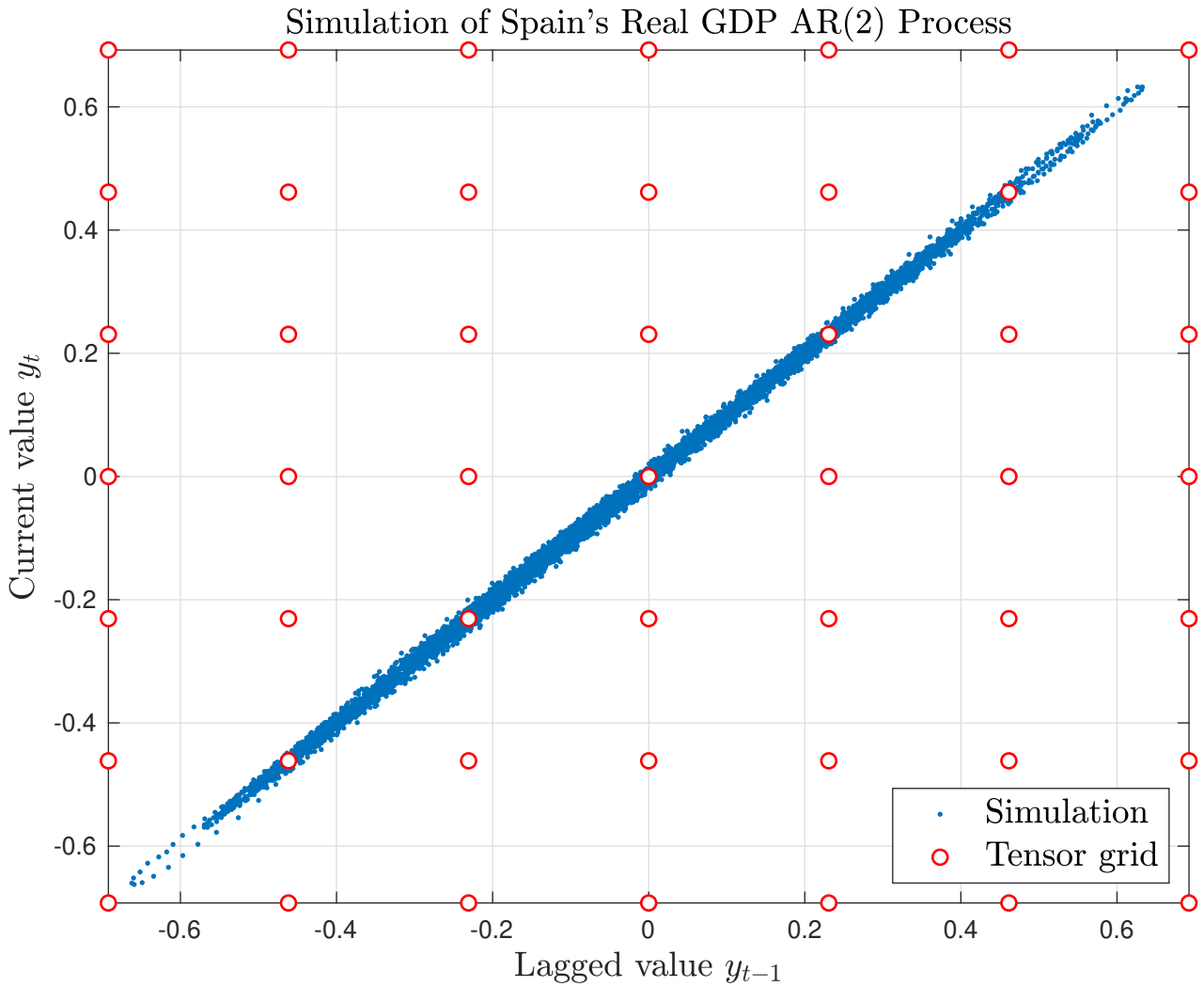
- if y_t follows an AR(2), then $[y_t, y_{t-1}]'$ follows a VAR(1); or,
- if x_t and y_t each follow an AR(1), $[x_t, y_t]'$ follows a VAR(1).

Discretizing VARs then can be important.

The standard approach is to use a tensor grid in each dimension, but this is inefficient.

Discretizing VARs

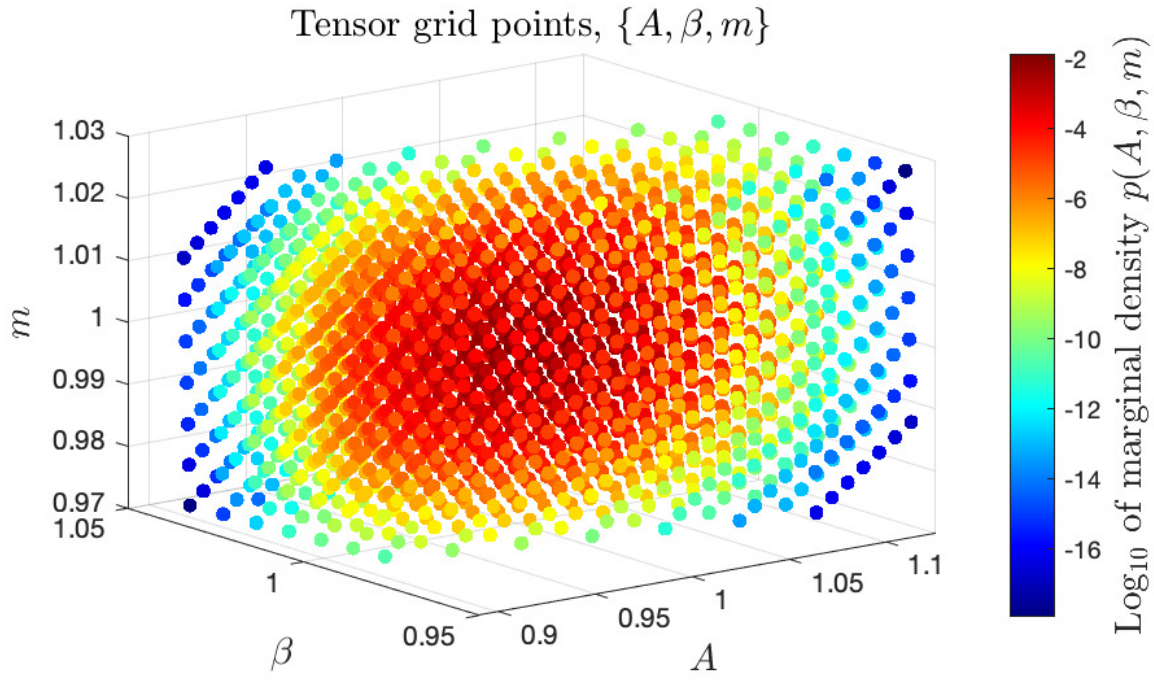
For instance, here's an estimated AR(2) from log GDP in Spain:



All those points outside the ergodic set are wasted.

Discretizing VARs

Or, if you have a couple shocks, or they're correlated...



Note the scale is log_{10}. The density drops off rapidly in the corners.

VAR discretization

Here's a regular VAR:

$$z_t = c + Az_{t-1} + \eta_t \tag{2}$$

with $\eta_t \overset{i.i.d.}{\sim} N(0, \Sigma)$, where z_t is a $D \times 1$ vector.

Because Σ is real and symmetric (not necessarily full rank, though), you can decompose it as $\Sigma = L\Lambda L'$ for $L'L = I$ and Λ diagonal (see Gordon, 2021 for details).

Defining $\tilde{z} = L'z$, $\tilde{c} = L'c$, $\tilde{A} = L'AL$, and $\tilde{\eta}_t = L'\eta_t$, one has

$$\tilde{z}_t = \tilde{c} + \tilde{A}\tilde{z}_{t-1} + \tilde{\eta}_t \tag{3}$$

for $\tilde{\eta}_t \sim N(0, \Lambda)$. (This approach is due to Tauchen, 1986.)

Then $\tilde{z}_{t,d}|\tilde{z}_{t-1} \sim N(\tilde{c}_d + \tilde{A}_{(d,\cdot)}\tilde{z}_{t-1}, \Lambda_d)$, where $\tilde{A}_{(d,\cdot)}$ is the dth row of $\tilde{A}$, which can be approximated using Tauchen's logic.

Efficient VAR discretization

The transformation multiplying by L' eliminates correlations only in the innovations, not the unconditional correlations across states.

So, we're left with the inefficiency if we stick with a tensor grid.

But, pick a method for discretizing the VAR, such as a tensor grid, to get $\{z_{\mathbf{i}}\}_{\mathbf{i}}$ and $\{\pi(z_{\mathbf{j}}|z_{\mathbf{i}})\}_{\mathbf{i},\mathbf{j}}$

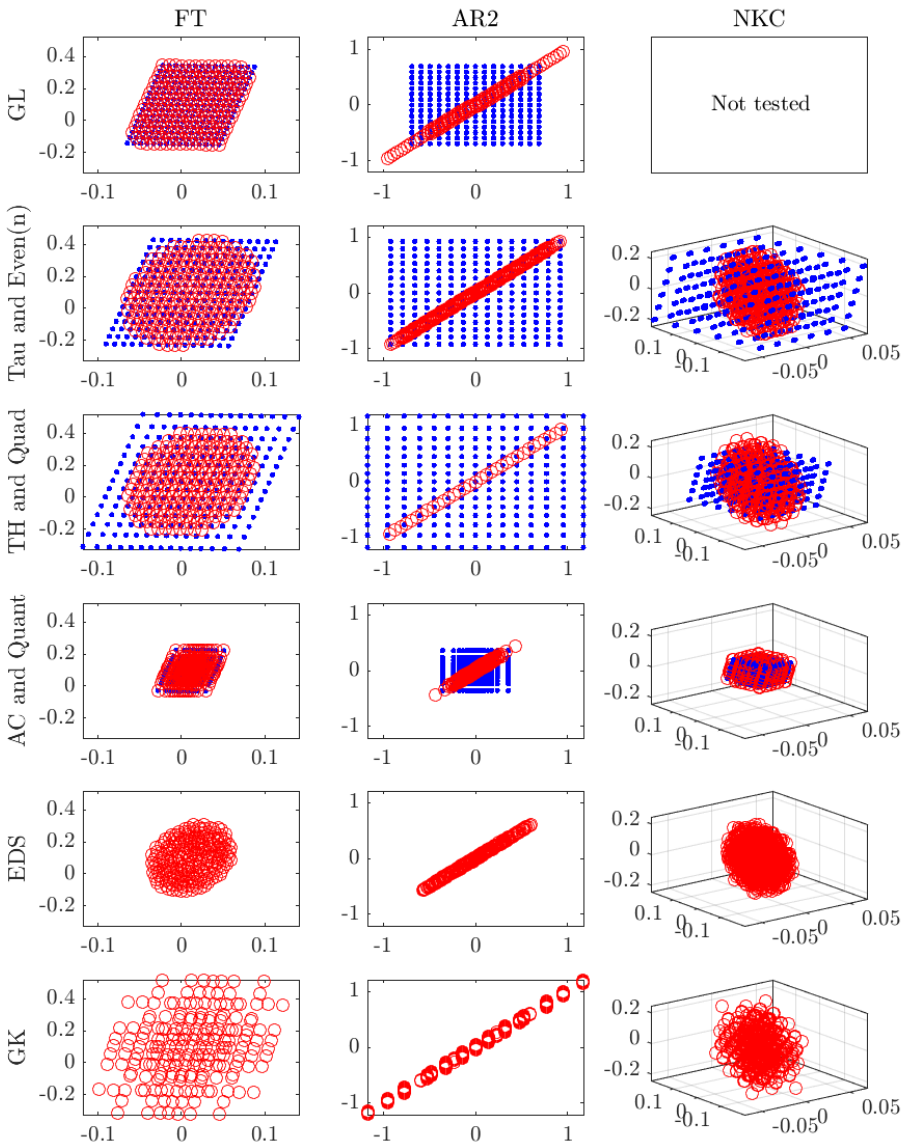
Gordon (2021) suggests approximating the invariant distribution across states by computing $\pi(z_{\mathbf{i}}) \propto N(z_{\mathbf{i}}; E, V)$ and dropping states with $\pi(z_{\mathbf{i}}) < \underline{\pi}$.

After dropping those states, $\sum_{\mathbf{j}} \pi(z_{\mathbf{j}}|z_{\mathbf{i}}) \neq 1$, typically. So one can either simply rescale so they sum to 1 or use the TT method to refine the approximation.

This gives a new set of states $\{\tilde{z}_{\mathbf{i}}\}_{\mathbf{i}}$ and transition probabilities $\{\tilde{\pi}(\tilde{z}_{\mathbf{j}}|\tilde{z}_{\mathbf{i}})\}_{\mathbf{i},\mathbf{j}}$.

Then scale up the initial discretization $\#\{z_{\mathbf{i}}\}_{\mathbf{i}}$ until you get close to a desired number of points $\#\{\tilde{z}_{\mathbf{i}}\}_{\mathbf{i}}$.

Efficient VAR discretization



The graph shows several different types of non-efficient discretizations (in blue) along with their efficient counterparts (in red).

Three different VAR specifications.

Efficient grids improve accuracy/speed, and especially when matching conditional means and variances using TT/FT.

See Gordon (2021) for more details.

Solving the Bellman

For discretizing the asset state space $[\underline{a}, \infty)$, many options.

Pick an upper bound $\overline{a}$, and then create $\#\mathcal{A}$ points $\{\underline{a} := a_1, \ldots, a_{\#\mathcal{A}} =: \overline{a}\}$.

- Linear spacing (most common)
- Geometric spacing (to cluster around certain areas, use $\underline{a}$) $\Rightarrow (a_{i+1} - a_i) = \gamma(a_i - a_{i-1})$ for user-chosen γ.
- Others

(Aside: if you have to find a position of some value in intervals defined on a grid, usually "binary search" is best.)

Solving the Bellman

Have to decide if also want a discrete choice space.

While if one restricts $a \in \mathcal{A}$, one commonly $a' \in \mathcal{A}$ requires as well $\Rightarrow$ no interpolation.

But, $a' \in [\underline{a}, \overline{a}]$ can also be done, or $a' \in \overline{\mathcal{A}}$ (for $\overline{\mathcal{A}}$ some other set).

Suppose for now that the choice space is also discrete with $a' \in \mathcal{A}$.

Solving the Bellman

Recall the pure problem:

$$V(a,e) = \max_{c,a'} u(c) + \beta \mathbb{E}_{e'|e} V(a',e')$$
$$\text{s.t. } c + a' = we + (1+r-\delta)a$$
$$c \geq 0, a' \geq \underline{a}$$

Write this like

$$V(i,j) = \max_{i' \in \{1,\ldots,\#\mathcal{A}\}} u(c(i,j,i')) + \beta \sum_{j'} F(j'|j) V(i',j')$$
$$\text{s.t. } c(i,j,i') = -a' + we + (1+r-\delta)a$$
$$c(i,j,i') \geq 0, a' \geq \underline{a}$$
$$e = \mathcal{E}_j$$
$$a = \mathcal{A}_i$$
$$a' = \mathcal{A}_{i'}$$

for $(i,j) \in \{1,\ldots,\#\mathcal{A}\} \times \{1,\ldots,\#\mathcal{E}\}$.

Solving the Bellman

$$V(i,j) = \max_{i' \in \{1,\ldots,\#\mathcal{A}\}} u(c(i,j,i')) + \beta \sum_{j'} F(j'|j) V(i',j')$$

$$\text{s.t. } c(i,j,i') = -a' + we + (1+r-\delta)a$$

$$c(i,j,i') \geq 0, a' \geq \underline{a}$$

$$e = \mathcal{E}_j$$

$$a = \mathcal{A}_i$$

$$a' = \mathcal{A}_{i'}$$

for $(i,j) \in \{1,\ldots,\#\mathcal{A}\} \times \{1,\ldots,\#\mathcal{E}\}$.

When $V(i,j)$ is not well-defined, define $V(i,j) = -10^{100}$.

Now, the computer can understand this. Note V is a matrix.

Solving the Bellman

A couple things to note:

- For efficiency, $W(i', j) := \beta \sum_{j'} F(j'|j) V(i', j')$ should be precomputed.
- The most efficient way is using matrix multiplication.
- If F is stored in matrix π like $\pi(j, j') = F(j'|j)$, then $W = \beta V \pi^T$.
- $c(i, j, i')$ on the other hand should not be precomputed — it is too large.
- The optimal policy function for this problem, call it $g(i, j)$, gives the asset index.
- It is monotone in i, but $c(i, j, g(i, j))$ generally is not.
- Computationally, CRRA of 2 is the best! $u(c) = -1/c$. Exponentiation can be 10x or more slower than division.

Solving the Bellman

How to solve for $V(i,j)$?

Fix a j (i.e., loop over j), and let $n = n' = \#\mathcal{A}$.

Define $\pi(i,i') = u(c(i,j,i')) + W(i')$.

The problem is to find $\Pi(i) = \max_{i' \in \{1,\ldots,n'\}} \pi(i,i')$ and $g(i)$ for $i = 1,\ldots,n$.

Solving the Bellman

Brute force does the following:

For each i, compute $\pi(i, i')$ for each i' and save the best.

(Fortran code on the next slide; nearly the same as Matlab.)

Solving the Bellman

Simplest way: brute-force grid search.

```
do iE = 1,nE
  do iA = 1,nA
    ! Set default utility/policies
    V(iA,iE) = -1d100
    c(iA,iE) = -1d0
    api(iA,iE) = 1 ! set to 1 to guarantee monotonicity
    ! Loop over all iAp
    do iAp = 1,nA
      if (gA(iAp)<borrowlim) cycle
      ctmp = -gA(iAp) + gA(iA)*(1d0+r-deprec) + w*gE(iE)
      if (ctmp>0d0) then
        vtmp = PU(ctmp) + CV(iAp,iE)
        ! Compare utility versus best so far
        if (vtmp>V(iA,iE)) then
          V(iA,iE) = vtmp
          c(iA,iE) = ctmp
          api(iA,iE) = iAp
        end if
      end if
    end do
```

Solving the Bellman

One can go $\approx$ twice as fast using a small modification.

Because $g(i-1) < g(i)$, can restrict the choice space for $g(i)$ to $\{g(i-1), \ldots, n\}$.

Call this simple monotonicity.

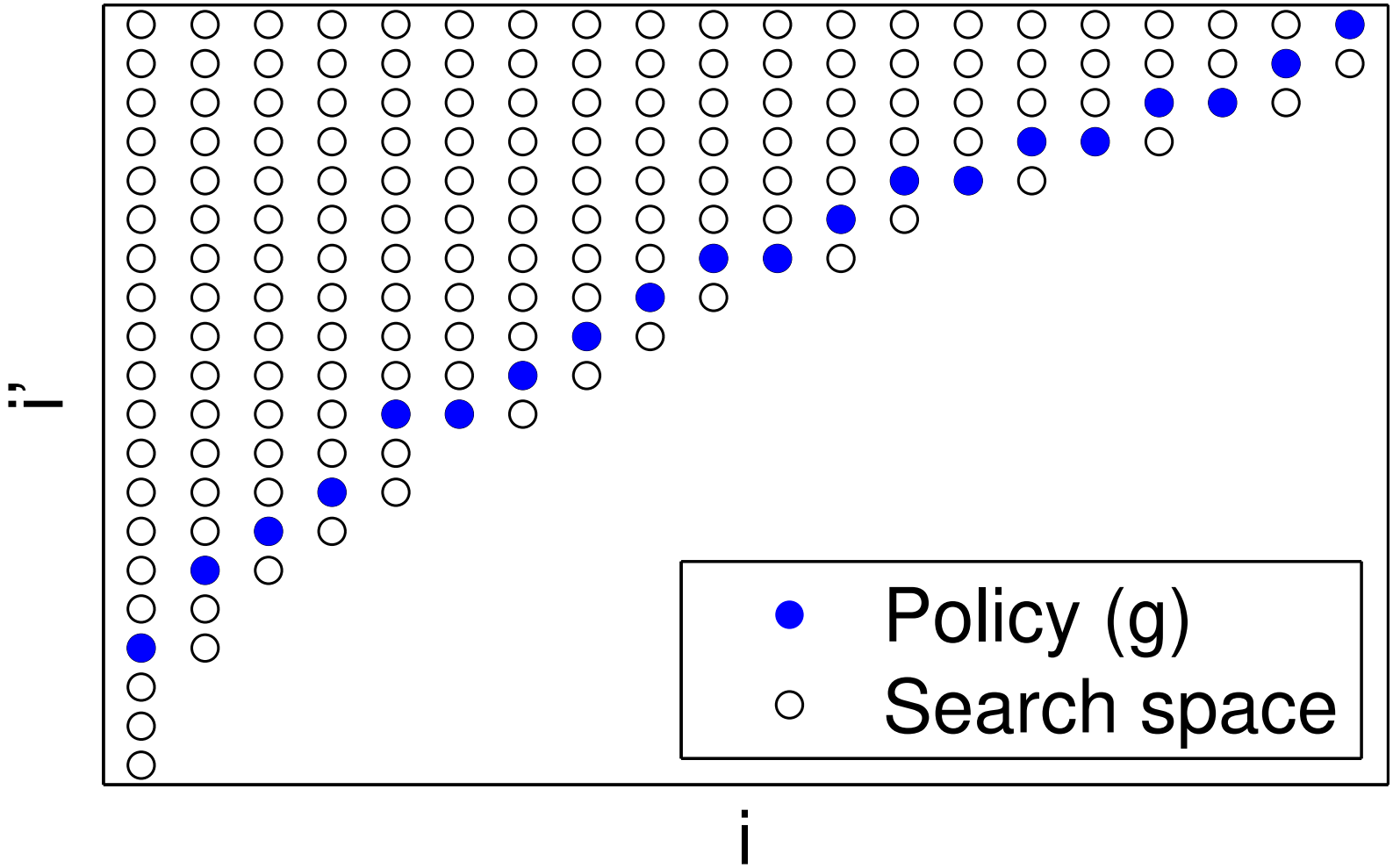
Solving the Bellman

Simple monotonicity code:

```
do iE = 1,nE
  do iA = 1,nA
    ! Set default utility/policies
    V(iA,iE) = -1d100
    c(iA,iE) = -1d0
    api(iA,iE) = 1 ! set to 1 to guarantee monotonicity

    ! Exploit the simple monotonicity condition
    lb = 1
    if (iA>1) lb = api(iA-1,iE)

    ! Loop over iAp restricting the search space
    do iAp = lb,nA
      ! ... same as before
    end do
  end do
end do
```

Two lines, twice as fast!

Graphical illustration



Can we do better than this? Yes, way better.

Moving right to left, have an upper bound not being used.

Binary monotonicity

Suppose $g(\underline{i})$ and $g(\bar{i})$ are known for $\underline{i} < \bar{i}$.

Then for any $i \in \{\underline{i}, \ldots, \bar{i}\}$, $g(i) \in \{g(\underline{i}), \ldots, g(\bar{i})\}$.

Computing $g(m)$ for m the integer midpoint $\lfloor (\underline{i} + \bar{i})/2 \rfloor$, can use it

1. as an upper bound for $g(i)$ with $i \in \{\underline{i} + 1, m - 1\}$
2. as a lower bound for $g(i)$ with $i \in \{m + 1, \bar{i} - 1\}$.

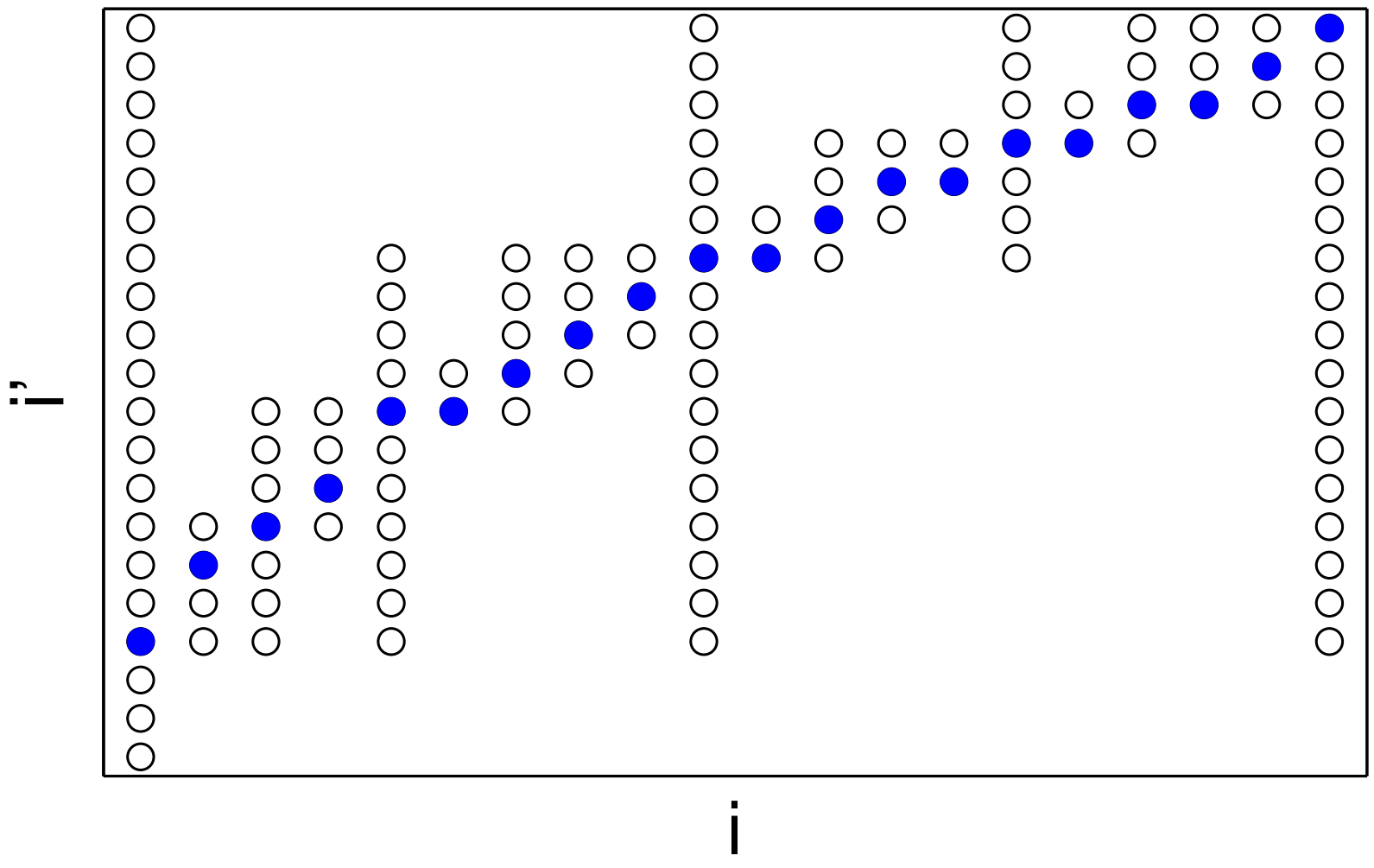
Binary montonicity

This suggests the binary monotonicity algorithm (Gordon and Qiu, 2018).

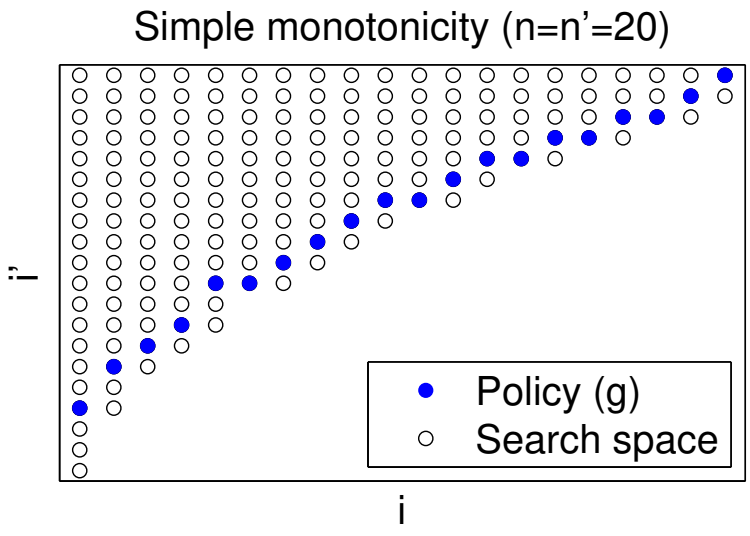
1. Define $\underline{i} = 1$, $\bar{i} = n$.
2. Solve for $g(\underline{i})$ by checking all $i' \in \{1, \ldots, n'\}$.
3. Solve for $g(\bar{i})$ by checking all $i' \in \{g(1), \ldots, n'\}$.
4. If $\bar{i} = \underline{i} + 1$, STOP.
5. Let $m = \lfloor (\underline{i} + \bar{i})/2 \rfloor$ and solve for $g(m)$ checking $i' \in \{g(\underline{i}), \ldots, g(\bar{i})\}$.
6. Go to step 4 twice, once redefining $(\underline{i}, \bar{i}) = (\underline{i}, m)$ and once $(\underline{i}, \bar{i}) = (m, \bar{i})$.

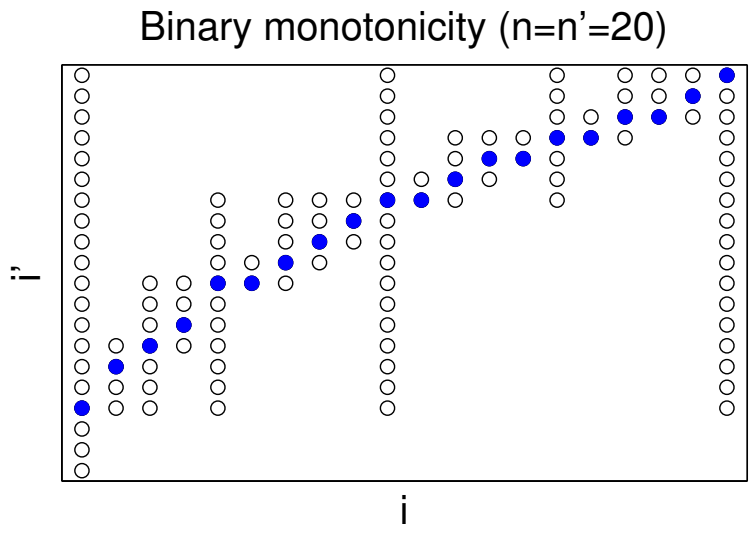
This solves for the entire optimal policy (and value).

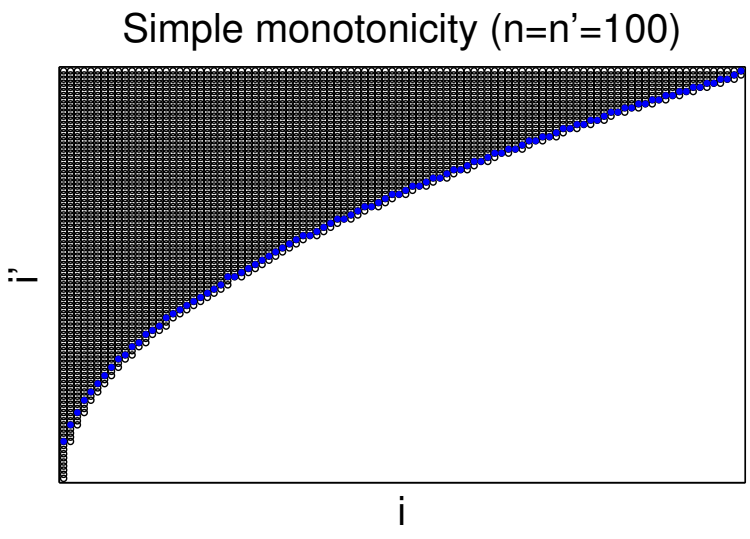
Graphical illustration

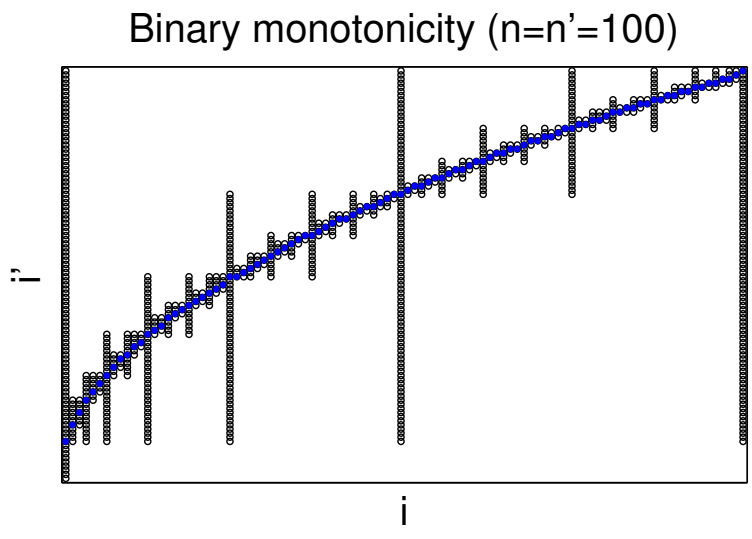


Graphical illustration

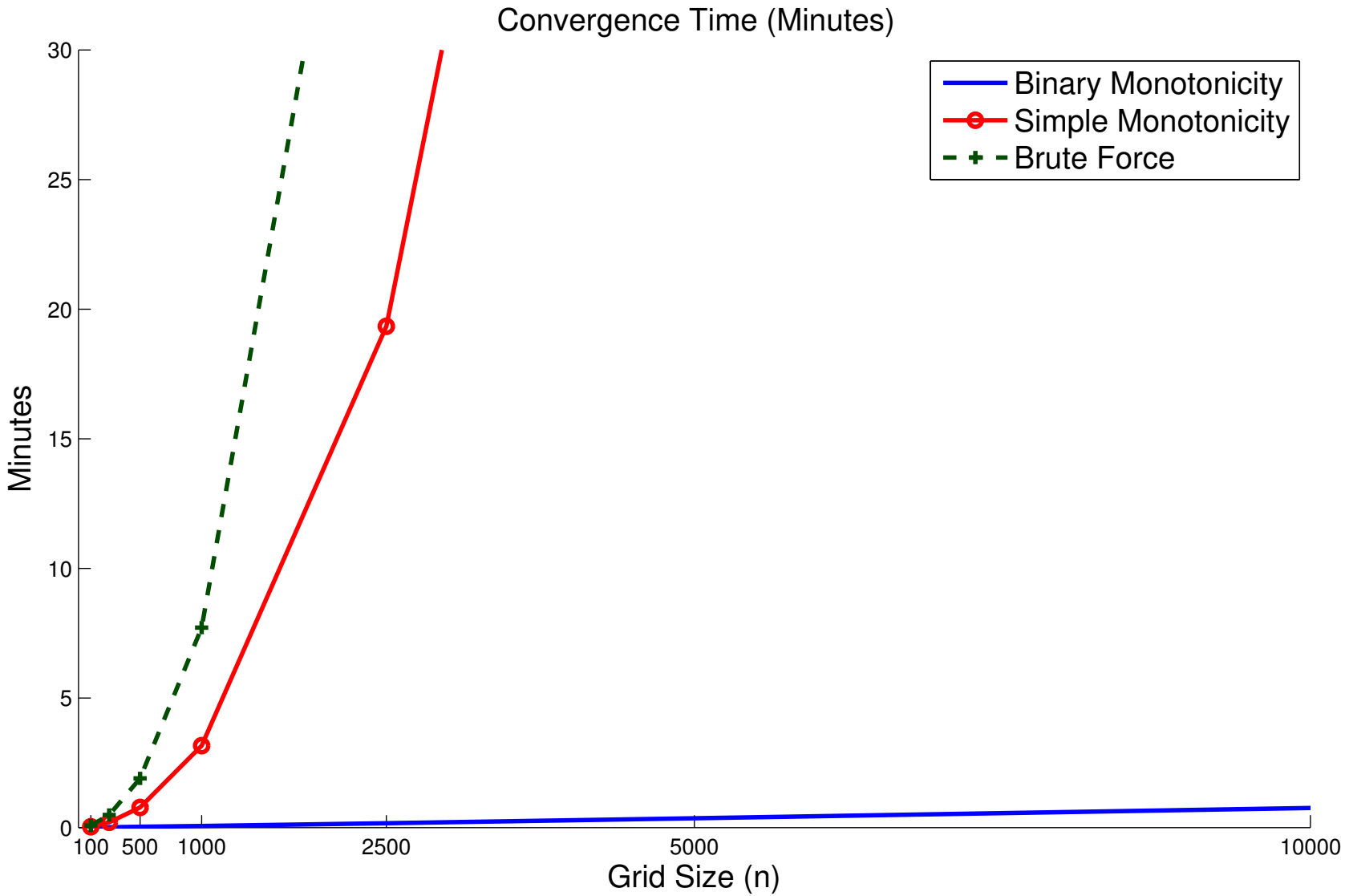








Run-time comparison



Other grid search approaches

Gordon and Qiu (2018) outline the other grid search approaches, including

1. simple concavity
2. binary concavity
3. two-state binary monotonicity

All the monotonicity can be combined with all the concavity and vice-versa.

Sufficient conditions for monotonicity are provided and very generally hold. Basically, need

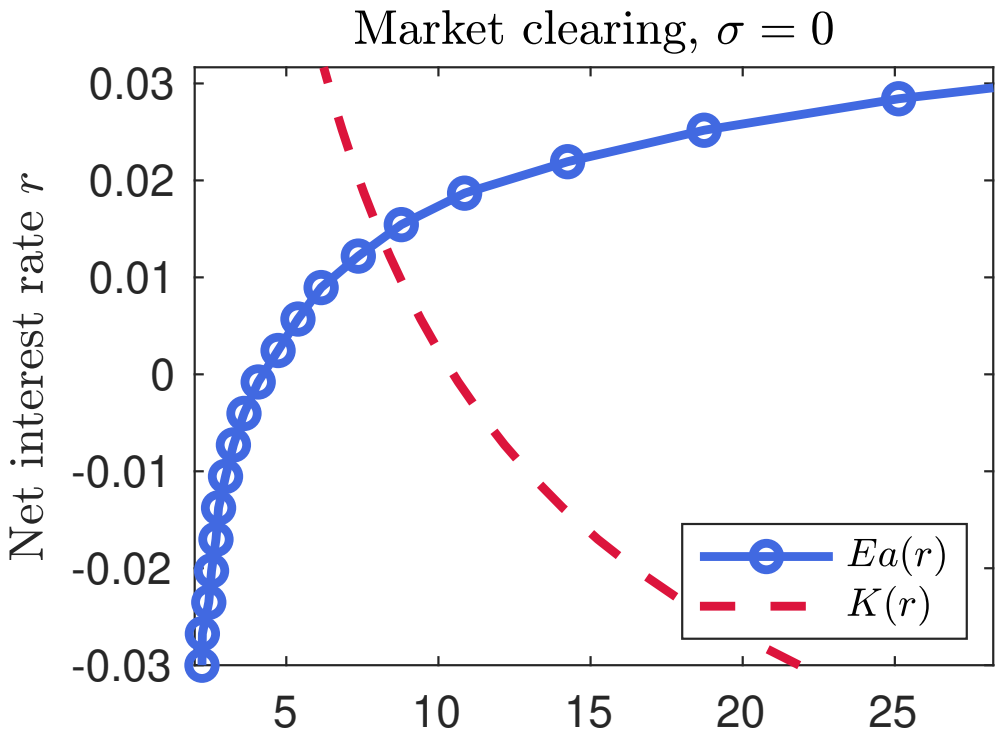
1. increasing continuation utility
2. a concave flow utility function $u(c)$
3. a budget constraint like $c = f(state, control)$ with "$\frac{\partial^2 f}{\partial state \partial control} \geq 0$" (but need not be differentiable).

Convergence

A common problem in rich models is a lack of convergence.

E.g., if you just solve the discretized Aiyagari model in GE, you will find you cannot exactly clear the market.

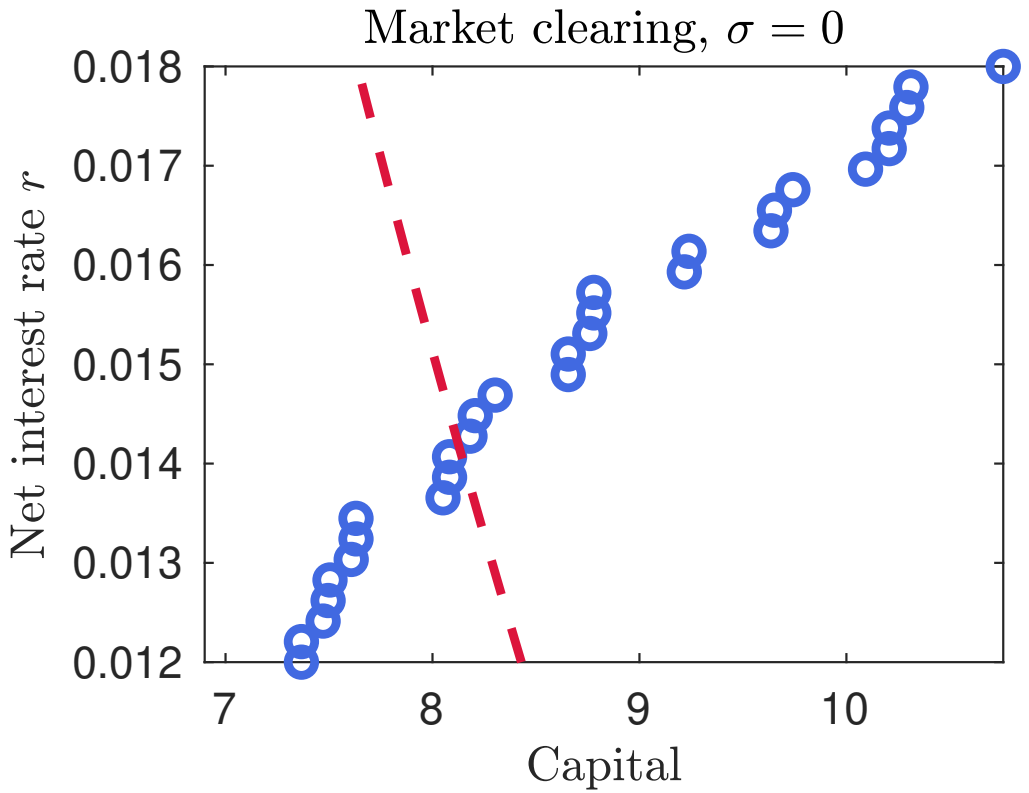
At first blush, it might look continuous:



Convergence

But, the model is discrete and is therefore inherently jumpy.

If you zoom in and don't add interpolated lines,



In this example, it's easily ameliorated by making the approximation finer (more earnings or asset states).

Convergence

For this, let's look at a simple but widely used model, long-term sovereign debt (Hatchondo and Martinez, 2009; Chatterjee and Eyigungor, 2012).

The sovereign's problem may be written

$$V(b,y) = \max_{d\in\{0,1\}} (1-d)V^r(b,y) + dV^d(y)$$

$$V^r(b,y) = \max_{b'\in\mathcal{B}} u(c) + \beta\mathbb{E}_{y'|y} V(b',y')$$

$$\text{s.t. } c = y - q(b',y)(b' - (1-\lambda)b) + (\lambda + (1-\lambda)\kappa)b$$

$$V^d(y) = u(y-\phi(y)) + \mathbb{E}_{y'|y}\left[(1-\xi)V^d(y') + \xi V(0,y')\right]$$

Let the optimal bond policy for the V^r problem be denoted $a(b,y)$.

For a fixed q, one can prove this converges.

Lack of convergence

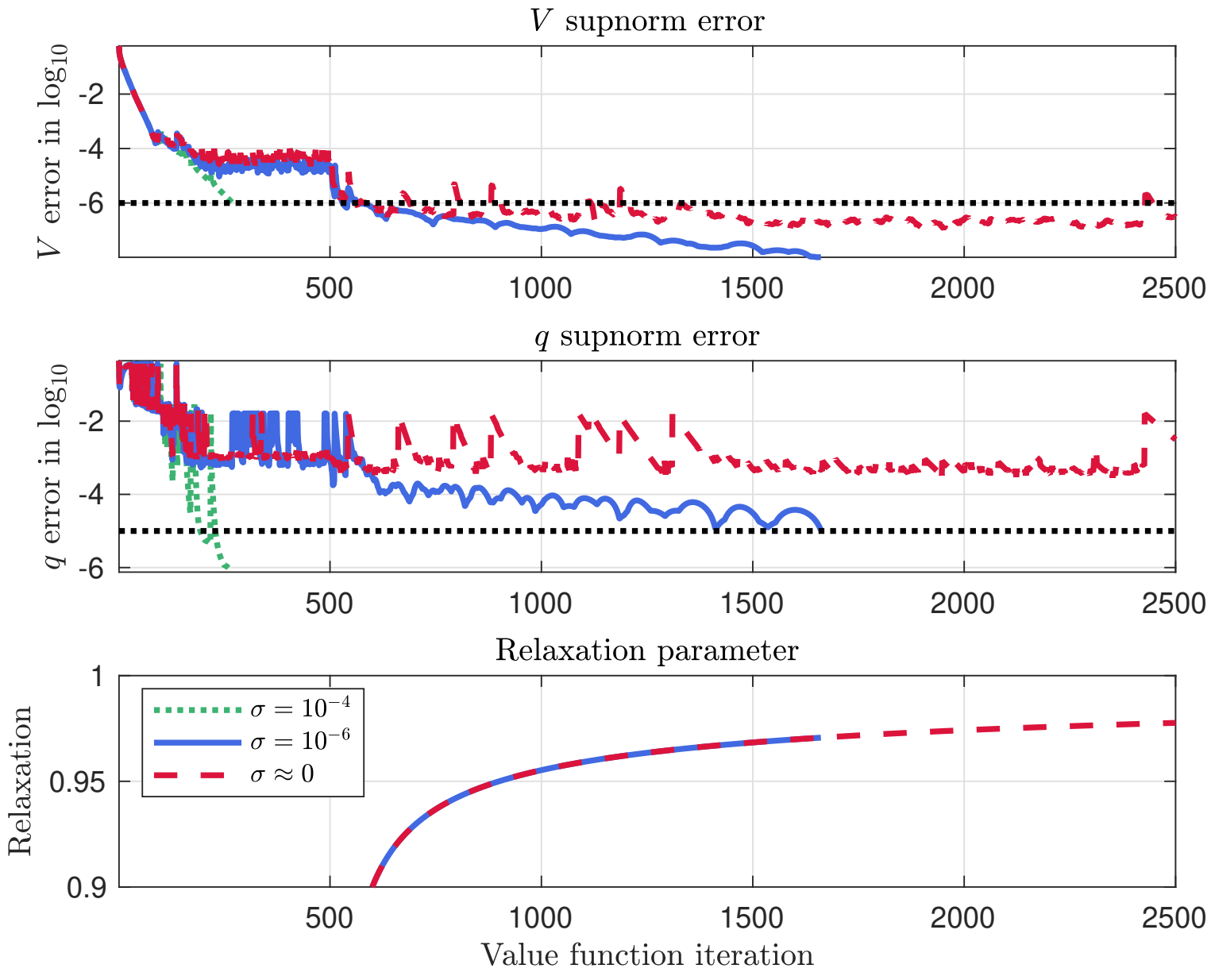
The trouble comes in the debt pricing:

The price schedule q is a solution to $q = T \circ q$ where

$$(T \circ q)(b', y) = \frac{1}{1 + r^*} \mathbb{E}_{y'|y} (1 - d(b', y')) \left(\lambda + (1 - \lambda)(\kappa + q(a(b', y'), y')\right.$$

A small change in q may cause a large jump in $a(\cdot)$, and then a discrete jump in $T \circ q$.

Staring at your computer, frustrated and confused...



Obtaining convergence

What can you do?

- You can add a ton of shocks and make the choice space continuous (Hatchondo and Martinez, 2009)
- You can explicitly add a continuous shock (Chatterjee and Eyigungor, 2012; generalization in Gordon and Guerron-Quintana, 2018)
- You can add taste shocks (Gordon, 2020; Dvorkin et al. 2021)

Adding a continuous shock, while it works, is quite hard. Taste shocks are a simple and intuitive alternative.

To introduce taste shocks, let's go to a more abstract case.

Taste shocks

Fix a state $i \in \{1, \ldots, n\}$ and let $U(i, i')$ denote the "fundamental utility" associated with choice $i' \in \{1, \ldots, n'\}$.

$$\max_{i' \in \{1,\ldots,n'\}} U(i, i') + \sigma \epsilon_{i'}. \tag{4}$$

If the $\epsilon_{i'} \overset{i.i.d.}{\sim} Type - I$ extreme value $\forall i'$, McFadden (1974) showed then the choice probabilities have a closed form expression:

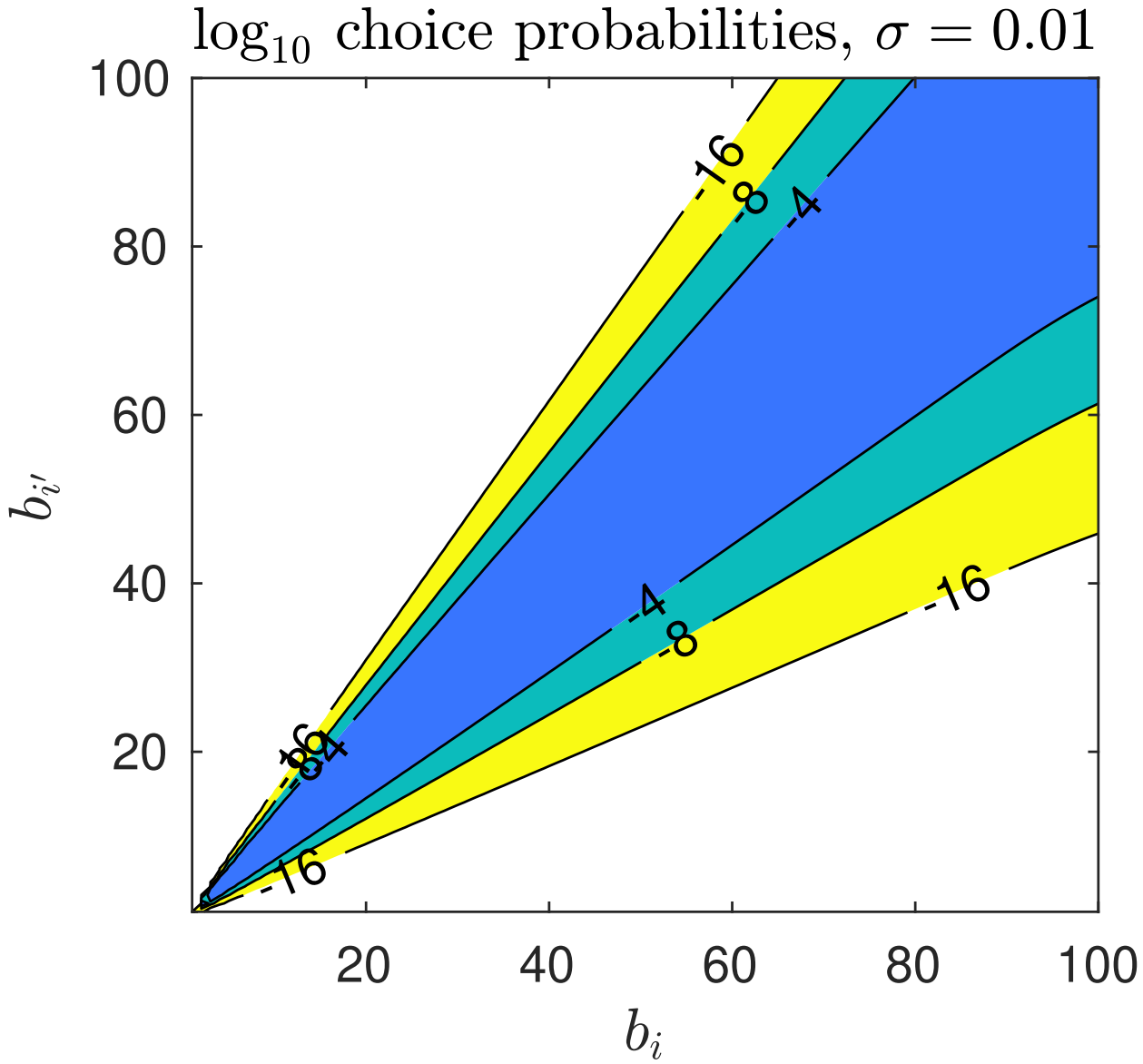
$$\mathbb{P}(i'|i) = \frac{\exp(U(i, i')/\sigma)}{\sum_{j'=1}^{n'} \exp(U(i, j')/\sigma)} \tag{5}$$

The expected value of the maximum is also given in closed form by the "log-sum" formula (Rust, 1987):
$\mathbb{E}\left[\max_{i' \in \{1,\ldots,n'\}} U(i, i') + \sigma \epsilon_{i'}\right] = \sigma \log\left(\sum_{i'=1}^{n'} \exp(U(i, i')/\sigma)\right).$

(Note: As n' grows, this expectation tends to ∞.)

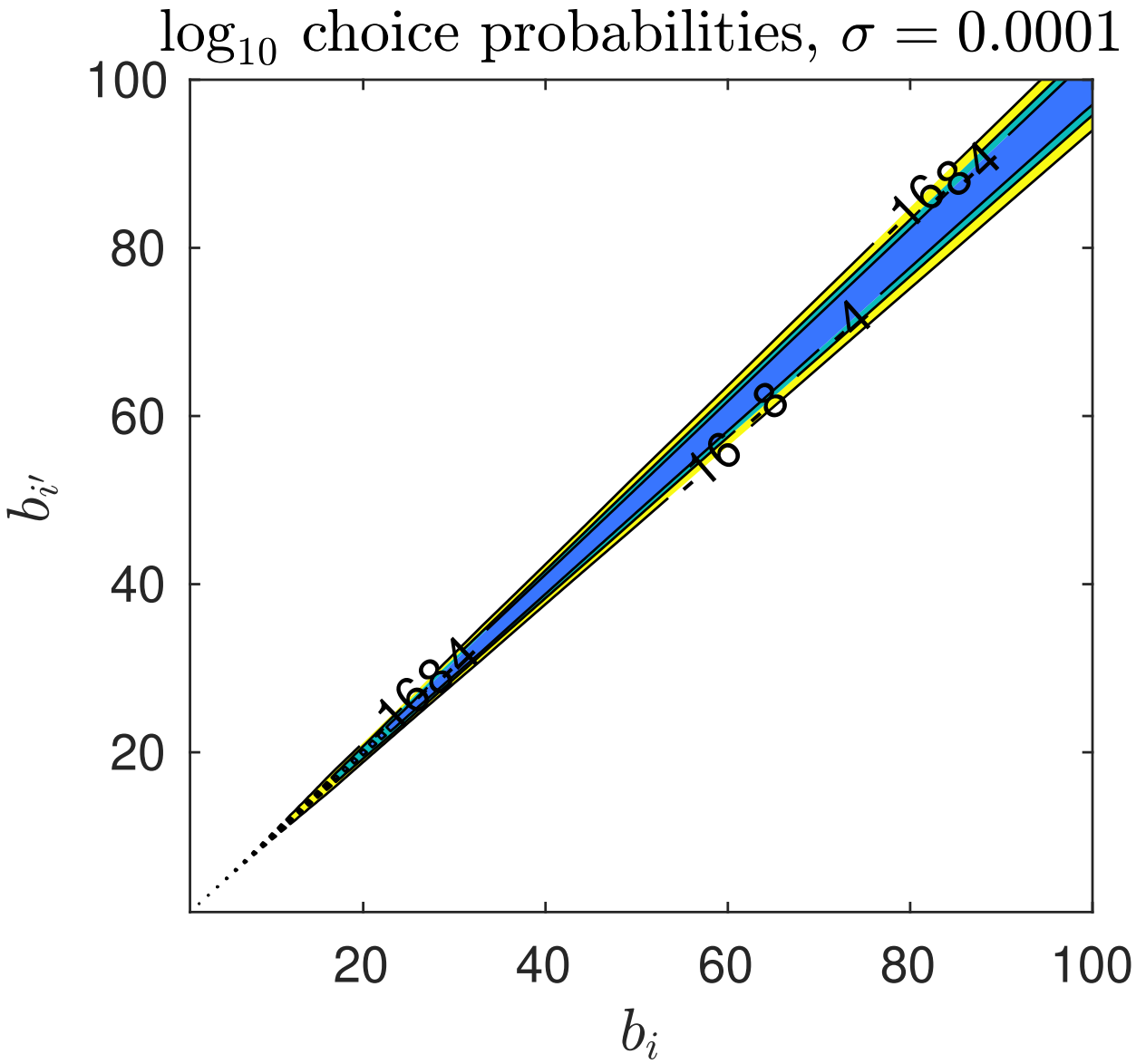
Taste shocks example

Here is $\mathbb{P}(i'|i)$ for $U(i,i') = \log(b_i - b_{i'}/2) + \log(b_{i'})$:



Taste shocks example

Here is $\mathbb{P}(i'|i)$ for $U(i,i') = \log(b_i - b_{i'}/2) + \log(b_{i'})$:



Back to sovereign debt

Reconsider the sovereign repayment problem, but add taste shocks:

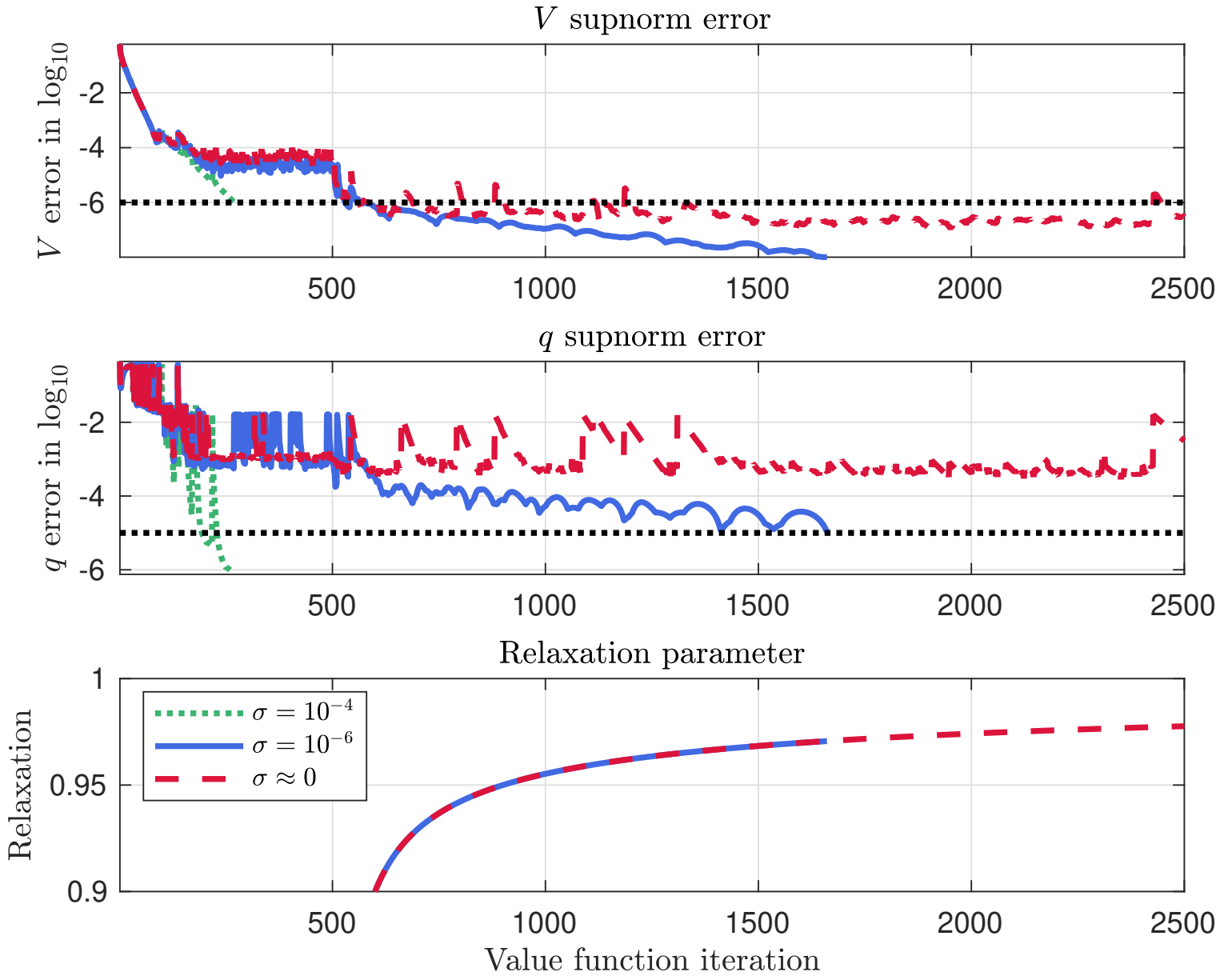
$$V^r(i,y) = \mathbb{E}_\epsilon \left[\max_{i' \in \{1,\ldots,n\}} u(c) + \beta \mathbb{E}_{y'|y} V(i',y') + \sigma \epsilon_{i'} \right]$$
$$\text{s.t. } c = y - q(b_{i'},y)(b_{i'} - (1-\lambda)b_i) + (\lambda + (1-\lambda)z)b_i$$

The price-schedule update is now

$$(T \circ q)(i',y) = \frac{1}{1+r^*} \mathbb{E}_{y'|y} \left[(1 - d(i',y')) \left(\lambda + (1-\lambda)(z + \sum_{i''} \mathbb{P}(i''|i',y') q(i'',y') \right) \right].$$

Now, because $\mathbb{P}(i'|i,y)$ moves continuously in q, $T \circ q$ moves continuously in q (except for jumps induced by d which, like in one-period debt, are not a significant problem).

Frustration resolved



Note: Gordon (2021) suggests a way to speed up when using taste

Practical note about implementation

Many people get stuck using taste shocks because they naively apply

$$\mathbb{P}(i'|i) = \frac{\exp(U(i,i')/\sigma)}{\sum_{j'=1}^{n'} \exp(U(i,j')/\sigma)}$$

If σ is small or U is large, you will have numbers that cause an arithmetic overflow on the computer: $exp(U/\sigma) = NaN$.

What you ought to do is exploit

$$\mathbb{P}(i'|i) = \frac{\exp((U(i,i') - U^*(i))/\sigma)}{\sum_{j'=1}^{n'} \exp((U(i,j') - U^*(i))/\sigma)}$$

where U^* is the maximum over i' for a given U. Then, at most you'll be exponentiating 0. You can modify the log-sum formula as well.

The underflows are not usually a problem, but you can use $\max\{\delta, (U - U^*)/\sigma\}$ instead of $(U - U^*)/\sigma$ for δ sufficiently small (say -200).

Calibration and optimization

Calibration

Most calibrations are $\approx$ generalized method of moments (GMM).

GMM assumes the existence of parameters θ^* such that a model's moments $m(\theta)$ match target moments m^*, i.e., $m(\theta^*) = m^*$. The parameters θ^* are identified if $m(\theta) \neq m^*$ for all $\theta \neq \theta^*$.

If this is the case, θ^* is found as the solution of

$$\min_{\theta \in \Theta} (m(\theta) - m^*)' W (m(\theta) - m^*) \tag{6}$$

where W is a positive definite weighting matrix (often diagonal).

Note:

- Can always "identify" a θ from (6), but does $m(\theta) = m^*$?
- "Actual" GMM has $m(\theta)$ noisy, and constructs confidence intervals and hypothesis tests (such as $m(\theta) = m^*$).
- In quantitative models, one often works with the population moments—so $m(\theta)$ is not noisy, CIs should be 0, and the J-stat should be 0.

Calibration

Our intuition will often inform the moment choices.

E.g., we'll reason β affects debt, default costs χ affect interest rates, etc. and choose moments based on that.

That may well be the case. But that rationale is really about identification.

Existence is important: If $m(\theta)$ is really far from m^*, a referee may reject the paper ("the model can't explain x,y,z").

Calibration

How do you know if you have existence?

Method 1: solve (6) and check if $m(\theta) \approx m^*$.

This mostly works, but has a couple disadvantages:

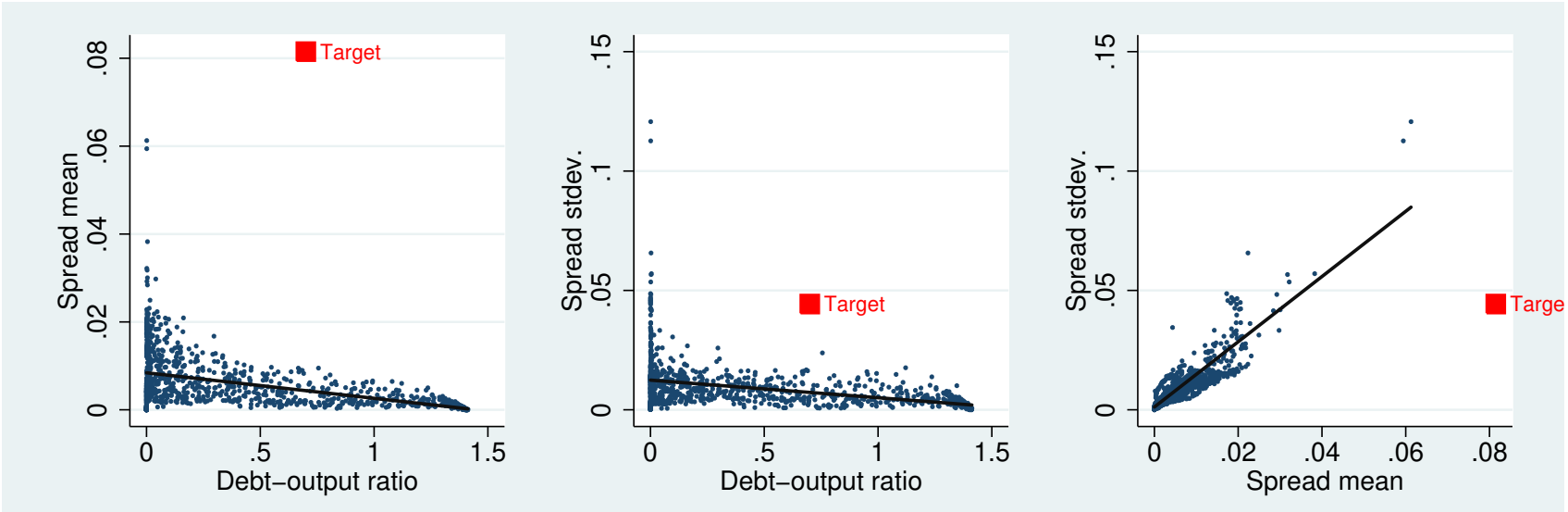
- Solving (6) can take a long time
- Solving (6) is error prone: A failure to match may reflect a poor minimization algorithm, not a model failure.

Method 2: generate data from $m(\theta)$ for a bunch of θ draws and check if m^* looks like an outlier wrt $\{m(\theta)\}$.

- Far faster and far less error prone
- Aids in debugging (investigate strange patterns / param.)
- Can check multiple different calibration strategies at the same time. (E.g., should you match debt and spreads, or debt and correlation between output and spreads?)
- Can check if an over-identified strategy would work (more precisely pinning down θ)

Calibration

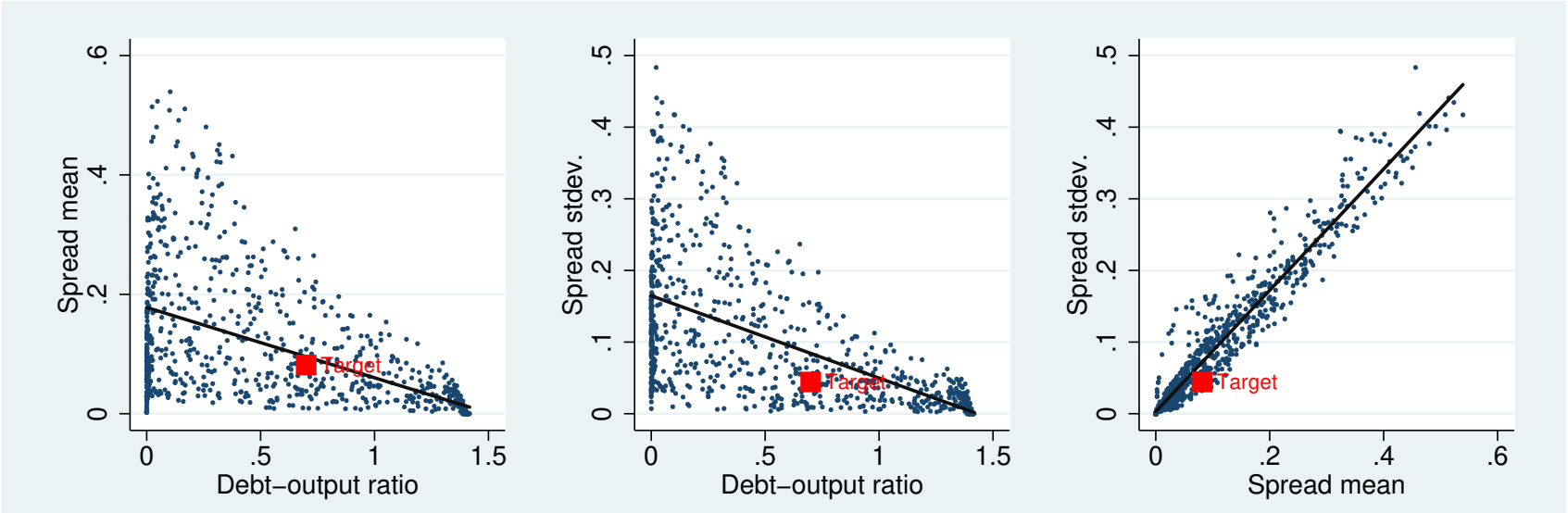
Here's an example from short-term sov. debt of non-existence:



- The blue dots are $m(\theta)$ values for random draws of θ from the parameter space
- The model can reproduce the debt-output ratio and the spread stdev. But it cannot reproduce them at the same time (middle panel).
- The model cannot really reproduce the spread mean
- The model has a tight linkage between the spread mean and stdev., and the data is far from that.

Calibration

In contrast, with long-term sovereign debt, existence seems to hold:



- For each of the moment combinations, existence holds.
- But, can it match all three at the same time? Can't tell here.

This exercise is a "predictive prior" exercise with flat priors.

And just by plotting a few scatter plots, it's pretty obvious whether the model has a chance or not.

Calibration

Cao and Gordon (2019) formalize the process of checking whether θ such that $m(\theta)$ can possibly generate m^* by casting the problem into an outlier detection.

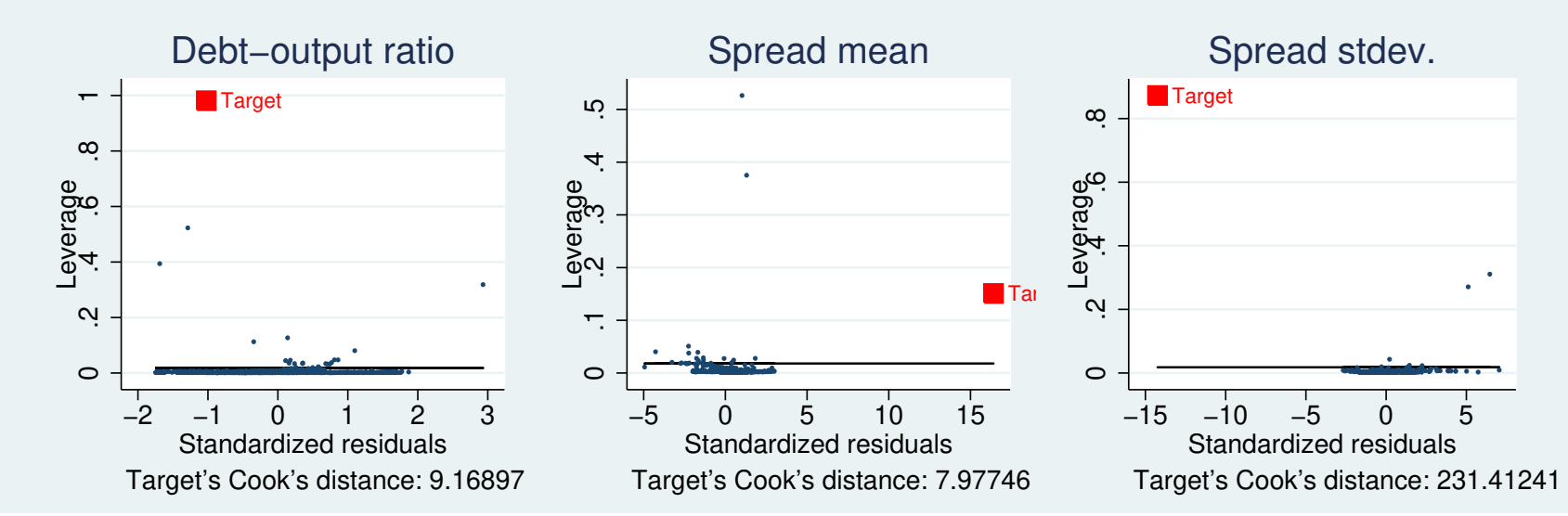
Given sample points $\{m(\theta_i)\}_i$, is m^* an outlier?

If it is an outlier, you can either adjust the targeted moments or rework / debug the model.

There are a bunch of tests one can run to check for outliers.

Leverage and Cook's Distance

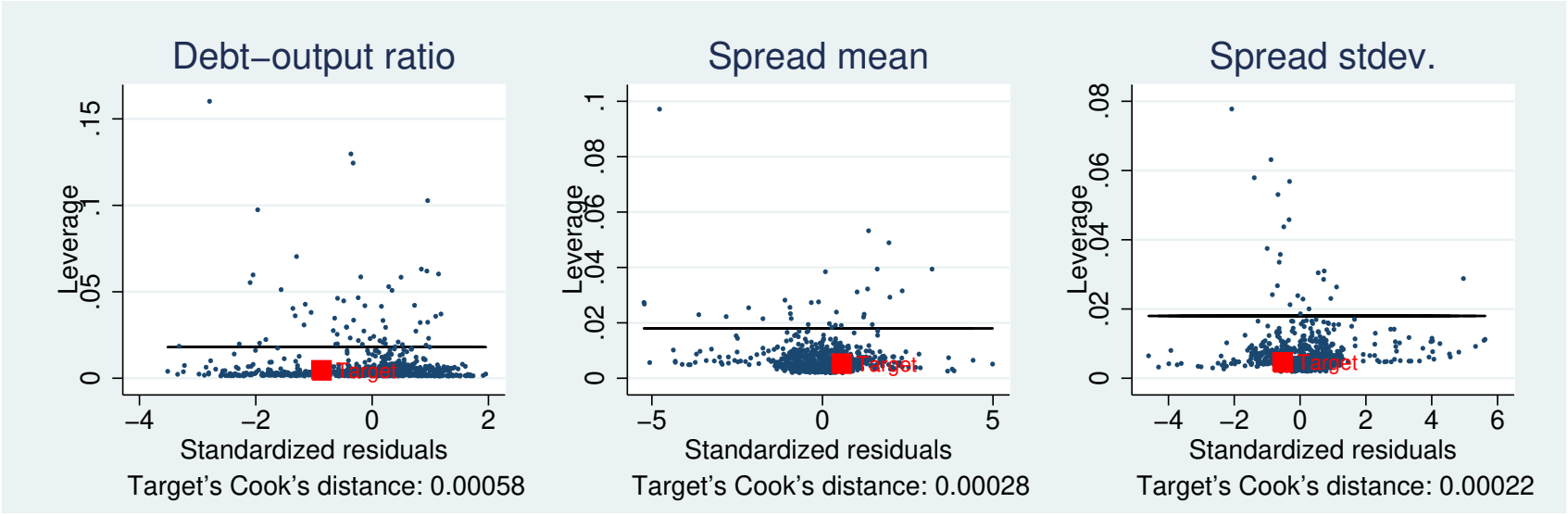
One way is to specify a regression where component d of $m(\theta_i)$ is regressed on the other components using a flexible specification.



When evaluating the m^* as a data point in that regression, does the regression have large leverage or Cook's distance?

Leverage and Cook's Distance

One way is to specify a regression where component d of $m(\theta_i)$ is regressed on the other components using a flexible specification.



When evaluating the m^* as a data point in that regression, does the regression have large leverage or Cook's distance?

Mahalanobis distance

If one assumes $\{m\} \sim N(\mu, \Sigma)$ follows a multivariate normal, then the Mahalanobis distance

$$d(m) = \left((m-\mu)'\Sigma^{-1}(m-\mu)\right)^{1/2}$$

has $d(m)^2$ is distributed $\chi^2_{dim(m)}$.

So,

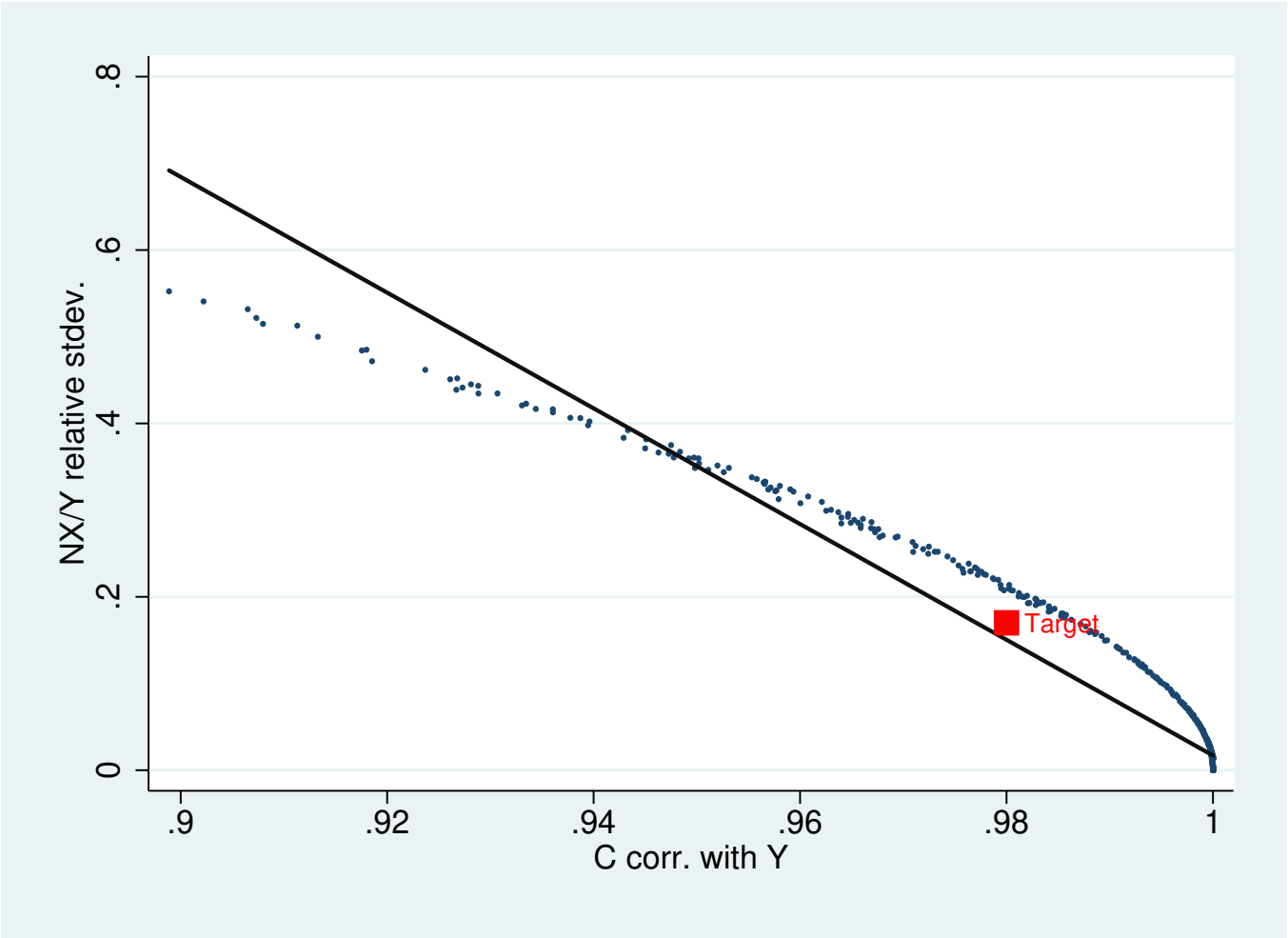
$$Pr(d(m) \geq d(m^*)) = Pr(d(m)^2 \geq d(m^*)^2) = 1 - \chi^2_{dim(m)}(m^*).$$

Hence, the probability of encountering an observation with greater Mahalanobis distance than the target moments' distance is $1 - \chi^2_{dim(m)}(m^*)$, the "M-test" value.

For our example, long-term (short-term) debt has a M-test value of 0.59 (6×10^{-63}).

Mahalanobis distance

The M-test can be tricked:



How can we detect this type of case?

Local outlier factors

In the previous example, locally about the target there's no points.

In contrast, the simulated points usually had a fair number of points close to them.

A test comparing the empirical "density" at one value with a typical empirical density could detect this.

One approach that does this is local outlier factors (LOFs, Bruenig et al. 2000).

(Note: the ideas are connected to very old ideas in Loftsgaarden and Quesenberry, 1965.)

Local outlier factors

Consider some set X and some metric $d(.,.)$.

Let $N_k(x)$ denote the k-nearest neighbors (kNN) of $x \in X$.

Define the "k-distance" as $\epsilon_k(x) = \max_{o \in N_k(x)} d(x, o)$, which is the radius of the smallest ball containing the kNN of x.

A "reachability distance" from x to some point p is defined as $r_k(x, p) = \max\{\epsilon_k(p), d(x, p)\}$.

The "local reachability density" is defined as

$$lrd_k(x) = (\frac{1}{k} \sum_{p \in N_k(x)} r_k(x, p))^{-1}.$$

A relatively high value means x can be easily reached its kNN.

The LOF is defined as

$$LOF_k(x) = \frac{1}{k} \sum_{p \in N_k(x)} \frac{lrd_k(p)}{lrd_k(x)}.$$

Local outlier factors

The LOF is defined as

$$LOF_k(x) = \frac{1}{k} \sum_{p \in N_k(x)} \frac{lrd_k(p)}{lrd_k(x)}.$$

If LOF_k is large, x is further from its neighbors than x's neighbors are from their neighbors.

An LOF_k value close to or less than 1 indicates that x is not an outlier while a value much greater than 1 suggests x is an outlier.

This method is non-parametric and local, and is related to an actual density estimate (see the discussion in Cao and Gordon, 2019).

For long-term debt, the LOF is in $[1.02, 1.05]$ for $k \in \{5, 20, 40\}$.

For short-term debt, the LOF varies from $[16.6, 32.1]$.

Checking multiple strategies at one time

Using one of simulated moments, you can check not just one "strategy" (e.g., match $b/y, r, \sigma(r)$) but many strategies all at the same time.

E.g., here are some of the best and worst when taking combinations of moments:

Long-term debt, best

Moments	M-test	LOF value (pctile)
μ_b,μ_r,ρ_r	0.89	1.14 (83.7)
$\mu_r,\sigma_c,\sigma_{nx}$	0.86	1.05 (52.1)
μ_b,σ_r,ρ_r	0.85	1.08 (74.2)

Long-term debt, worst

Moments	M-test	LOF value (pctile)
$\sigma_{nx},\rho_c,\rho_r$	3×10^{-4}	2.03 (98.8)
$\sigma_r,\sigma_{nx},\rho_c$	3×10^{-4}	1.94 (99.4)
$\sigma_{nx},\rho_c,\rho_{nx}$	7×10^{-5}	2.92 (100)

Short-term debt, best

Moments	M-test	LOF value (pctile)
μ_b,σ_{nx},ρ_c	0.95	2.30 (100)
$\mu_b,\sigma_c,\sigma_{nx}$	0.21	2.42 (99.9)
μ_b,σ_c,ρ_r	0.18	3.01 (99.9)

Short-term debt, worst

Moments	M-test	LOF value (pctile)
μ_r,σ_r,σ_c	1×10^{-57}	19.1 (100)
μ_b,μ_r,σ_r	6×10^{-63}	21.6 (100)
μ_r,σ_r,ρ_{nx}	2×10^{-63}	21.3 (100)

BTW: We have made code available for doing these tests.

Optimization

At some point, you do have to minimize the objective function.

What algorithms should you use?

This is a general question that goes beyond calibration to very many economic problems.

Optimization

A quite general optimization problem is

$$\begin{aligned} & \min_{x \in \mathbb{R}^n} f(x) \\ \text{s.t. } & g_i(x) = 0 \\ & h_j(x) \leq 0 \end{aligned}$$

for $f, g_i, h_j : \mathbb{R}^n \to \mathbb{R} \; \forall \; i, j$.

Note: usually the computational literature focuses on function minimization. However, all the methods can be directly applied to maximization since $\arg\min f = \arg\max -f$.

Optimization

There are three classes of optimization methods:

1. Comparison methods: no derivatives required
2. Methods using gradients: C^1 required.
3. Methods using gradients and Hessian: C^2 required.

Note: NLOPT `https://nlopt.readthedocs.io/en/latest/` has a ton of routines implemented and is callable from $\approx$ everything.

Optimization

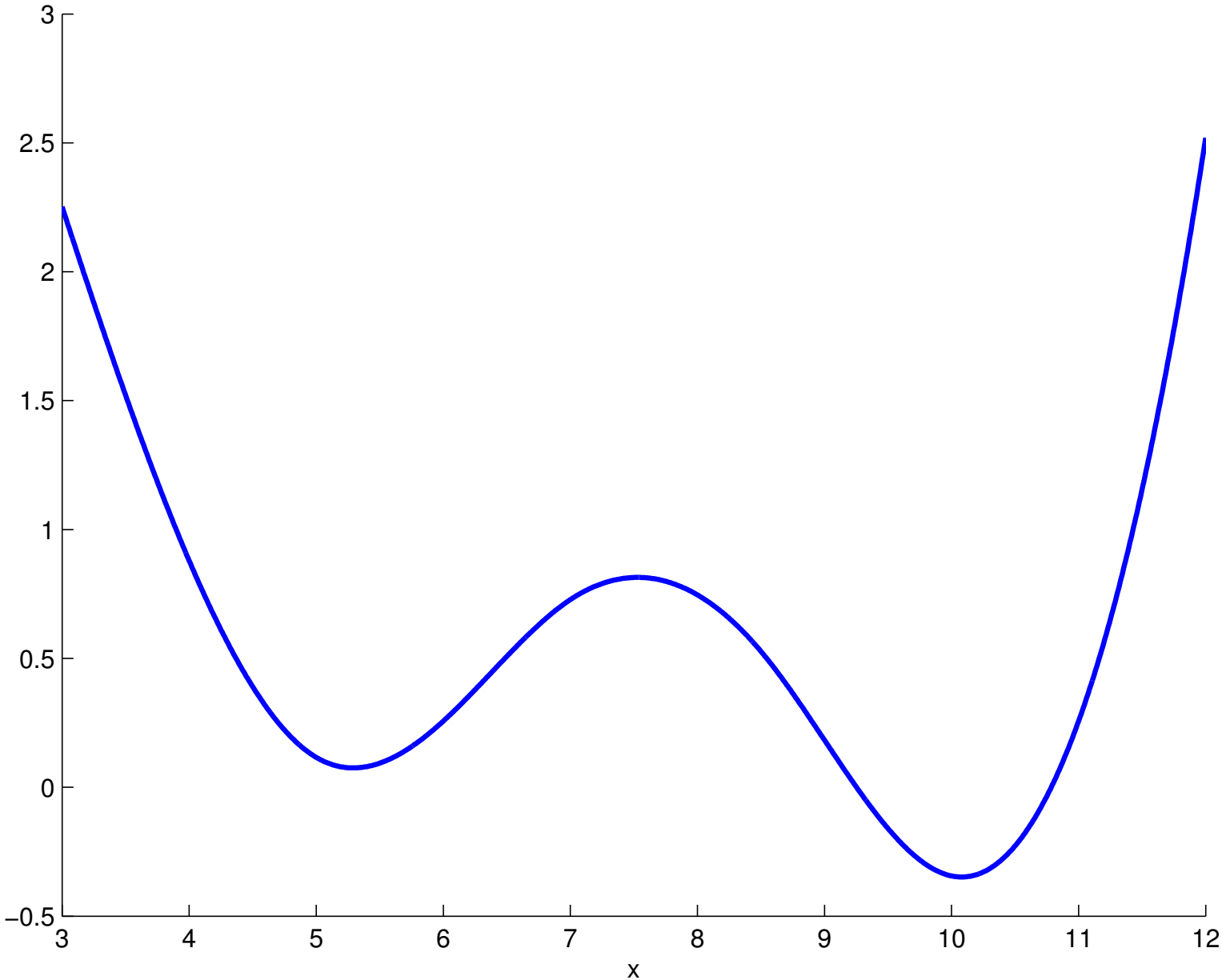
First, consider the simplest possible case, one-dimensional minimization with no constraints:

$$\min_{x \in \mathbb{R}} f(x)$$

In one-dimension, there are special methods available.

Consider the following function.

Bracketing a minimum



Bracketing a minimum

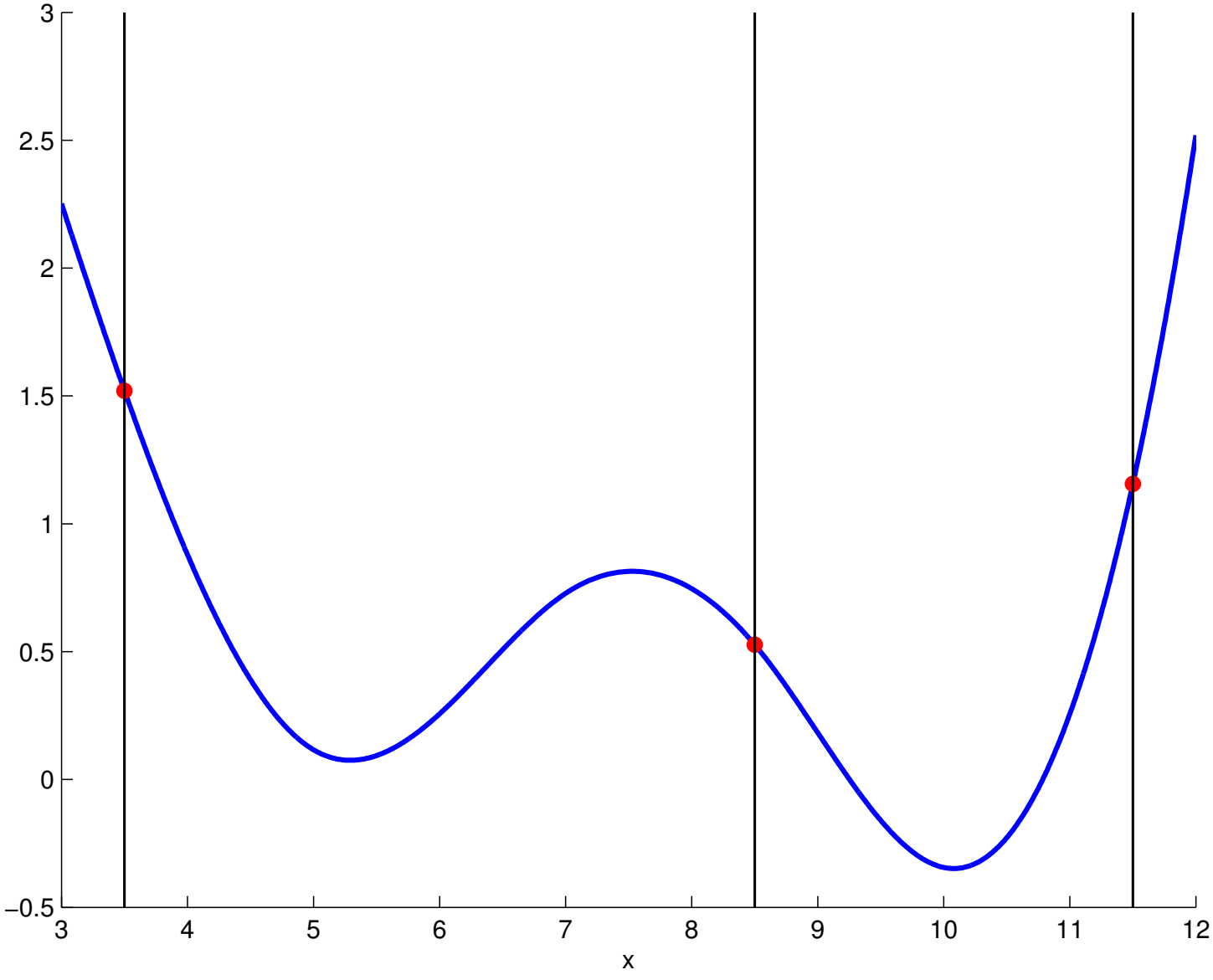
Suppose we have three points satisfying

- $a < b < c$
- $f(b) \leq f(a)$
- $f(b) \leq f(c)$

Then, if f is a C^0 function, one must have a minimum in $[a, c]$.

Consequently, a, b, c are said to bracket a minimum.

Bracketing a minimum



Bracketing a minimum

To find a minimum, we need to shrink the bracket a, b, c (i.e. decrease $c - a$).

How can this be done?

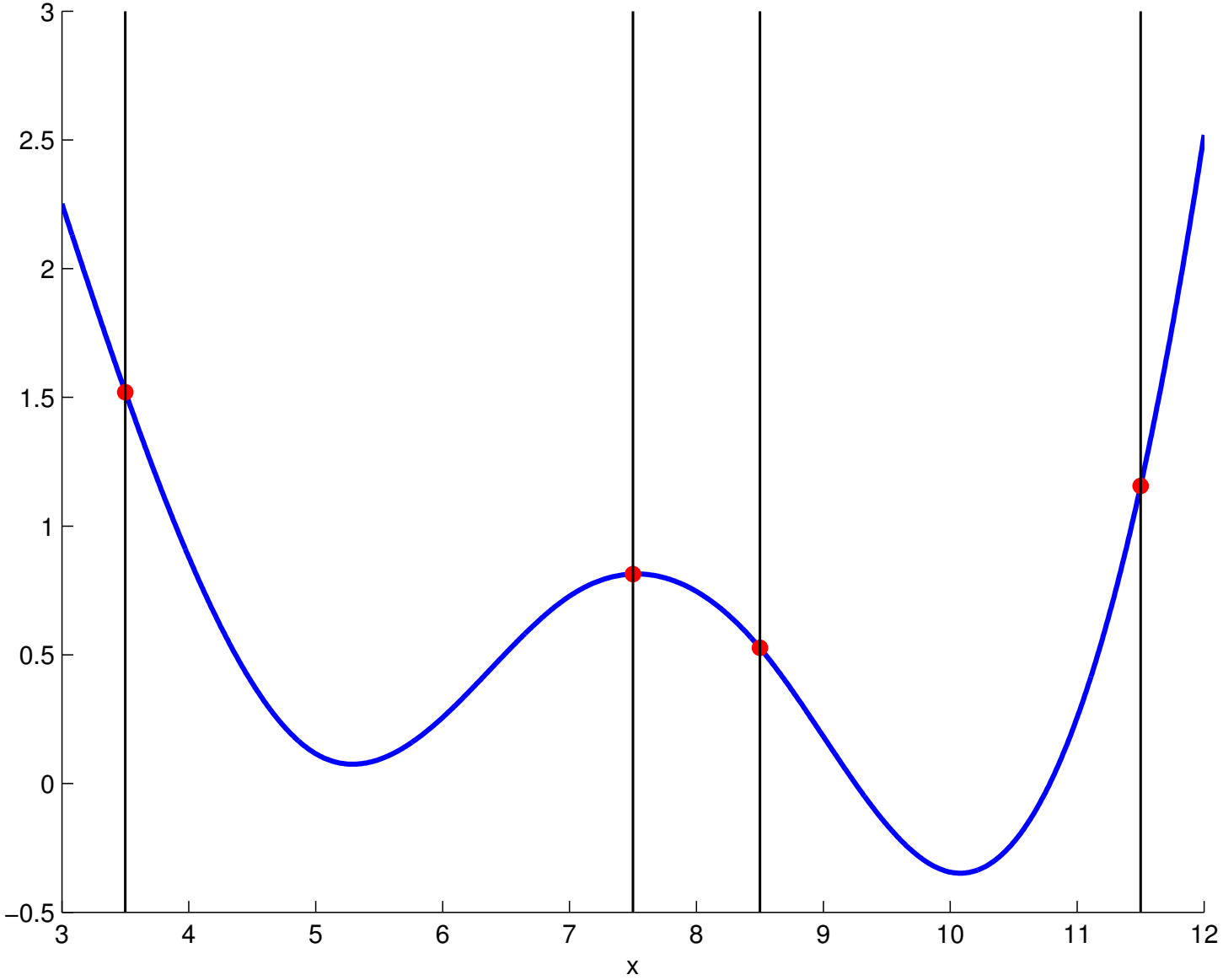
- Insert a point d in (a, c) with $d \neq b$.
- If you chose $d < b$, then you new bracket will either be (a, d, b) or (d, b, c).
- If you chose $d > b$, then you new bracket will either be (a, b, d) or (b, d, c).

Consider an example.

Suppose you chose $d = (a + c)/2$. How would you get a new bracket?

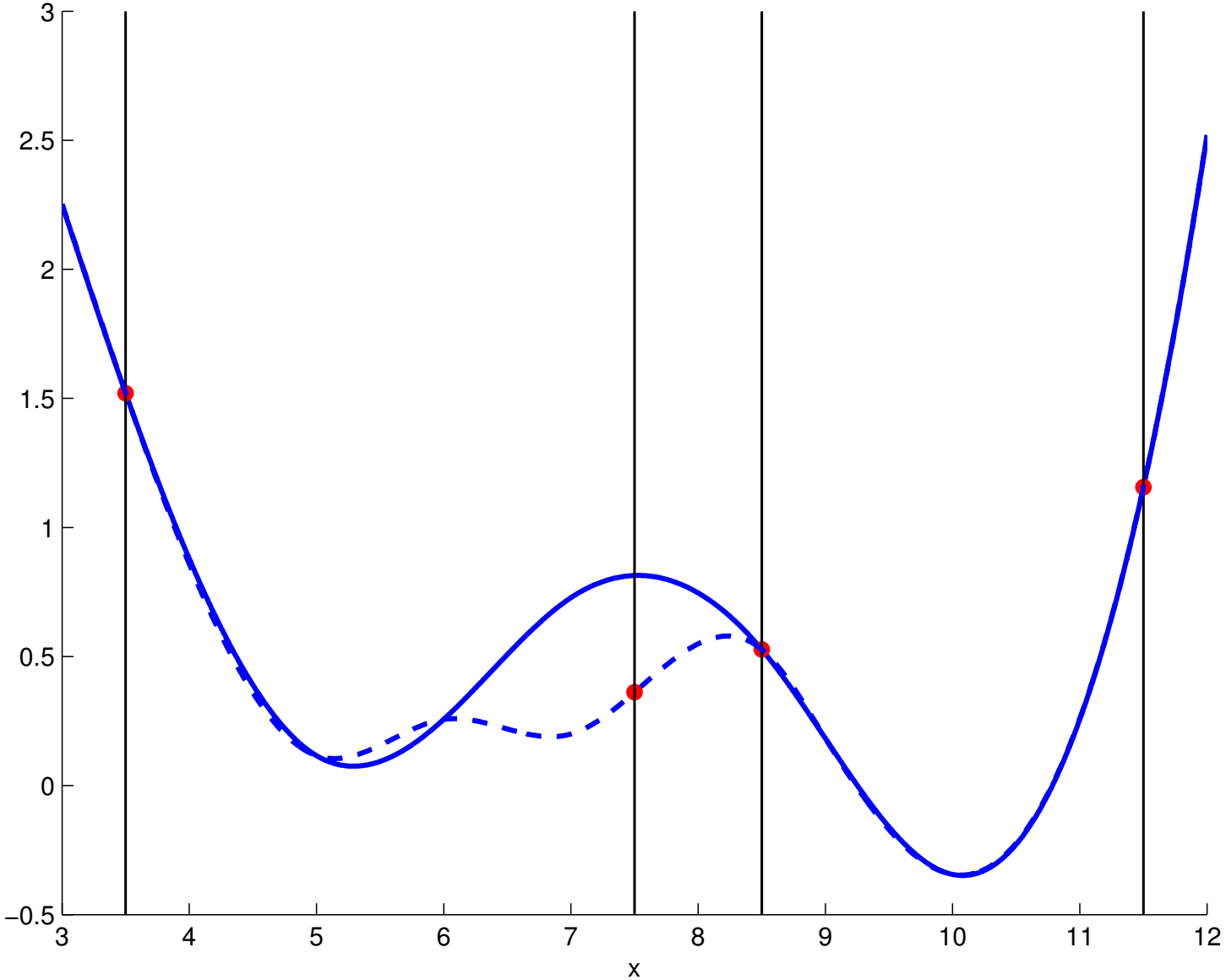
Bracketing a minimum

If $f(d) > f(b)$, then your new bracket is d, b, c.



Bracketing a minimum

If, however, $f(d) < f(b)$, then your new bracket is a, d, b.

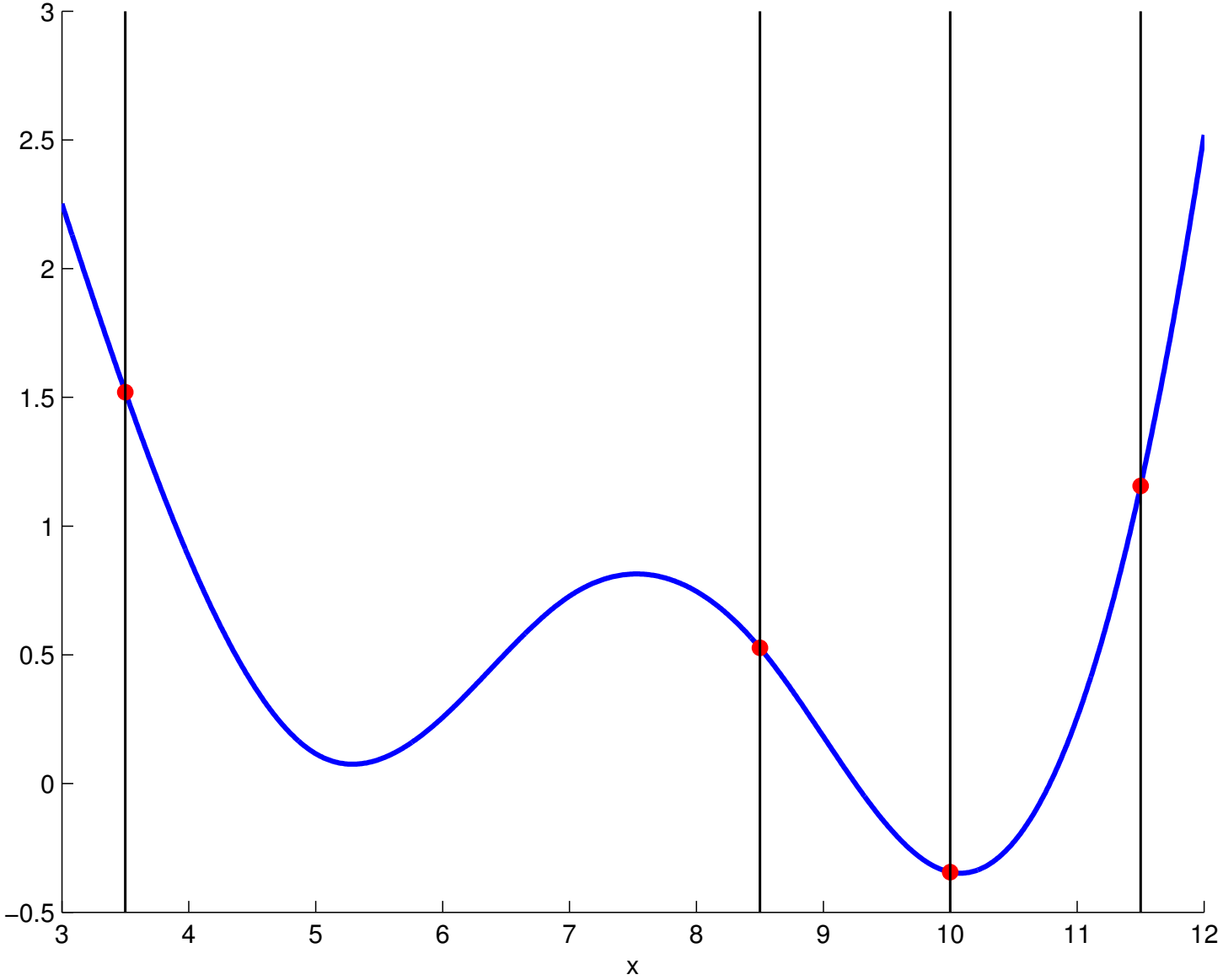


Bracketing a minimum

Now, suppose instead you chose $d = (b + c)/2$. How would you get a new bracket?

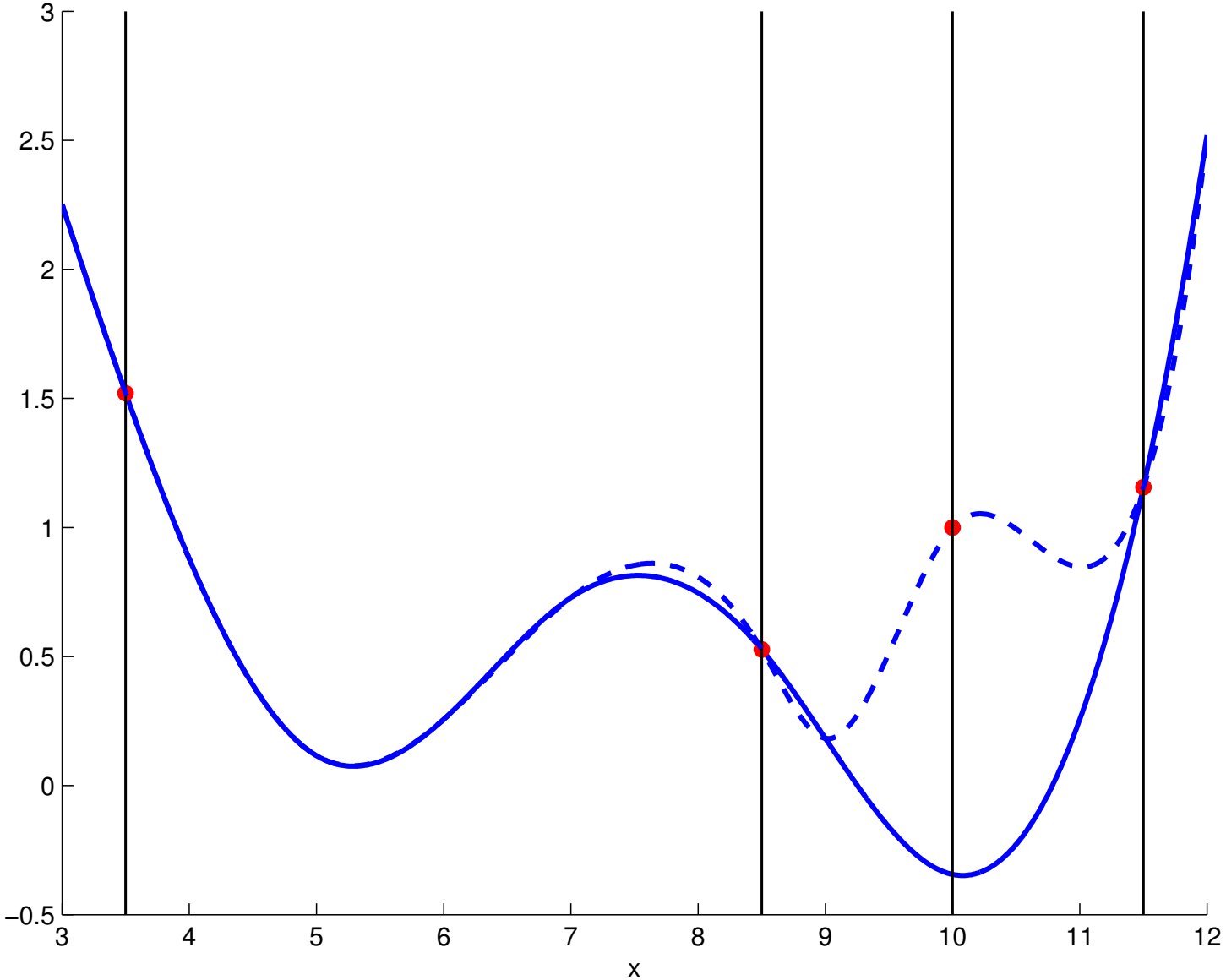
Bracketing a minimum

If $f(d) < f(b)$, then your new bracket is b, d, c.



Bracketing a minimum

If, however, $f(d) > f(b)$, then your new bracket is a, b, d.



Bracketing a minimum

So, it is clear how one gets a new bracket for a given d.

But what should d be?

A very conservative approach is to try to minimize the worst-case error (in terms of shrinking bracket sizes)

Bracketing a minimum

Here are the details for minimizing the worst-case error:

- If $d < b$, then the possible brackets are (a, d, b) and (d, b, c) with worst-case error $\max\{b - a, c - d\}$. Ignoring $d < b$, this is minimized by a d such that $b - a = c - d$ or $d = a + (c - b)$.
- If $d > b$, then the possible brackets are (a, b, d) and (b, d, c) with worst-case error $\max\{d - a, c - b\}$. Ignoring $d > b$, this is minimized by a d such that $d - a = c - b$ or $d = a + (c - b)$.
- Since these agree, d should always be $a + (c - b)$.

Note d always goes in the larger interval: if $b < (c + a)/2$ then $d > b$ and if $b > (c + a)/2$ then $d < b$.

This selection of $d^* = a + (c - b)$ is called golden-section search.

Bracketing a minimum

How do we find an initial bracket?

Often theory can provide (demand 0 or ∞ at ...)

If not, you can flop around potentially until you find one.

A general strategy to keep in mind is this:

- If you need to find j such that $x \in [x_j, x_{j+1})$ for some list $\{x_i\}_{i=0}^n$, you can do this in $O(\log n)$ time.
- How? Subdivide the space in half repeatedly. (Binary search.)
- In reverse, if you want to find a δ such that a point x has $x \pm \delta$ bracketing a minimum of f, you can double δ every time.
- "Equivalently," you can search a huge space using linear steps if you work with $\log x$ instead.

Brent's method

Golden section search chose intervals to minimize worst-case behavior. This is natural, but unnecessarily conservative.

Often Brent's method is faster.

In the interest of time, I'll refer you to my lecture notes on the subject.

Minimization in n dimensions

We know broaden our scope to cover

$$\begin{aligned} & \min_{x \in \mathbb{R}^n} f(x) \\ \text{s.t. } & g(x) = 0 \\ & h(x) \leq 0 \end{aligned}$$

for $f : \mathbb{R}^n \to \mathbb{R}$.

Usually, we will focus on the unconstrained case.

Note there is an equivalence between the two if we use $\tilde{f}(x) = f(x)$ for x satisfying the constraints and ∞ o/w.

If at all possible, it is usually best to handle equality constraints through direct substitution.

Minimization in n dimensions

This is a very important problem because of two key applications:

- Calibration / GMM estimation (minimizing $SSRS(x)$ for $f : \mathbb{R}^n \to \mathbb{R}^m$—note there is probably not a root if $m > n$.)
- Value function maximization with more than one asset.

Minimization in n dimensions

When working with discrete choice models or discrete state spaces, a key problem is that things are not differentiable. Consequently, methods that require or assume differentiability may not apply or may not work.

There are a few very robust techniques to handling such cases.

My go-tos:

- Local search, C^0 or lower: Simplex / Amoeba / Nelder-Mead method
- Local search, C^1 or higher: BFGS or Levernberg-Marquardt depending on the problem
- Search in globally concave C^3 problems with analytic derivatives: Newton's method
- Global search: Controlled random search with local mutation

I will briefly mention Newton's method, too.

Method comparison

This accuracy comparison lines up well with my experience:

Comparison in terms of accuracy

Summary, median ranking

The summary table shows the median ranking across all the test problems. See *detailed results by test problem (accuracy)*.

Alternatively, see the *detailed results when using weighted least squares as cost function*.

	BFGS	Conjugate gradient (Fletcher-Reeves imp.)	Conjugate gradient (Polak-Ribiere imp.)	Damping	Levenberg-Marquardt	Levenberg-MarquardtMD	Simplex	SteepestDescent
NIST, "lower" difficulty	*3.841*	*3.003*	*3.003*	*1*	*1*	*1*	*1.017*	*6.436*
NIST, "average" difficulty	*269*	*22.35*	*22.5*	*1*	*1*	*1*	*79.31*	*315.9*
NIST, "higher" difficulty	*36.33*	*10.82*	*8.366*	*1.847*	*1*	*1*	*18.55*	*100.7*

(Note: 1 is best.)

Method comparison

The same thing for run times:

Comparison in terms of run time

Summary, median ranking

The summary table shows the median ranking across all the test problems. See *detailed results by test problem (run time)*.

Alternatively, see the *detailed results when using weighted least squares as cost function*.

	BFGS	Conjugate gradient (Fletcher-Reeves imp.)	Conjugate gradient (Polak-Ribiere imp.)	Damping	Levenberg-Marquardt	Levenberg-MarquardtMD	Simplex	SteepestDescent
NIST, "lower" difficulty	*1.258*	*1.412*	*1.391*	*1*	*1.094*	*1.036*	*1.622*	*11.83*
NIST, "average" difficulty	*1.326*	*9.579*	*7.935*	*1*	*1.11*	*1.035*	*1.901*	*12.97*
NIST, "higher" difficulty	*1.02*	*1.84*	*2.155*	*1.244*	*1.044*	*1.198*	*1.206*	*5.321*

`https://docs.mantidproject.org/v3.7.1/concepts/FittingMinimizers.html`

Simplex method

The simplex method goes by many names: simplex, amoeba, Nelder-Mead, and polytope method.

Why? Because people like it.

The general idea is that the method always goes "downhill," and so always finds a minimum of a continuous function.

Simplex method

First things first, what is a simplex?

For $x \in \mathbb{R}^n$, a simplex is defined by $n+1$ vertices $\{x^1, \ldots, x^{n+1}\}$:

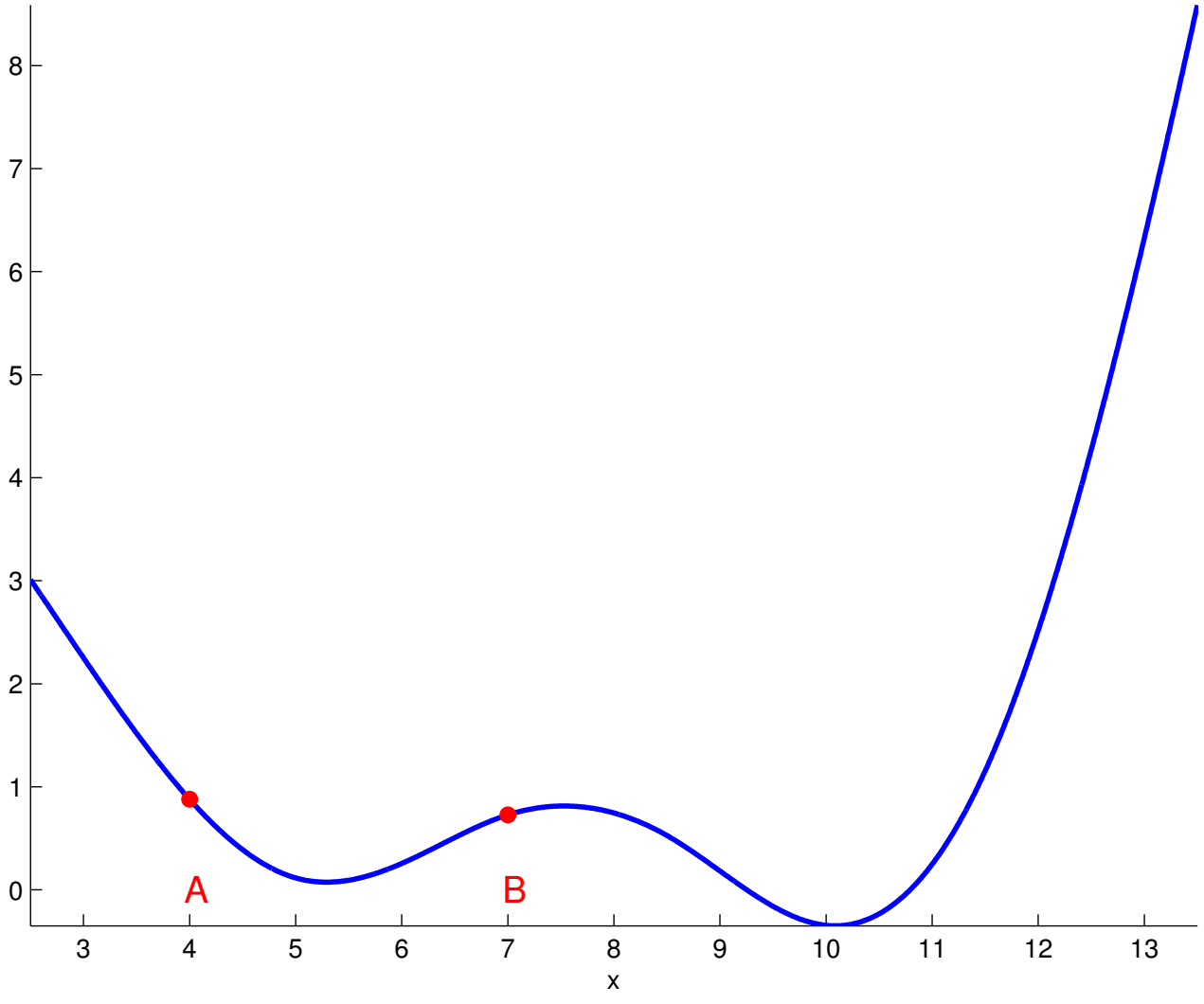
- $n = 1 \Rightarrow$ simplex is an interval
- $n = 2 \Rightarrow$ simplex is a triangle
- $n = 3 \Rightarrow$ simplex is a tetrahedron

One needs to be a little careful to ensure the method's initial simplex does not lie on a hyperplane.

Following is an illustration of the simplex method with $n = 1$.

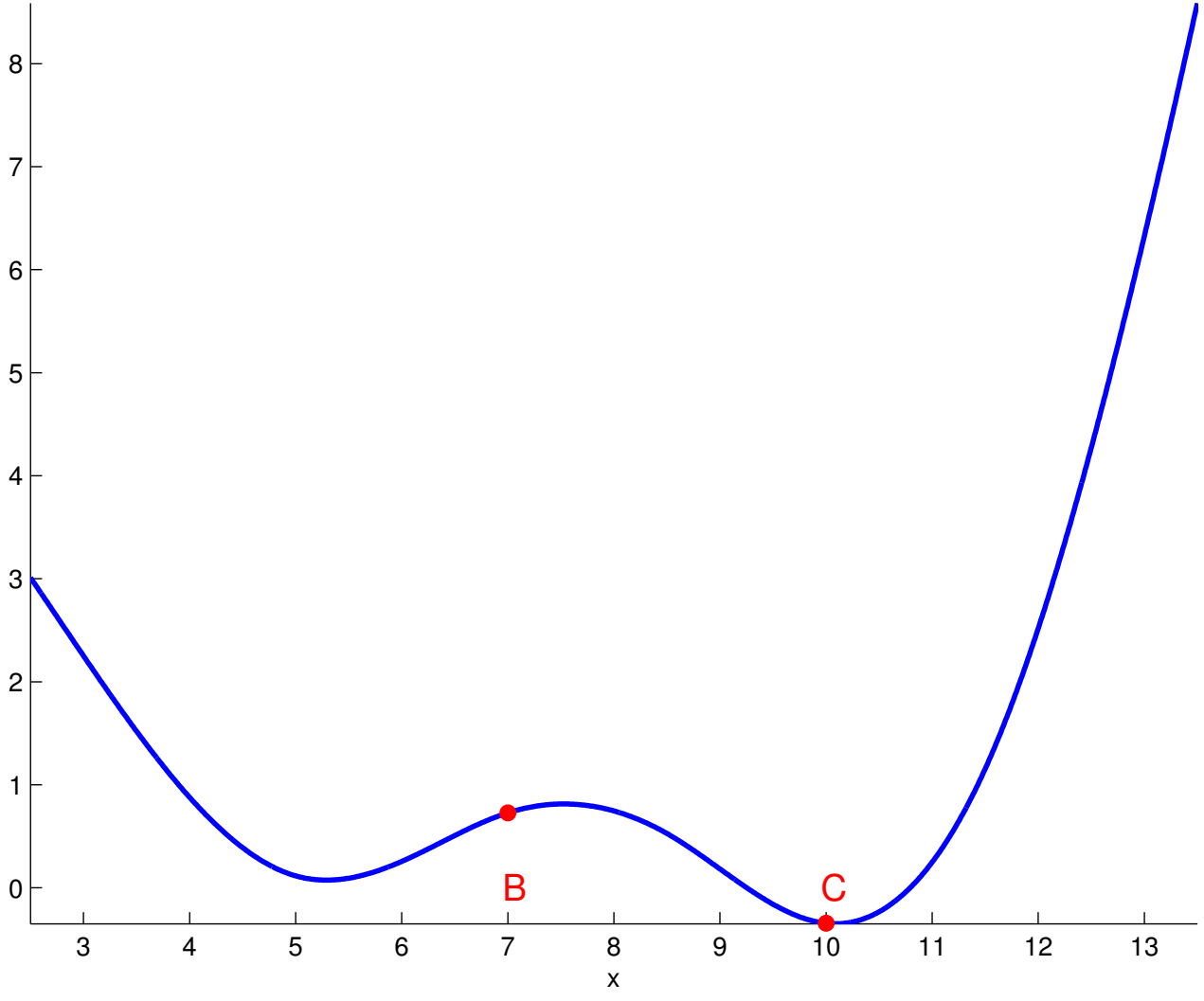
Simplex method

Initially, you have a simplex, $[A, B]$.



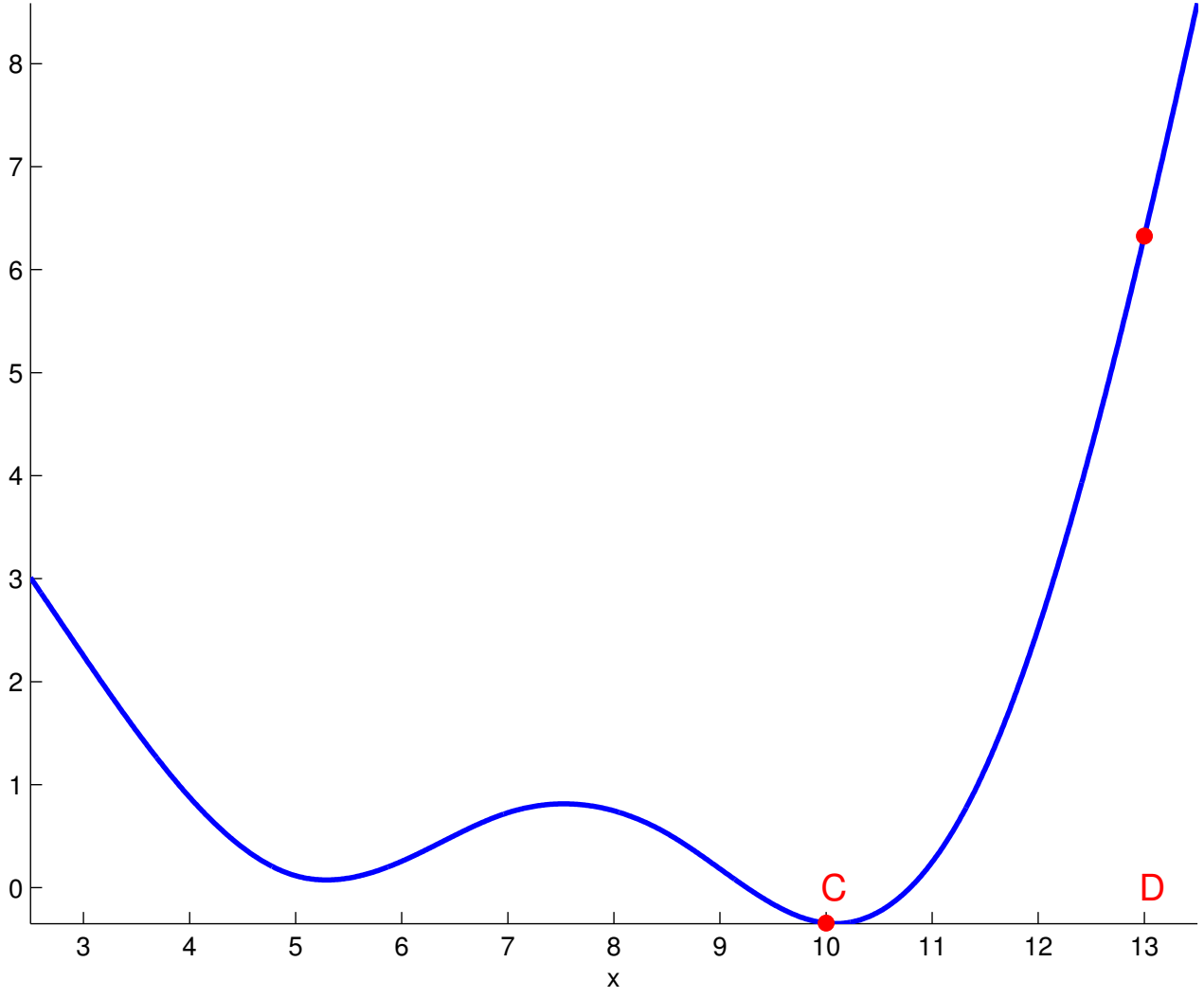
Simplex method

You check the rotation through the "center of gravity," here B:



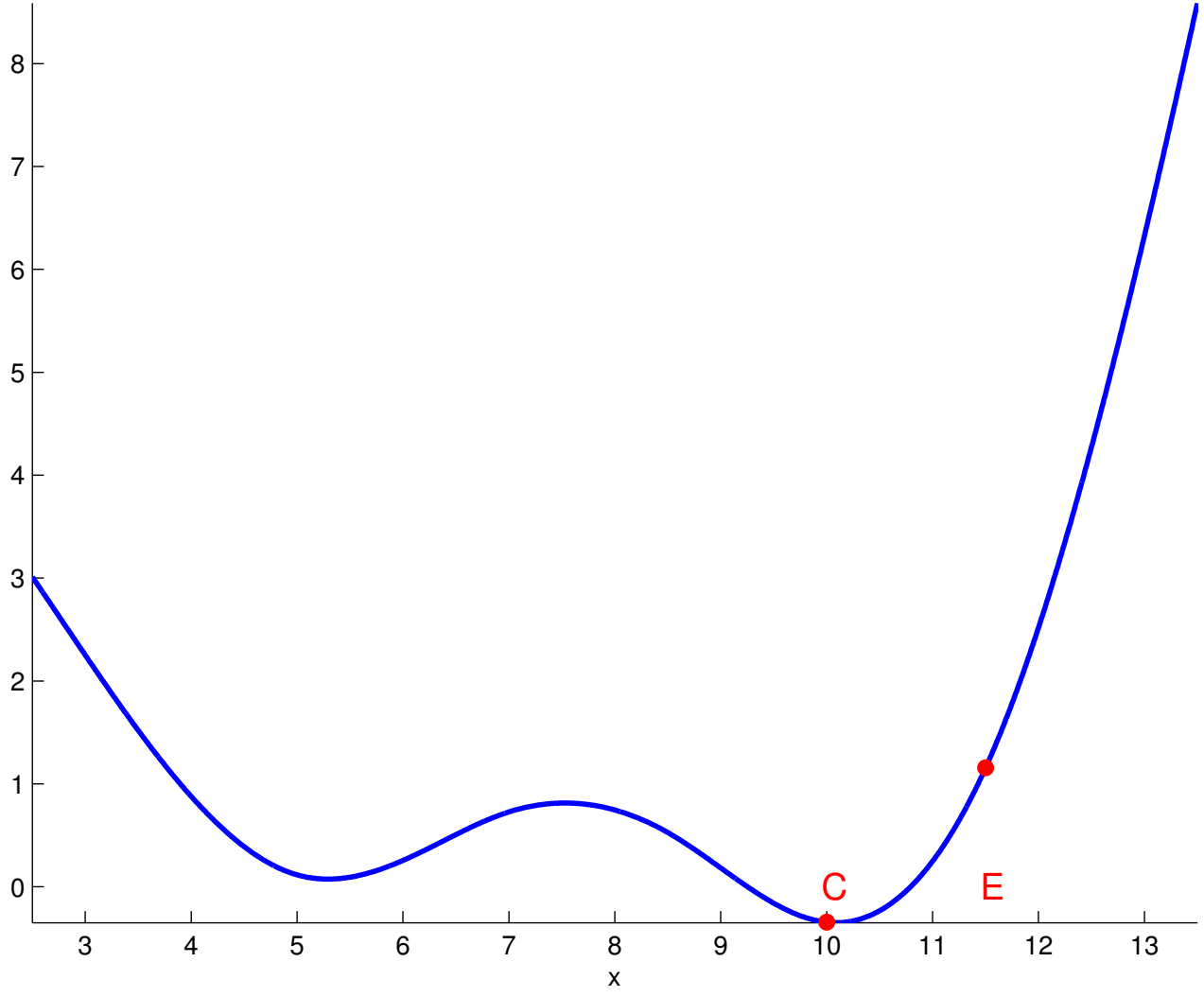
Simplex method

If C improves things, you keep going:



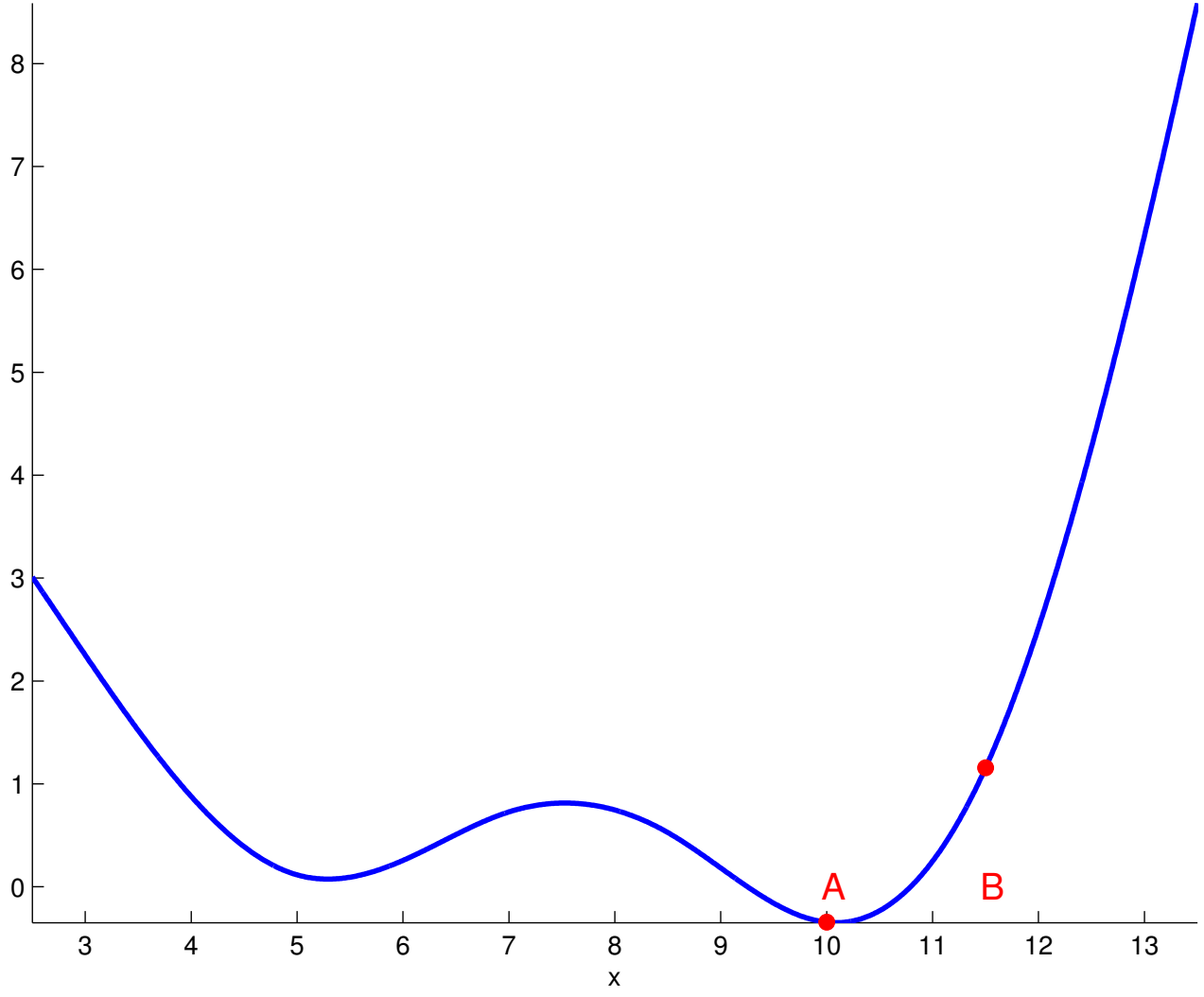
Simplex method

If D makes things worse, you move back towards C:



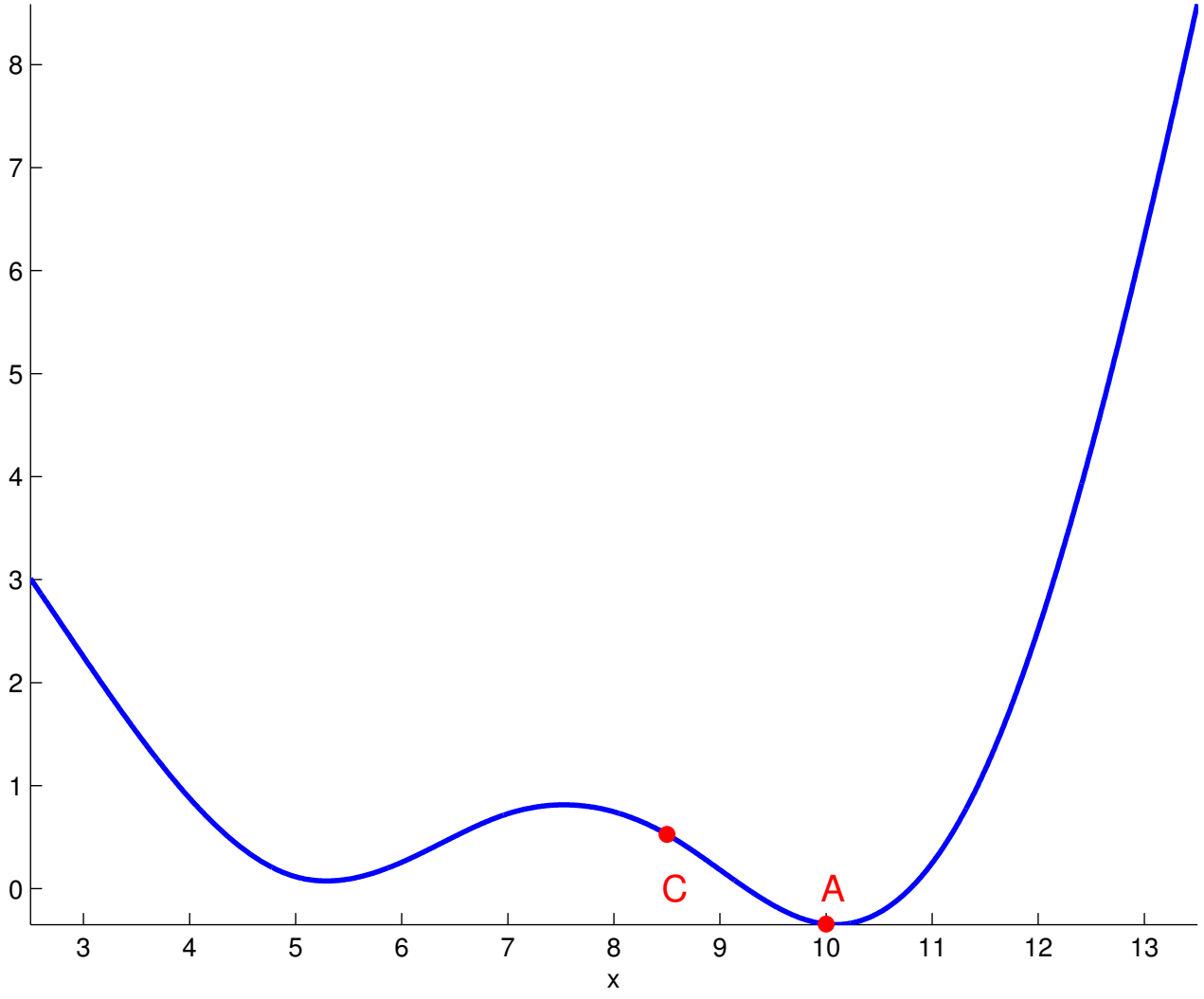
Simplex method

You now have a new simplex and proceed from the beginning:



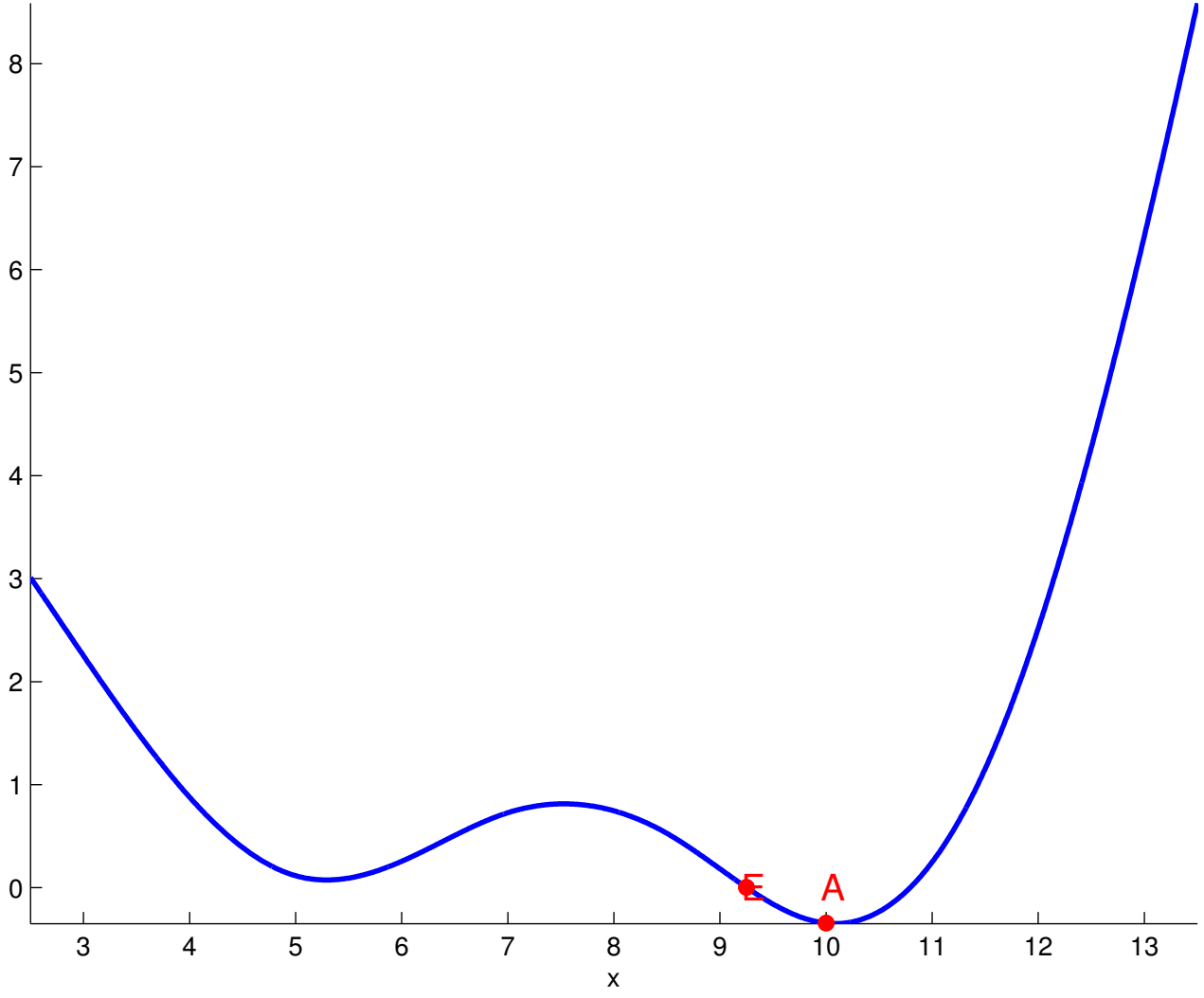
Simplex method

Rotate through the best point, check if better:



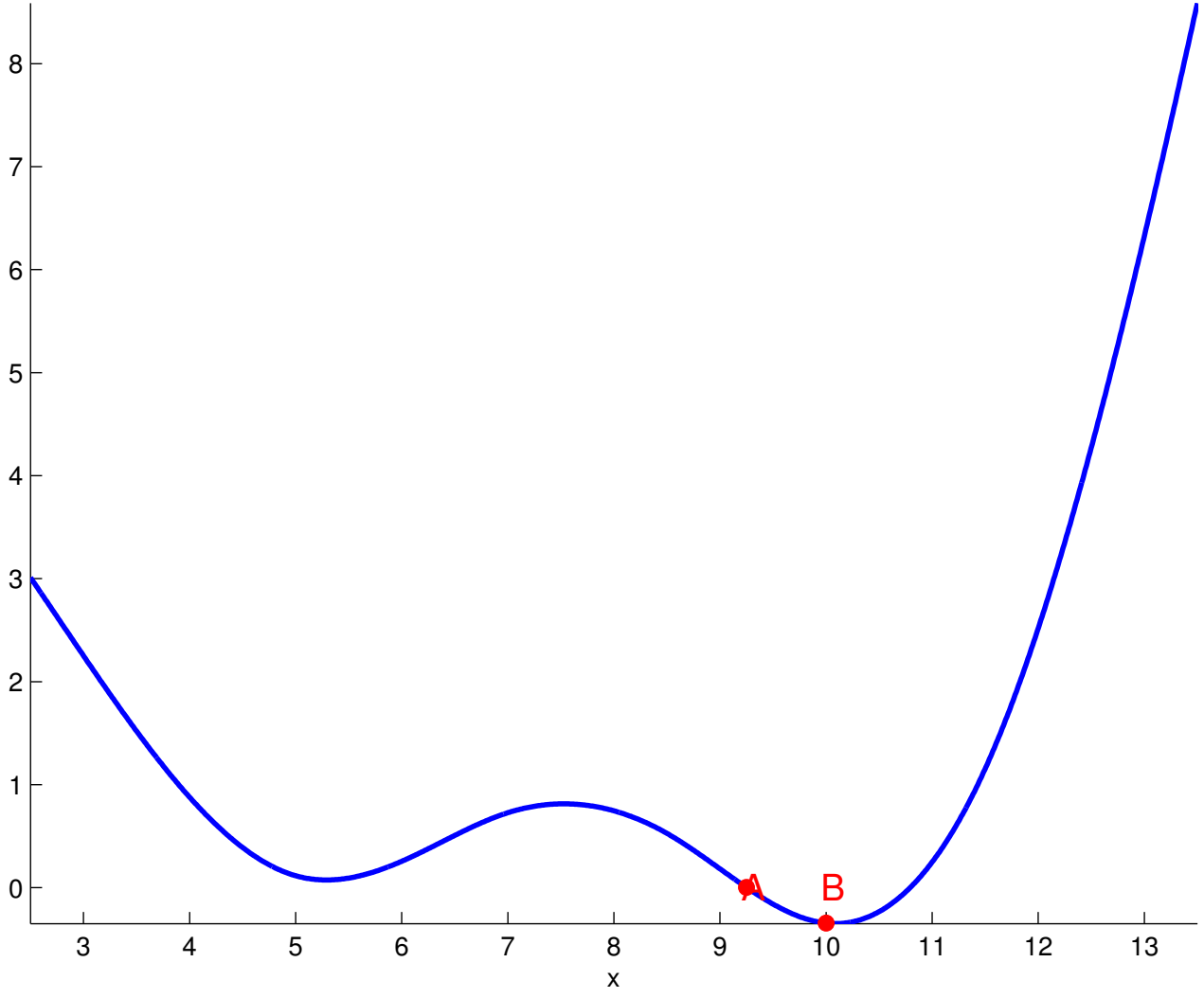
Simplex method

It wasn't, so shrink the simplex:



Simplex method

Relabel and start again:



Simplex method

Here are the details for the simplex method:

1. **Init** Construct a simplex.
   Fix $\alpha \in (0,1)$, $\beta > 1$, and $\gamma \in (0,1)$.
2. Rank the simplex so that $f(x_1) \leq \ldots \leq f(x_{n+1})$.
3. Check whether the simplex is "too small." If so, return x_1 and stop; o/w go to 4.
4. Calculate the "center of gravity" $x_g = \sum_{j=1}^{n} x_j/n$.
5. Compute $x_r = x_g + (x_g - x_{n+1})$.
   - 5.1 If $f(x_r) < f(x_1)$, try to expand. Set $x_e = x_g + \beta(x_g - x_{n+1})$.
     - 5.1.1 If $f(x_e) \leq f(x_r)$, replace x_{n+1} with x_e. Go to 2.
     - 5.1.2 If $f(x_e) > f(x_r)$, replace x_{n+1} with x_r. Go to 2.
   - 5.2 O/w, if $f(x_r) < f(x_n)$, then replace x_{n+1} with x_r and go to 2.
   - 5.3 O/w, if $f(x_r) \geq f(x_n)$, compute $x_c = x_g + \gamma(x_{n+1} - x_g)$.
     - 5.3.1 If $f(x_c) < f(x_{n+1})$, contract. Replace x_{n+1} with x_c and go to 2.
     - 5.3.2 O/w, contract all. Replace $x_i = x_1 + \alpha(x_i - x_1) \forall i \neq 1$ and go to 2.

Newton's method

Newton's method for optimization (as opposed to root-finding) requires a C^2 objective function. In this sense, it is the polar opposite of the direct search methods like simplex and compass.

How does it work?

- Construct a second-order Taylor expansion.
- Optimize (note: it doesn't distinguish between max and min) the Taylor expansion.
- Use the arg min as your new point.

Newton's method

The technical aspects are not too difficult.

The second-order Taylor expansion of $f : \mathbb{R}^n \to \mathbb{R}$ is

$$f(x+s) \approx f(x) + \nabla f(x)s + \frac{1}{2}s'H(x)s$$

where ∇ is the gradient (a row vector of first derivatives) and H is the Hessian.

The Hessian is a symmetric matrix of second derivatives:

$$H(x) = \begin{bmatrix} f_{11}(x) & f_{12}(x) & \dots & f_{1n}(x) \\ f_{12}(x) & f_{22}(x) & \dots & f_{2n}(x) \\ \vdots & \vdots & \ddots & \\ f_{1n}(x) & f_{2n}(x) & \dots & f_{nn}(x) \end{bmatrix}$$

Newton's method

Given some point x_n, we want to find $x_{n+1} = x_n + s_n$ that will hopefully be closer to the min.

Newton's method tries to solve

$$\min_s f(x_n) + \nabla f(x_n)s + \frac{1}{2}s'H(x_n)s$$

using the FOC

$$\nabla f(x_n) + H(x_n)s_n = 0$$

$$\Leftrightarrow s_n = -H(x_n)^{-1}\nabla f(x_n).$$

The new trial point is $x_{n+1} = x_n + s_n$.

Note: The one-dimensional case has $s_n = -f'(x_n)/f''(x_n)$.

Newton's method

We have a theoretical result that is very similar to the root-finding case:

Theorem

Suppose $f \in C^3$ *with an optimum at* x^* *and* $H(x^*)$ *nonsingular. Then* $\exists \epsilon > 0$ *s.t.* $||x_0 - x^*|| < \epsilon$ *implies* $x_n \to x^*$ *quadratically.*

The meaning is that Newton's method will converge for a good enough guess and do so quickly. However, it could converge to a maximum.

Newton's method

What stopping criteria should you use here? Three:

- Test whether $||s_n|| < \epsilon(1 + ||x_n||)$ for some $\epsilon > 0$. If it is, then the x_n have converged.
- If the x_n have converged, test whether $\nabla f(x) \approx 0$ (since it has to be 0 at an optimum). Specifically, test whether $||\nabla f(x)|| \leq \delta(1 + |f(x)|)$ for some $\delta > 0$.
- Set a maximum number of iterations or function evaluations since it may not converge.

A final step if the gradient is close to zero is to determine whether you are at a maximum or a minimum. If at a minimum, H should be positive semi-definite.

Newton's method

Advantages

1. Quadratic convergence near the solution.
2. Well known.

Disadvantages

1. Not guaranteed to converge to anything.
2. Cost of computing derivatives numerically substantially increasing in n.

Note: remember that in some applications you can use symbolic math in Matlab, Mathematica, Maple, etc. to compute H for you. There's also a technique called automatic differentiation.

Direction Set Methods

Newton's method constructs a search direction $s_n = -H(x_n)^{-1}\nabla f(x_n)$ and immediately goes to

$$x_{n+1} = x_n + s_n.$$

Direction set methods (also called line-search methods) are similar in that they specify a search direction, but they relax the step using

$$x_{n+1} = x_n + \lambda_n s_n$$

where

$$\lambda_n = \arg\min_{\lambda \geq 0} f(x_n + \lambda s_n).$$

Direction Set Methods

How does one solve for λ_n? Can use "backtracking line search" or golden section etc.

It's important to have it sufficiently accurate, because some methods are using it to construct Hessian approximations etc.

The Wolfe conditions are good criteria for stopping the search:

1. $f(x_n + \lambda_n s_n) \leq f(x_n) + c_1 \lambda_n \nabla f(x_n) s_n$
2. $-\nabla f(x_n + \lambda_n s_n) s_n \leq -c_2 \nabla f(x_n) s_n$

for $0 < c_1 < c_2 < 1$. Typically, $c_1 = 10^{-4}$ and $c_2 = .9$.

If the gradient $\nabla f(x_n) \neq 0$ and was accurate, then going δ in that direction improves: Can satisfy 1 with a small enough λ.

Satisfying 2 is important because it will ensure, e.g., that the Hessian estimate in BFGS remains positive definite.

Aim for 1, ex post check 2, and if 2 is violated, stop.

Direction Set Methods

Examples of direction set methods

1. Newton with line search (can be good)
2. Coordinate direction set (do not use this)
3. Gradient descent (generally, don't use this)
4. BFGS (good)
5. Levenberg-Marquardt (good, but SSRS problems only)

Note: Chris Sims' "csminwel" uses BFGS.

Gradient Descent

Gradient descent is an important method:

- For $f \in C^1$, if $\nabla f \neq 0$, one can always decrease f by moving an infinitessimal amount in the direction $-\nabla f'$.
- Because of this, it provides a way to find a search direction that will decrease the objective.
- It does not require a Hessian, which requires C^2.
- Other methods partly use this, such as Levenberg-Marquardt.

On the computer, of course this doesn't hold exactly.

BFGS

Newton's method with line search can be desirable in several respects, namely the fast convergence locally.

Unfortunately, the Hessian is costly to compute. How can one economize on that?

BFGS (Broyden-Fletcher-Goldfarb-Shanno) uses an approximate Hessian constructed using only function values and gradients.

BFGS

The search direction s_n is

$$s_n = -H_n^{-1} \nabla f(x_n).$$

Then $x_{n+1} = x_n + \lambda_n s_n$.

H_n is updated recursively using

$$H_{n+1} = \begin{cases} H_n - \frac{H_n z_n z_n' H_n}{z_n' H_n z_n} + \frac{y_n' y_n}{y_n z_n} & \text{if } y_n z_n > \delta \\ H_n & \text{otherwise} \end{cases}$$

where $H_0 = I$, $\delta > 0$ is some threshold, and

$$z_n = x_{n+1} - x_n \text{ and } y_n = \nabla f(x_{n+1}) - \nabla f(x_n).$$

The elements in red are scalars.

BFGS

Advantages

1. For C^2 functions, converges to a minimum.
2. Much faster than gradient descent.
3. Slower than Newton's method, but no need to compute Hessian.

Disdvantages

1. Requires a gradient.
2. The theory needs a Hessian.

Levenberg-Marquardt

Suppose you want to minimize the sum of squared residuals,

$$S(x) = \frac{1}{2} f(x)' f(x) \quad \left(= \frac{1}{2} \sum_{i=1}^{n} f_i(x)^2 \right)$$

where $f : \mathbb{R}^m \to \mathbb{R}^n$ and $S : \mathbb{R}^m \to \mathbb{R}$.

The Levenberg-Marquardt algorithm provides one way to do this.

Levenberg-Marquardt

First off, take the Taylor approximation of f about x:

$$f(x+s) \approx f(x) + J(x)s$$

where J is the Jacobian. Substituting this in to the expression for S, we have

$$S(x+s) \approx \frac{1}{2}(f(x) + J(x)s)'(f(x) + J(x)s)$$

which rearranges to

$$S(x+s) \approx \frac{1}{2}\left(f(x)'f(x) + s'J(x)'J(x)s + f(x)'J(x)s + s'J(x)'f(x)\right)$$

Levenberg-Marquardt

We have found

$$S(x+s) \approx \frac{1}{2}\left(f(x)'f(x) + s'J(x)'J(x)s + f(x)'J(x)s + s'J(x)'f(x)\right)$$

Now at the minimum of S, we have the gradient of $\nabla S = 0$. Taking the gradient with respect to s and setting it to 0, (i.e., the FOC wrt s) we have

$$\nabla S(x+s) = J(x)'J(x)s + \frac{1}{2}\left((f(x)'J(x))' + J(x)'f(x)\right) = 0$$

or

$$J(x)'J(x)s = -J(x)'f(x)$$

What does this mean? By choosing s to solve this system of linear equations ($J'J$ and matrix, $-J'f$ a vector), we are optimizing the linear approximation of the SSRS S.

Levenberg-Marquardt

At this point, this is basically a pure Newton step. Levenberg adjusted this to a damped version:

$$(J(x)'J(x) + \lambda I)s = -J(x)'f(x)$$

where $\lambda > 0$ and I is the identity matrix.

- If $\lambda = 0$, s is a Newton step.
- For large values of λ, s becomes proportional $-J'f$ — the gradient of $S(x)$ is $J'(x)f(x)$ so this becomes gradient descent.
- This feature guarantees that the method goes downhill for large enough λ.
- Stated differently, there exists a $\lambda \geq 1$ such that $S(x + s) \leq S(x)$.

Levenberg-Marquardt

Marquardt's contribution is to scale the step size to avoid slow convergence in the case of large λ.

$$(J(x)'J(x) + \lambda diag(J(x)'J(x)))s = -J(x)'f(x)$$

Levenberg-Marquardt

Advantages

1. Always goes downhill for sufficiently large λ.
2. Frequently used. One case is non-linear least squares fitting.
3. Well-conditioned, probably better attempting to minimize S directly w/o exploiting the special SSRS structure.

Disadvantages

1. A Newton-type method, will easily get trapped in local minima.
2. Needs derivatives.

Global Optimization

All the methods discussed so far can get stuck in a local minima. How can we try to find a global minimum?

The primary way is to generate several starting points x_0 and compute the optimum conditional on each x_0. The best is hopefully the global.

Controlled random search with local mutation

My favorite global optimization algorithm formalizes this procedure.

CRS2LM works as follows:

1. Init: generates an initial pool of N points $\{x_i\}_{i=1}^{N}$ for $x_i \in X \subset \mathbb{R}^D$ and evaluates f at each.
2. Pick $D+1$ points randomly to form a random simplex.
3. Try one Nelder-Mead step (reflecting the worst point through the center of gravity).
4. If it's better, replace the worst point in the pool; otherwise, if it is worse,
   - **Global variant:** try again by going to step 2.
   - **Local variant:** take a convex combination of the worst point and the best point, where the convexity weight is $U[0,1]$, and replace the worst point.
5. Check termination criteria, STOP if appropriate. O/w, go to step 2.

The local variant is much more tolerable in convergence times.

Controlled random search with local mutation

The beauty of this algorithm is it is very parallelizable.

You can have a master node distribute points and update the pool as information comes in.

All that needs to be sent is the parameter vector.

It also handles very nasty objective functions, and I've found it to be super robust.

Handling constrained problems

To handle the constrained problem

$$\min_x f(x)$$
$$\text{s.t. } g_i(x) = 0$$
$$h_j(x) \leq 0$$

the usual approach is to first get rid of all g_i, i.e. equality constraints, by direct substitution.

E.g., if we are solving

$$\max_{c,a'} u(c) \text{ s.t. } c + a' = we + a \text{ and } c \geq 0,$$

solve instead

$$\max_{a'} u(c(a')) \text{ s.t. } c(a') \geq 0$$

where $c(a') := we + a - a'$.

Handling constrained problems

Supposing one can eliminate all the equality constraints in this way. One is left with

$$\min_x f(x)$$
$$\text{s.t. } h_j(x) \leq 0.$$

The difficulty of this problem varies:

- If $f : \mathbb{R} \to \mathbb{R}$ and the h_j just capture $x \in [a, b]$, then we know how to handle this. E.g., use Brent's method.
- If $f : \mathbb{R}^n \to \mathbb{R}$ and the h_j just capture $x_i \in [a_i, b_i] \forall i$, then there are special methods for this. See Coleman and Li (1996) which is the method optionally used by Matlab's lsqnonlin.
- O/w, this is much harder.

Handling constrained problems

One approach is to replace the minimization with an equivalent problem

$$\min_x \tilde{f}(x)$$

where

$$\tilde{f}(x) = \begin{cases} f(x) & \text{if } h_j(x) \leq 0 \forall j \\ \infty & \text{if } h_j(x) > 0 \text{ for some } j \end{cases}$$

If you were going to use a direct search / comparison method, this approach has virtually no drawbacks.

Handling constrained problems

Another approach is to replace the minimization with a related, but not equivalent, problem

$$\min_x \tilde{f}(x)$$

where

$$\tilde{f}(x) = f(x) + \sum_j P_j \max\{0, h_j(x)\}^2.$$

Then for differentiable f and h in C^1, $\tilde{f}$ is also C^1. The first derivative is

$$\frac{\partial \tilde{f}(x)}{\partial x_i} = \frac{\partial f(x)}{\partial x_i} + 2 \sum_j P_j \mathbf{1}[h_j > 0] \frac{\partial h_j(x)}{\partial x_i} h_j(x)$$

which is continuous.

Handling constrained problems

As the $P^j \uparrow \infty$, presumably the argmin and min agree with the original problem values.

The primary way to assess how close the penalty method solution is to the true solution is test by "how much" the $h_j \leq 0$ constraints are violated.

Computational note:

- You would probably solve P_j for values close to 0 initially.
- Then, increase P_j by some constant factor, say 10.
- At each step, you should use your old solution as the initial guess.
- You should make P_j as large as possible while still obtaining convergence.

Handling constrained problems

One last way to handle constrained problems applies primarily when f has a limited domain.

A limited domain problem is when f is not defined on all of $\mathbb{R}^n$.

Note that in this case, you cannot use the penalty method as stated.

But you can sometimes transform the problem using a change of variables:

$$\min_{x \in X} f(x) \Leftrightarrow \min_{z \in g^{-1}(X)} f(g(z))$$

for some function g. If $g^{-1}(X) = \mathbb{R}$, then this is an unconstrained problem.

Handling constrained problems

Example:

- $X = \mathbb{R}^+$, $g^{-1}(x) = \log(x)$, and $g(z) = \exp(z)$.

You can obtain derivatives of the transformed problem using the chain rule:

$$\frac{df(g(z))}{dz} = f'(g(z))g'(z)$$

This can be a useful trick, but note that if the optimum is a corner solution, your optimal choice may be $\pm\infty$ (implying your minimization routine will not converge).

Summary

We considered

$$\min_x f(x)$$
$$\text{s.t. } g_i(x) = 0$$
$$h_j(x) \leq 0$$

for $f : \mathbb{R}^n \to \mathbb{R}$.

- Best way to deal with equality constraints is direct substitution.
- Inequality constraints can be handled by converting to an unconstrained problem:
  - using a comparison method with $\tilde{f} = f + \mathbf{1}[h_j > 0]\infty$.
  - using a penalty method, $\tilde{f} = f + P \max\{0, h_j\}^2$.
  - use a change of variables, $\tilde{f} = f \circ g$.

Summary

For the unconstrained problem,

$$\min_x f(x)$$

for $f : \mathbb{R}^n \to \mathbb{R}$, we found

- $n = 1$ is easy to handle if f is C^0 because of bracketing.
  You should probably use Brent's method.
- $n > 1$ with f in C^2, or C^3 is more difficult because of
  - local minima
  - number of derivatives involved

  Best choice here is Levenberg-Marquardt if applicable, BFGS o/w.
- $n > 1$ with $f \notin C^2$ has fewer options.
  Probably best to use a comparison method, like Nelder-Mead.
- Usually hard to find global minima, try many different initial points if worried about it.

OOP, Improving Speed, and Parallel Programming

Object-oriented vs procedural programming

Until the 1980s, all programming was procedural

- Statements are executed by procedures in some order
- The data types are pre-defined, like `real(8)`

In contrast, object-oriented programs (OOP)

- use classes, a sort of template that contains
  - instance variables (data)
  - methods (procedures operating on class instances)
  - objects (the instance of a class)
- many other features such as encapsulation and inheritance.

Key advantage to me of OOP is reducing coding times b/c of

1. debugging: Code is often more intuitive / simpler
2. flexibility: E.g., can add a parameter/state without recoding interfaces or using globals

Derived-Data Types

Derived-data types

- Are called derived because they are derived from other types (possibly other derived types).
- Each component is either a built-in type or a derived type.
- Enable grouping of items into one conceptual unit.
- Enable one to "hide" a complicated underlying structure.

Derived-Data Types

Example of a derived data type:

```
! defined type
type choices
  real(8) :: b,k
  integer :: i
  logical :: isfeas
end type
type(choices) :: policy, policies(10)
```

`b,k,i,isfeas` are called components of the derived type.

Derived-Data Types

```
type(choices), dimension(1000) :: bkGrid
type(choices) :: bkBest

!Setup uniformly distributed random grid
call random_number(bkGrid%b)
call random_number(bkGrid%k)
bkGrid%b = bkGrid%b*(maxB-minB) + minB
bkGrid%k = bkGrid%k*(maxK-minK) + minK
do i = 1,1000
  bkGrid(i)%i = i
end do
```

Note:

- `bkGrid%x` (where `x` is any component) is a vector.
- `bkGrid(i)%x` (where `x` is any component) is a scalar.
- `bkBest%x` (where `x` is any component) is a scalar.

Derived-Data Types

```
!Find best (b,k) combination subject to b+k>=0d0
bkGrid%isfeas = bkGrid%b + bkGrid%k >= 0d0
do i = 1,1000
  ! Copy bkGrid(i) to bkChoice
  bkChoice = bkGrid(i)
  if (bkChoice%isfeas) then
    call sub_value(bkChoice,...)
  else
    ...
  end if
  ...
end do
```

Derived-Data Types

Alternatively, not necessarily scalar items can be grouped:

```
type smolyakFcn
  integer :: d, mu
  integer, allocatable, dimension(:) :: i
  real(8), allocatable, dimension(:,:) :: T
end type
```

Derived-Data Types

```
subroutine smolyakApprox1(s,x,d,mu,lb,ub)
  implicit none
  ! Derived type smolyakFcn is declared with intent(out)
  ! Because of intent(out), s%i and s%T auto deallocated
  type(smolyakFcn), intent(out) :: s
  ! x auto deallocated
  real(8), dimension(:,:), allocatable, intent(out) :: x
  integer, intent(in) :: d, mu
  real(8), dimension(d), intent(in) :: lb, ub
  !local
  integer :: i

  ! Copy some input variables to s components
  s%d = d
  s%mu = mu

  ! s%i is an array
  allocate(s%i(100))
  do i = 1,100
    s%i(i) = i  !Note how indexing is for the component
  end do
  ...
end subroutine smolyakApprox1
```

Type-Bound Procedures

Modules combine data and procedures in one unit.

A related way to do this is with type-bound procedures.

```
type timer
    real(8) :: refpt ! reference point
    logical :: ispaused = .false.
    real(8) :: total = 0d0
contains
    procedure, pass :: stop => timer_stop
    procedure, pass :: start => timer_start
    procedure, pass :: elapsed => timer_elapsed
end type
```

The `a => b` renames `b` as `a`.

Note the `pass` attribute (we will revisit this shortly).

Type-Bound Procedures

How can you call this?

```
type(timer) :: ta, tb
call ta%start()
do i = 1,100
  call tb%start()
  call foo()
  call tb%stop()
  do j = 1,100
    ...
  end do
end do
print*,'total time: ',ta%elapsed()
print*,'time on foo: ',tb%elapsed()
```

This works because `ta` and `tb` have their own data, `refpt`, `ispaused`, `total`.

Type-Bound Procedures

The pass attribute of procedure means that the variable is passed as an argument to the routine.

E.g., the routine timer_stop is declared as

```
subroutine timer_stop(this)
  use omp_lib
  implicit none
  class(timer) :: this ! note "class" declaration
  this%total = this%total + &
               (omp_get_wtime()-this%refpt)
  this%ispaused = .true.
end subroutine
```

However, it is called as

```
call ta%stop()
```

There is "no argument" to stop because ta is passed (pass attribute).

Type-Bound Procedures

For your reference, here is the rest:

```
subroutine timer_start(this)
  use omp_lib
  implicit none
  class(timer) :: this
  this%refpt = omp_get_wtime()
  this%ispaused = .false.
end subroutine

real(8) function timer_elapsed(this)
  use omp_lib
  implicit none
  class(timer) :: this
  if (this%ispaused) then
    timer_elapsed = this%total
  else
    timer_elapsed = this%total &
                  + (omp_get_wtime()-this%refpt)
  end if
end function
```

Best do-loop order

Array storage and best do-loop order.

- Fortran arrays are stored in column-major order.
- What this means for a 2x2 array `A`:

$$\begin{bmatrix} \texttt{A(1,1)} \\ \texttt{A(2,1)} \\ \texttt{A(1,2)} \\ \texttt{A(2,2)} \end{bmatrix}$$

  Note is essentially stored as a vector exhausting columns first.
- Another way to put it: the inmost dimension varies fastest.
- Matlab also stores in column-major order but C does it in row-major order.

Best do-loop order

Why do we care? Performance

- If we access the elements in continuous segments, it can be much faster than having to jump around.

For instance, this code is fast,

```
do k = 1,n3
  do j = 1,n2
    do i = 1,n1
      A(i,j,k) = B(i,j,k) + 1d0
    end do
  end do
end do
```

Best do-loop order

In contrast, this code is not as fast (unless the compiler reoptimizes it for you):

```
do i = 1,n1
  do j = 1,n2
    do k = 1,n3
      A(i,j,k) = B(i,j,k) + 1d0
    end do
  end do
end do
```

It's subtle, but it can matter a lot (one time this was the slowest part of computing an invariant distribution). But it won't if the loop is complicated.

Best do-loop order

Another case where column-major order is important:

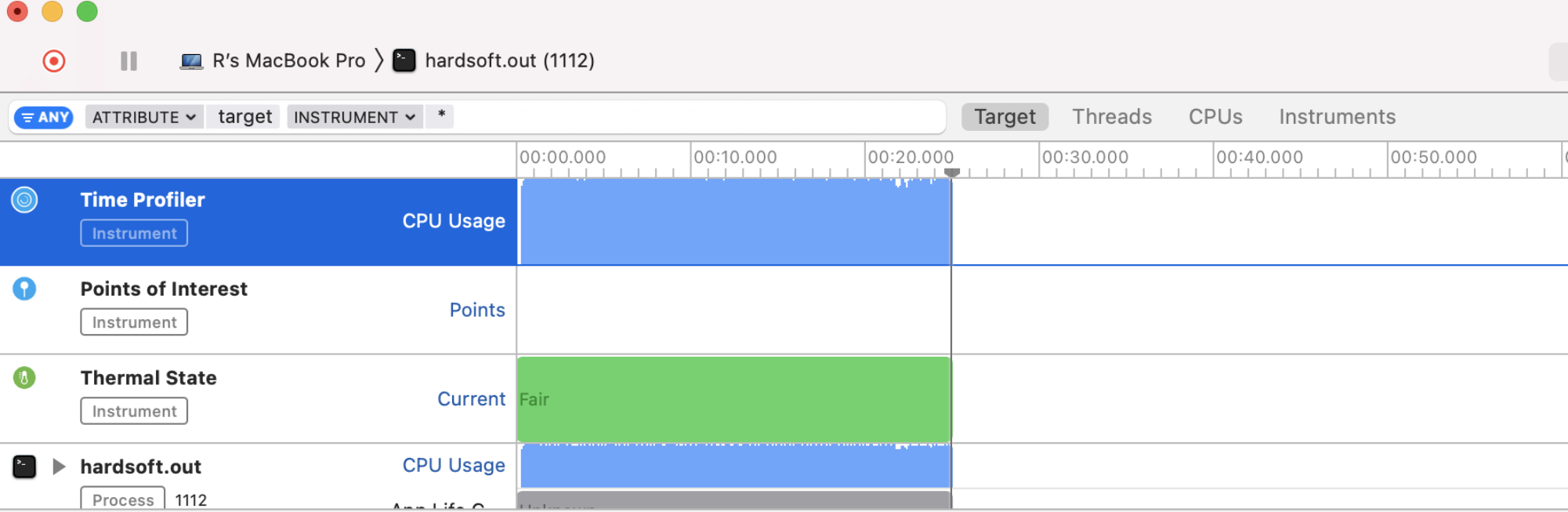
```
real(8) :: Mfast(50,20), Mslow(20,50)

! fast: passing a contiguous array section
do i = 1,20
  call sub_foo(Mfast(:,i))
end do

! slow: passing a non-contiguous array section
do i = 1,20
  call sub_foo(Mslow(i,:))
end do
```

In the second case, the compiler must make a temporary array, copy `Mslow(i,:)` to it, and then pass the temp to `sub_foo`. It must do this 20 times.

Profiling

Profiling your code is a great way to figure out what is slowing the code.

On the next slides I've put examples from Mac's "Instruments" (and the Time Profiler option in it) which is free but you have to install XCode.



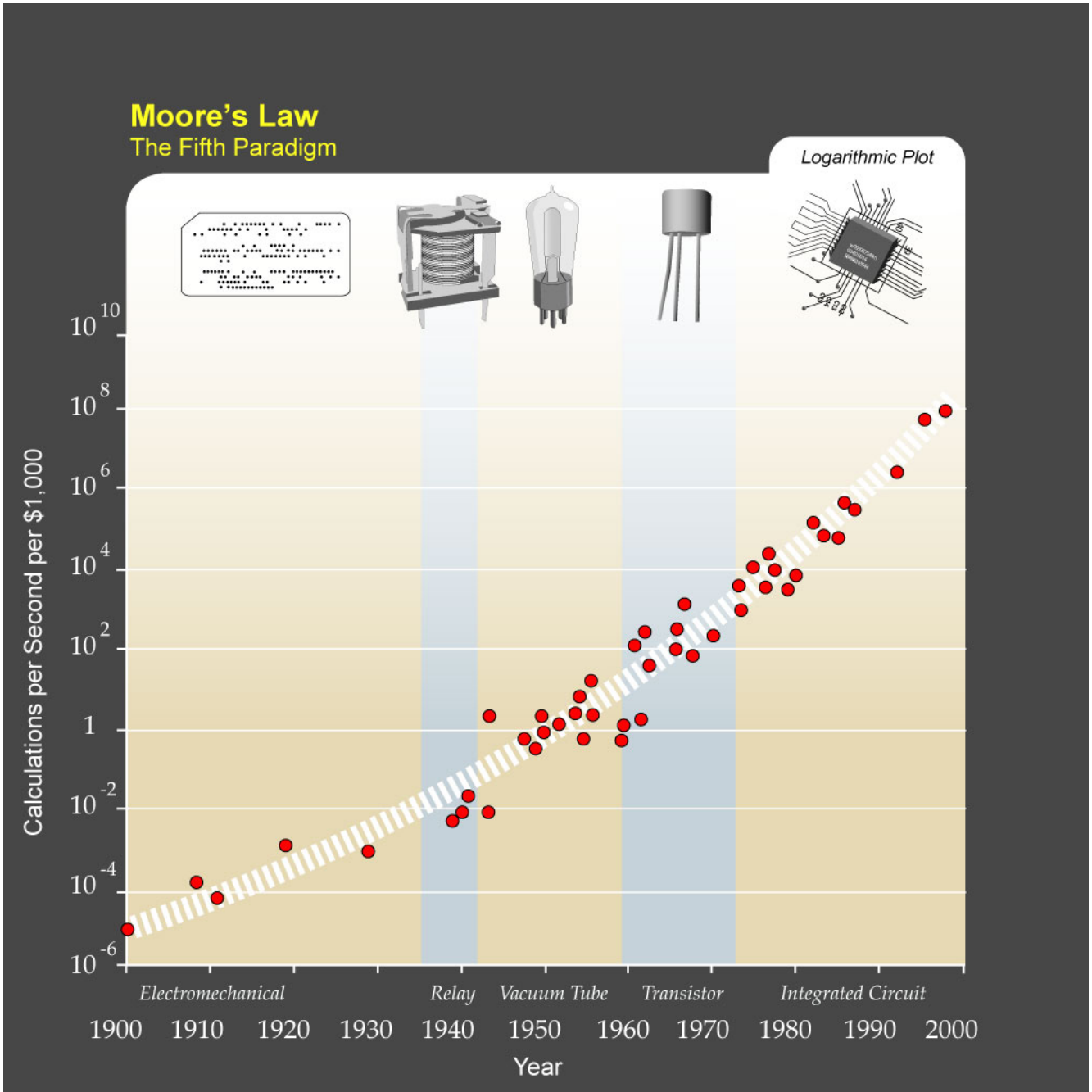
Time Profiler 〉 Profile 〉 Root

Weight	Self	Symbol Name
3.10 min 100.0%	0 s	hardsoft.out (1112)
22.20 s 11.9%	0 s	cthread_yield 0x176cdd1
22.11 s 11.8%	0 s	__kmp_counting_lock libiomp5.dylib
22.11 s 11.8%	0 s	__kmp_invoke_task_func libiomp5.dylib
22.11 s 11.8%	2.00 ms	__kmp_invoke_microtask libiomp5.dylib
9.14 s 4.9%	2.41 s	mod_solvesub_vfi_mp_updater_ hardsoft.out
5.81 s 3.1%	1.39 s	mod_param_mp_condexpectbpx_ hardsoft.out
4.54 s 2.4%	2.22 s	mod_solvesub_vfi_mp_updateq_ hardsoft.out
2.48 s 1.3%	724.00 ms	mod_solvesub_vfi_mp_updateh_ hardsoft.out
101.00 ms 0.0%	0 s	<Unknown Address>
36.00 ms 0.0%	36.00 ms	0x7fff2a4e0290 libsystem_m.dylib
73.00 ms 0.0%	0 s	bool _INTERNAL_25_____src_kmp_barrier_cpp_a115b87c::__kmp_wait_template<kmp_flag_64<false, true>, true,
8.00 ms 0.0%	0 s	mod_core_mp_bsearch_ hardsoft.out
4.00 ms 0.0%	0 s	thread_start libsystem_pthread.dylib
1.00 ms 0.0%	0 s	0x36f
1.00 ms 0.0%	0 s	<Unknown Address>
1.00 ms 0.0%	0 s	__kmp_finish_implicit_task libiomp5.dylib
20.92 s 11.2%	0 s	cthread_yield 0x176ccf6
20.66 s 11.0%	0 s	cthread_yield 0x176cdd0
19.10 s 10.2%	0 s	cthread_yield 0x176cdcf

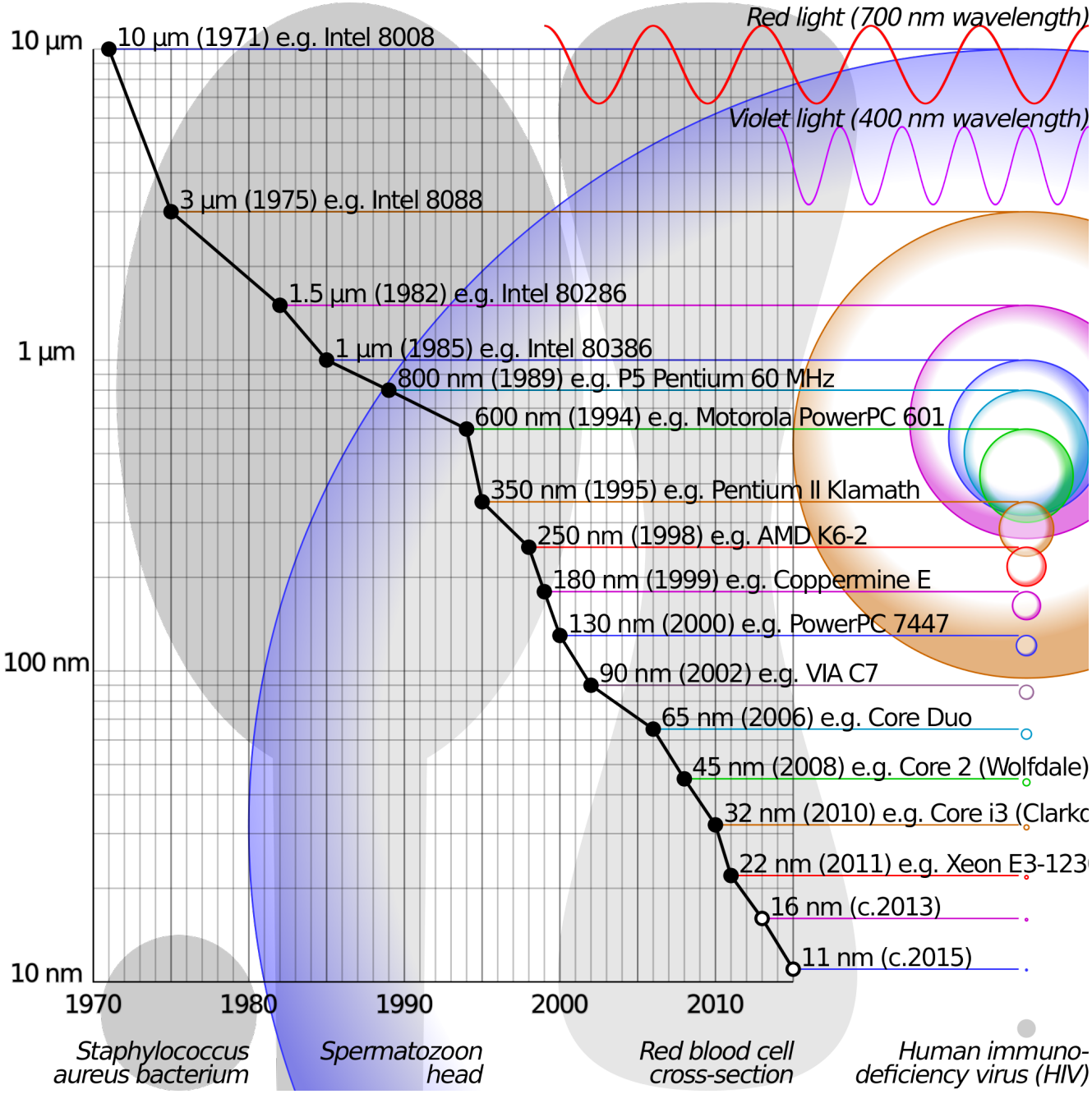
```
239             do istate = 1,p%nBYst; iBYst = state(istate)                                                0.01%
240
241                 ! Get a = b/g using the map from the pseudo debt-output \mathfrak{b}_t and \phi
242                 a = m%gBYst(iBYst) / phi                                                                0.01%
243
244                 ! Before maximizing, set the default very bad so it is replaced
245                 m%R(iBYst,iX)%V = p%death                                                               0.02%
246                 bestRERdev = -1d100
247
248                 maxBorr = 0d0
249
250                 ! Get the bp range --- with binary mono, this too involves a mapping
251                 lb = 1;       if (stateForLb(istate)>0) lb = m%R(stateForLb(istate),iX)%byChIndices(1)  0.05%
252                 ub = p%nBYch; if (stateForUb(istate)>0) ub = m%R(stateForUb(istate),iX)%byChIndices(1)  0.01%
253                 do iBYch = lb,ub                                                                        0.54%
254
255                     ! Map this choice value for debt-output into a choice variable for b_{t+1}
256                     bp = m%gBYch(iBYch) / phi ! NOTE: should be phi, not phi' because that depends on future shocks  1.65%
257
258                     ! Evaluate the old model's bond price schedule
259                     q = m0%evalQifR(iBYch,iX,p)                                                         12.76%
260
261                     ! If violated minQ / maxSpread, go to next bond (where will also likely be violated)
262                     if (m%gBYch(iBYch)/=0d0 .and. q<m%minQ) then                                        5.76%
263                         cycle
264                     end if
265
266                     debtServ = p%lambda2 * a                                                            0.31%
267                     tax = - q * (bp - (1d0-p%lambda)*a) + debtServ                                      0.69%
268                     maxBorr = max(maxBorr,debtServ - tax)                                               0.45%
269
270                     cT = m%gX(iX)%z - tax                                                               1.57%
271
272                     if (cT>0d0) then                                                                    0.39%
273                         psi  = p%psi(cT * m%gX(iX)%e)                                                   17.85%
274                         c = cT * psi                                                                    0.26%
275                     else
276                         psi = p%psi(1d-10)                                                              47.39%
277                         c = -1d0
278                     end if
279
280                     v = p%U(c,psi,m%gX(iX),def=.false.) + W(iBYch,iX)                                   5.05%
281
282                     ! Normally, this choice is better if it maximizes utility
283                     better = m%R(iBYst,iX)%V < v                                                        1.14%
```

Parallel programming

Moore's Law


Moore's Law
The Fifth Paradigm
Logarithmic Plot
Calculations per Second per $1,000
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
1
10^{-2}
10^{-4}
10^{-6}
Electromechanical
Relay
Vacuum Tube
Transistor
Integrated Circuit
1900
1910
1920
1930
1940
1950
1960
1970
1980
1990
2000
Year

Introduction to Parallel Programming



Introduction to Parallel Programming

As transistors shrink, you can fit more of them on one chip and each one requires less energy.

These "die shrinks" have driven computing power for 40 years.

However, most people think that we cannot keep shrinking transistor sizes.

Recently, there has been a move from single-core processors to multi-core processors in order to mitigate the die-shrink issue.

Introduction to Parallel Programming

The fundamental limitation to parallel programming:

Amdahl's Law

Letting P denote the number of processors and α denote the fraction of running time a program spends running on non-parallelizable parts, then the maximum speedup possible is

$$\lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}.$$

E.g., if you can only parallelize 50% of the code, can at most speed by 2x.

Profilers then will let you know what to parallelize.

"Embarrassingly parallel" is $\alpha \approx 1$.

Two lessons from Amdahl's law:

1. Parallelizing the most costly sections of code will have the greatest effect (because it makes α small).
2. Additional processors have diminishing marginal benefit.

How big can P be?

Machine	P
CPU	
Laptop	1-4
Desktop	2-6
Workstation	4-32
Node of a cluster	10-32
Super computers	1000-30000+
GPU	
GPU (Nvidia GTX 980)	2048
GPU (Nvidia Titan Z)	5760

Introduction to Parallel Programming

GPU computing is most efficient if code requires no "if" statements.

In my experience, GPUs are not that great b/c of this.

Best applications: Large linear algebra problems (e.g., Kalman filtering).

If you want to try GPU computing, lowest cost may be Matlab with GPU toolbox.

Parallel Programming

MPI

- MPI (Message Passing Interface) can require much more work non-comment code will run.
- It will only compile and run if MPI is installed.
- Each node has its own memory (distributed memory). So they cannot see each other's data.
- Data must be sent back and forth over a network (this greatly reduces speed).
- Works on many PCs networked together into a cluster.

Sample MPI code:

```
! Get chunks of A from all nodes and redistribute
call MPI_Allgatherv(&
  A(1,lo_iproc),& ! first element assigned to ipr
  nrow*nelts(iproc+1),& ! total # of elts assigne
  MPI_DOUBLE_PRECISION,& ! data type of send elem
  A(1,1),& ! first element of receive
  nrow*nelts,& ! number of elements to be receive
  offset,& ! entry iproc specifies the displaceme
  MPI_DOUBLE_PRECISION,& ! data type of receive e
  MPI_COMM_WORLD,&
  ierr)
```

Parallel Programming

OpenMP

- Very easy to add to Fortran do loops.
- Only worthwhile if a loop is doing a lot of work.
- Uses comments (called pragmas in C) that the compiler can read or ignore depending on what you specify.
- All cores can see the same data (shared memory).
- Works only on a single PC.

Parallel Programming

Sample OpenMP code:

```
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(iE,iA)
do iE = 1,nE
     do iA = 1,nA
          ! Do some work
          c(iA,iE) = V(iA,iE) + 12d0
          call sub_bellman(V(iA,iE),c(iA,iE),W)
          ...
     end do
end do
!$OMP END PARALLEL DO
```

OpenMP vs MPI

	OpenMP	MPI
Compilers	Universal	Library
Ease of coding	Easy	Harder
Code readibility	Superb	Not great
Scalability	1 node	Unlimited
Memory sharing	Low cost	Costly
Help	Good	Good

OpenMP

OpenMP

OpenMP has code divided into sequential and parallel sections.

```
program foo
  real(8) :: x,y
  integer :: i
  ! Sequential
  ...
  !$OMP PARALLEL SHARED(x) PRIVATE(y,i)
    ! Parallel
    ...
  !$OMP END PARALLEL
  ! Sequential
  ...
end program
```

At the start of the code, the code is sequential (one thread) until a parallel section is met. Then it splits into threads and recombines at the end of the parallel section.

OpenMP

OpenMP has two basic types of data:

Shared	All threads can see and modify the data.
Private	Each thread creates its own copy of the data. The value is undefined at the beginning of the parallel section and afterwards.

There are variations such as Firstprivate and Lastprivate that guarantee values at the beginning and end of the parallel section.

OpenMP

For the best performance, minimize the number of private arrays.

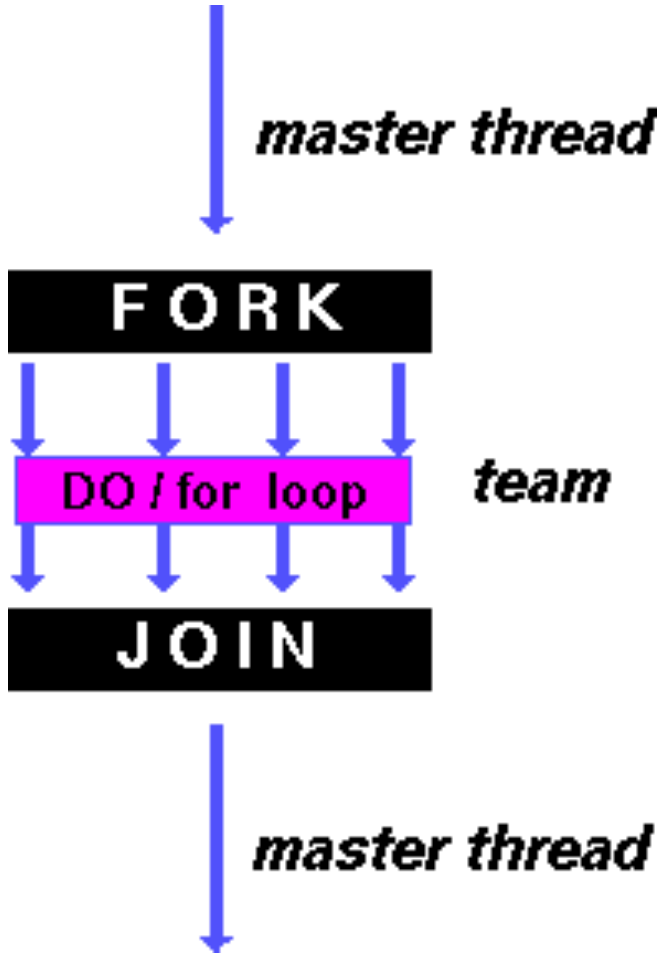
Generally, you are going to get the best performance when you have very only scalar private variables.

Note: in the examples that follow, the parallel code will often be slower because there is too little work being done to compensate for the overhead. But this is not the case for things like updating a value function.

OpenMP

Within a parallel section, there are different ways of sharing work among threads.

By far the most common is a `do` loop:



OpenMP

Here is an example of a parallel do loop:

```
integer, parameter :: n = 100 ! shared
real(8) :: x(n), y(n), z(n) ! shared or private
...
!$OMP PARALLEL DO SHARED(x,y,z)
do i = 1,n
  z(i) = sqrt(x(i)**2 + y(i)**2)
end do
!$OMP END PARALLEL DO
```

In the above, `i` is assumed private since it is the do loop variable.

This is equivalent to the following:

```
integer, parameter :: n = 100 ! shared
real(8) :: x(n), y(n), z(n) ! shared or private
...
!$OMP PARALLEL SHARED(x,y,z)
!$OMP DO
do i = 1,n
  z(i) = sqrt(x(i)**2 + y(i)**2)
end do
!$OMP END DO
!$OMP END PARALLEL
```

OpenMP

How could compute the sum of a vector `A` in parallel?

However, you can also do it quite easily on your own:

```
b = 0d0
!$OMP PARALLEL DO SHARED(A,b) REDUCTION(+:b)
do i = 1,n
  b = b + A(i)
end do
!$OMP END PARALLEL DO
```

The default initialization of `b` is `0d0`, but you should make it explicit so that the code runs properly in serial.

Again: This parallel code will be SLOWER than serial, b/c not enough work done by each thread.

OpenMP

What else can be done in a reduction?

You can do `REDUCTION([operator]:var)` where `operator` is one of

```
+,*,-,.and.,.or.,.eqv.,.neqv.,max,min
```

The initializations of the shared variable `var` are what they should be.

For instance, a reduction with $*$ would be something like $var = var * a(i)$, and the initialization of var is `1`. With `max`, you would be doing something like $var = max(var, a(i))$ and so the initialization is the most negative number possible.

OpenMP

Try programming this in parallel using a reduction:

$$V = \max_{i \in \{1,\ldots,n\}} u(c(i)) + W(i)$$

Assume u is a function and that c and W are vectors.

OpenMP

```
V = -huge(0d0)
!$OMP PARALLEL DO SHARED(c,W,V,n) &
!$OMP PRIVATE(i) REDUCTION(max:V)
do i = 1,n
  V = max(V,u(c(i))+W(i))
end do
!$OMP END PARALLEL DO
```

Note that in the above example, I have used a continuation line.

OpenMP

I cheated on that last example. Why? Because we only had to keep track of the value function, not the policy function.

Now assume we need the value function and policy function associated with

$$V = \max_{i \in \{1,\ldots,n\}} u(c(i)) + W(i).$$

You cannot use a simple "reduction" command. However, you can do a custom reduction.

OpenMP

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,Vtmp,tmp,itmp)
Vtmp = -huge(0d0); itmp = -1
!$OMP DO
do i = 1,n
  tmp = u(c(i))+W(i)
  if (tmp>Vtmp) then
    Vtmp = tmp
    itmp = i
  end if
end do
!$OMP END DO

Vbest = -huge(0d0)
!$OMP CRITICAL
if (Vtmp>=Vbest) then
  Vbest = Vtmp
  ibest = itmp
end if
!$OMP END CRITICAL
!$OMP END PARALLEL
```

OpenMP

One restriction with reductions is that the reduction variable must be a scalar.

But what if you want to compute

$$V(z) = \max_{i \in \{1,\ldots,n\}} u(c(i,z)) + W(i,z)$$

for each z?

OpenMP

```
V = -huge(0d0)
!$OMP PARALLEL DO SHARED(c,W,V,n,nz) &
!$OMP PRIVATE(i,iz)
do iz = 1,nz
  do i = 1,n
    V(iz) = max(V(iz),u(c(i,iz))+W(i,iz))
  end do
end do
!$OMP END PARALLEL DO
```

OpenMP

This code seems pretty efficient.

In fact, it may not be because of something called false sharing.

In the code, each thread is accessing logically distinct pieces of memory, `V(iz)`,`c(i,iz)`,`W(i,iz)`.

However, in order for `V(iz)` to be accessed, the computer may have to use a "lock" that prevents nearby values, like `...,V(iz-1),V(iz+1),...` etc from being accessed until `V(iz)` has been accessed.

This can result in horrible performance (e.g., the parallelized code being slower then the sequential!).

Reducing false sharing #1

How can one reduce the risk of false sharing in our example?

```
!$OMP PARALLEL DO SHARED(c,W,V,n,nz) &
!$OMP PRIVATE(i,iz,vtmp)
do iz = 1,nz
  vtmp = -huge(0d0)
  do i = 1,n
    vtmp = max(vtmp,u(c(i,iz))+W(i,iz))
  end do
  V(iz) = vtmp
end do
!$OMP END PARALLEL DO
```

A small change, but it can have a huge performance impact.

Reference: Dr Dobbs.

Reducing false sharing #2

An alternative way is to specify a chunk size.

```
V = -huge(0d0)
!$OMP PARALLEL DO SHARED(c,W,V,n,nz) &
!$OMP PRIVATE(i,iz) SCHEDULE(STATIC,40)
do iz = 1,nz
  do i = 1,n
    V(iz) = max(V(iz),u(c(i,iz))+W(i,iz))
  end do
end do
!$OMP END PARALLEL DO
```

This prevents threads from writing to close to each others' data most of the time.

Note that here if `nz` is less than 40, only one thread will execute the code! Need to choose chunk size so threads work on different parts but still have enough work to do.

OpenMP

What happens if you need to call a function inside a parallel section?

```
integer, parameter :: n = 100 ! shared
real(8) :: x(n), y(n) ! shared or private
...
!$OMP PARALLEL DO SHARED(x,y) PRIVATE(i)
do i = 1,n
  call workonx(x(i),y)
end do
!$OMP END PARALLEL DO
```

This code will always be "work" as long as `workonx` satisfies some restrictions.

Unfortunately, in my experience it is slower than literally coding workonx inside the parallel. This may change.

OpenMP

Also, to make your life easier, you can declare a default type, either SHARED or PRIVATE:

```
integer, parameter :: n = 100 ! shared
real(8) :: x(n), y(n) ! shared or private
...
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i)
do i = 1,n
  x(i) = maxval(y(1:i))
end do
!$OMP END PARALLEL DO
```

Normally, I declare everything SHARED. Then look for things to the LHS of an equal sign and determine should this be SHARED or PRIVATE.

OpenMP

```
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i)
do i = 1,n
  do j = 1,n
    x(:,i) = sqrt(i + y(j))
  end do
end do
!$OMP END PARALLEL DO
```

In the above, since x is indexed by the OMP thread index, it should be shared.

This is what Matlab calls a sliced variable.

At least a few versions ago, every variable had to be sliced in a parloop.

OpenMP

If no `DEFAULT` clause is specified, the behavior is the same as if `DEFAULT(SHARED)` was specified.

OpenMP

Parallel programming introduces a new class of bugs called **race conditions**.

A race condition is when the result depends on the scheduling of the threads or the speed at which each thread executes.

Here is an example of a race condition:

```
!$OMP PARALLEL DO SHARED(A)
do i = 1, 999
  A(i) = A(i+1)
enddo
!$OMP END PARALLEL DO
```

Here, the assignment of `i` values to threads and the speed will affect the result.

OpenMP

Note that this code might work.

```
!$OMP PARALLEL DO SHARED(A)
do i = 1, 999
  A(i) = A(i+1)
enddo
!$OMP END PARALLEL DO
```

E.g., suppose there are four threads. Then this will "work" if

- thread `j` is assigned `i` when `j==mod(i,4)`
- and the threads complete their executions in the order 0,1,2,3,0,1,2,3,0,....

This shows the sinister and characteristic feature of race conditions: Sometimes the code works, and sometimes it doesn't.

OpenMP

A subtle issue with OpenMP is that even if there is no bug, the results from the parallel code may not be exactly the same as the serial.

```
a = 0d0
!$OMP PARALLEL DO SHARED(a) REDUCTION(+:a)
do i = 0,2
  if (i==0) then
    a = a + 1d0
  elseif (i==1) then
    a = a + sqrt(huge(0d0))
  elseif (i==2) then
    a = a + (-sqrt(huge(0d0)))
  end if
end do
!$OMP END PARALLEL DO
```

What is a if the i order is 2,1,0? How about 0,1,2?

OpenMP

I have tried to highlight some of the most useful features of OpenMP.

There are many other feature that I have not covered.

References:

- `https://computing.llnl.gov/tutorials/openMP/`
- `http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf`

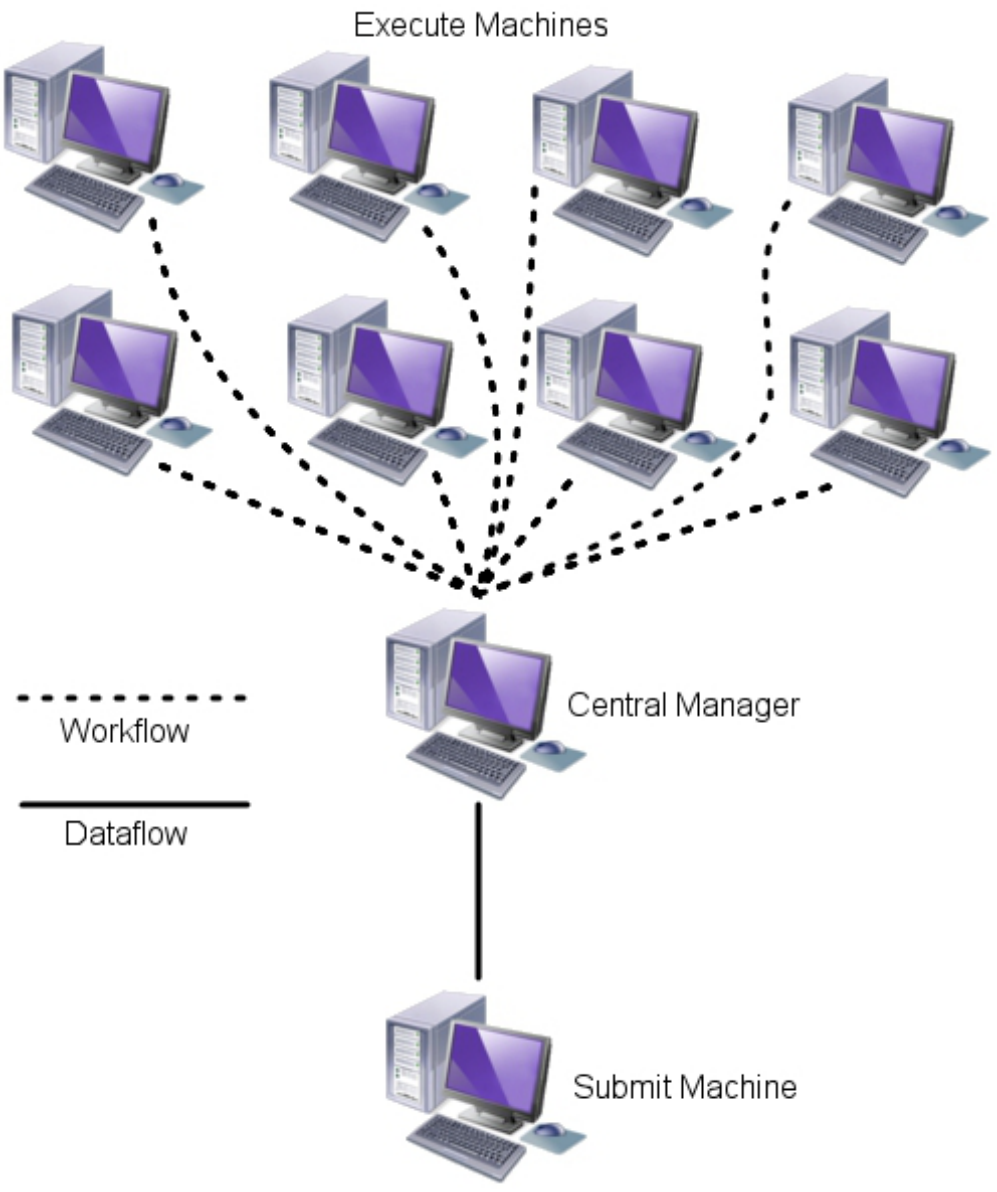
OpenMP vs MPI

OpenMP is limited to a single machine.

MPI (Message Passing Interface) is a collection of routines that lets multiple machines work on a problem.

MPI is the primary way to do distributed computing.

Distributed computing (supercomputer structure)



MPI

A key concept is an MPI **rank**.

If working with 4 16-core machines, could have

- 4 ranks (one for each machine)
- or 64 ranks (each core is a rank)
- or 8 ranks (8 cores on each machine form a rank)

Every MPI **rank** begins by executing the same program.

At some point, they will then split up and work on different problems.

Usually, they will then communicate that information with the rest of the program.

MPI

Here's the canonical hello world:

```
program main
    implicit none
    include 'mpif.h' ! lots of subroutines et al. declared
    integer :: i,n,ierr

    ! Do the required setup
    call MPI_INIT(ierr)

    ! Get the number of processors
    call MPI_COMM_SIZE(MPI_COMM_WORLD, n, ierr)

    ! Get the index of this rank i in {0,...,n-1}
    call MPI_COMM_RANK(MPI_COMM_WORLD, i, ierr)

    print*, 'Hello: I am rank ',i,' of ',n

    call MPI_FINALIZE(ierr)

end program
```

MPI

For instance, say trying to solve $V(i) = \max_j u(c(i,j)) + W(j)$.

Suppose W is a the output of the program up to some point

- E.g., let's say up to "line 627"

In an MPI program, every rank might do the same thing up to line 627, so they should all have the same W computed.

Then, every rank might take some of the i values.

Each rank computes the values it's responsible for.

Then, each rank shares the $V(i)$ it computed all the other ranks.

In the end, every rank has the same value for V.

MPI

Compute $V(i) = sqrt(i) + i^2$ for $i = 1, \ldots, 100$:

```
program simplecompute
  use mod_mympi ! easier procedures I wrote
  integer :: i,a,b; real(8) :: V(100)
  call mympi_init() ! initializes
  ! Split {1,...,100} into {a,...,b} for each
  ! rank (a/b are rank-specific)
  a = mympi_a(100)
  b = mympi_b(100)
  do i = a,b ! Only work on subset
    V(i) = sqrt(i) + i**2
  end do
  ! collects values from each rank and share
  ! with all. Requires mympi_a/b were used.
  call mympi_allgather(V)
  call mympi_finalize()
end program
```

MPI

In other applications, a "master" rank (MPI rank 0 usually) will distribute assignments to "workers" (MPI rank > 0).

E.g., $\max_x f(x)$, the master rank could be using a sophisticated algorithm to guess x values.

The workers could wait for an x from the master, compute f, and wait to return the value.

MPI can be very fast as long as work is large amount of data that needs to be passed.

Recommendations

I think the best strategy is this:

1. If calibrating/estimating, use MPI (or distributed computing toolbox in Matlab) to pass parameters around.
2. Given the parameters it was assigned, have each rank solve for the moments etc. using OpenMP.
   - If very large, I sometimes use MPI here, too.
3. Then, the rank sends back the moments/likelihood.

Very little info gets passed around this way: MPI fast.

And, get the advantages of OpenMP.

Coarse-grain parallelism

In your research, you will likely need to conduct robustness tests, solving the model for different assumptions.

One approach you might take here is writing the code to read in some value that will change the behavior.

For instance, `./a.out 2` might solve a model for risk aversion of 2 and and `./a.out 50` might solve for it risk aversion of 50.

In this case, you can do very "coarse-grain" parallelism. For instance, using Big Red II, you can submit two jobs, one running `./a.out 2` and one running `./a.out 50`.

How to actually use a super computer

Logging into the super computer:

- Windows: use Putty and login info from university instructions
- Mac: open "Terminal" and ssh in, something like "ssh rggordon@bigred2.iu.edu"

Once you're logged in, you'll need to use Linux.

When you login, you're in the login node. You don't run code there.

Key commands:

- ls: lists current folder contents
- cd: change directory / folder
- cp: copy file(s)/directories
- rm: remove file(s)/directories

For text editors, you'll need either Vim (vi) or Emacs pretty much.

How to actually use a super computer

Transferring files

- Windows: use WinSCP
- Mac: usually rsync, sometimes scp

Running jobs on the cluster

- Clusters usually have a scheduler, Torque or Slurm
- You submit job request(s) to the scheduler
- Scheduler decides when your code actually runs

Your code tends to be prioritized if you have lightly used the cluster in the last month.

How to actually use a super computer

Typical workflow (for me):

- Edit / debug code ONLY locally
- Use rsync to push all the (not synced) files to the cluster
- ssh into the cluster
- Compile the code
- Submit the code to a queue
- Check the output when it's finished