Advanced Macroeconomics II

Handout 4 - Optimization

Sergio Ocampo

Western University

October 14, 2020

Short recap

Prototypical DP problem:

$$V(k,z) = \max_{\{c,k'\}} u(c) + \beta E\left[V\left(k^{'},z^{'}\right)|z\right]$$
$$\text{s.t.}c + k^{'} = f(k,z)$$
$$z^{'} = h(z,\eta)\,;\eta \text{ stochastic}$$

- We are looking for functions $\mathbf{V}, \mathbf{g^c}, \mathbf{g^k}$: We cannot solve this

We need to solve an approximate problem:

1. Discretize state space (functions are now vectors)
2. Approximate continuous function: **Interpolation**
   - Requires "exact" solution of maximization problem: **Optimization**

Optimization - The problem

$$V\left(k,z\right)=\max_{\left\{c,k'\right\}} u\left(c\right)+\beta E\left[V\left(k^{'},z^{'}\right)|z\right]$$

Optimization - The problem

$$V\left(k,z\right)=\max_{\left\{c,k'\right\}} u\left(c\right)+\beta E\left[V\left(k^{'},z^{'}\right)|z\right]$$

Maximization:

- We can solve the problem directly looking for argmax of the RHS
- Local optimizers vs Global optimizers

Optimization - The problem

$$V\left(k,z\right)=\max_{\left\{c,k'\right\}} u\left(c\right)+\beta E\left[V\left(k^{'},z^{'}\right)|z\right]$$

Maximization:

- We can solve the problem directly looking for argmax of the RHS
- Local optimizers vs Global optimizers

Root finding:

- We ca solve the problem by looking at the FOC (Euler equation)
- We are looking for values that make the FOC be zero (hence the root)

Optimization - What is involved?

Regardless of the approach the steps are similar:

Optimization - What is involved?

Regardless of the approach the steps are similar:

1. Define an objective function (either $u(\cdot) + \beta V$, or the FOC)
2. Guess a solution (say a value for $\{c, k', \ell, \ldots\}$)
3. Evaluate the objective function in that solution
4. Update your guess if needed

Optimization - What is involved?

Some steps we will "outsource" to an optimizer:

Optimization - What is involved?

Some steps we will "outsource" to an optimizer:

1. Define an objective function (either $u(\cdot) + \beta V$, or the FOC)
2. Guess a solution (say a value for $\{c, k', \ell, \ldots\}$)
3. Evaluate the objective function in that solution
4. Update your guess if needed

Optimization - What is involved?

Some steps we will "outsource" to an optimizer:

1. Define an objective function (either $u(\cdot) + \beta V$, or the FOC)
2. Guess a solution (say a value for $\{c, k', \ell, \ldots\}$)
3. Evaluate the objective function in that solution
4. Update your guess if needed

⋆ We do have to help our optimizer with an initial guess of the solution

Optimization - What is involved?

Some steps we have to do on our own:

Optimization - What is involved?

Some steps we have to do on our own:

1. Define an objective function (either $u(\cdot) + \beta V$, or the FOC)
2. Guess a solution (say a value for $\{c, k', \ell, \ldots\}$)
3. Evaluate the objective function in that solution
4. Update your guess if needed

Optimization - What is involved?

Some steps we have to do on our own:

1. Define an objective function (either $u(\cdot) + \beta V$, or the FOC)
2. Guess a solution (say a value for $\{c, k', \ell, \ldots\}$)
3. Evaluate the objective function in that solution
4. Update your guess if needed

⋆ Evaluating the function is the key step

- Evaluating might require solving intermediate problems (c vs ℓ)
- Requires evaluation in points off the grid (interpolation)
- Requires taking expectations (we are not there yet)

Optimization - Packages

- A good overview in quantecon (click here)
- Native Julia optimization module (Optim.jl)
- Wrapper for C's NLopt functions (NLopt.jl)
- Root finding module for Julia (Roots.jl)

Optimization - Packages

- A good overview in quantecon (click here)
- Native Julia optimization module (Optim.jl)
- Wrapper for C's NLopt functions (NLopt.jl)
- Root finding module for Julia (Roots.jl)

Many of these will need derivatives (either for FOC or for speed)

- If you know the derivative it then use that. Always better
- If you do not know the derivative use ForwardDiff.jl

Optimization - Packages

- A good overview in quantecon (click here)
- Native Julia optimization module (Optim.jl)
- Wrapper for C's NLopt functions (NLopt.jl)
- Root finding module for Julia (Roots.jl)

Many of these will need derivatives (either for FOC or for speed)

- If you know the derivative it then use that. Always better
- If you do not know the derivative use ForwardDiff.jl

Ultimate source of all knowledge:

- Numerical Recipes: The Art of Scientific Computing by Press, Teukolki, Vetterling and Flannery

Local Optimizers

Local optimizers

- Local optimizers are generally faster than global methods
  - Drawback: We need to be "close to the solution"

Local optimizers

- Local optimizers are generally faster than global methods
  - Drawback: We need to be "close to the solution"
- To overcome drawback we invest in **bracketing the solution**
  - Bracketing with theory: steady state convergence, minimum consumption, time constraints
  - Numerical procedures (see section 10.1 of Numerical Recipes)

Warning

We want to maximize... but computer scientists always want to minimize

- Make sure to operate on the negative of your value function
- Be careful! Most early bugs in your code are a misplaced minus sign

Bracketing a minimum (in one dimension)

- A minimum is bracketed by three points: a, b, and c
  - Without loss we will have: $a < b < c$
- A minimum is bracketed if: $f(a), f(c) > f(b)$

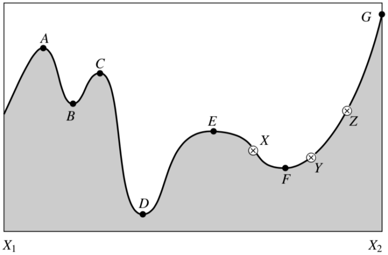


Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global minimum is at D. At point E, derivatives higher than the first vanish, a situation which can cause difficulty for some algorithms. The points X, Y, and Z are said to "bracket" the minimum F, since Y is less than both X and Z.

Bracketing a minimum

Theory:

- We know that $c^{\star} \in [\epsilon, f(k,z) - \underline{k}]$ for some $\epsilon, \underline{k} > 0$
- We know some problems are globally convergent so:
  - If $k \leq k_{ss}$ then $k'^{\star} \in [k, k_{ss}]$, otherwise $k'^{\star} \in [k_{ss}, k]$

Bracketing a minimum

Theory:

- We know that $c^\star \in [\epsilon, f(k,z) - \underline{k}]$ for some $\epsilon, \underline{k} > 0$
- We know some problems are globally convergent so:
  - If $k \leq k_{ss}$ then $k'^\star \in [k, k_{ss}]$, otherwise $k'^\star \in [k_{ss}, k]$

Numerically:

- Start with some "arbitrary" interval $[a, b]$
- Get c so as to get $b \in [a, c]$ (be smart in choosing c, read NR)
- Check if $f(a), f(c) > f(b)$
- Rinse and repeat

One-dimensional optimizers

To differentiate or not to differentiate: that is the question!

One-dimensional optimizers

To differentiate or not to differentiate: that is the question!

- A: Not to.

One-dimensional optimizers

To differentiate or not to differentiate: that is the question!

- A: Not to. We will get back to this.

One-dimensional optimizers

To differentiate or not to differentiate: that is the question!

- A: Not to. We will get back to this.

How do optimizers work?

- Bisection (Golden section)
- Parabolic interpolation (Brent)
  - Parabolic interpolation with derivative (dBrent)
- Higher polynomials (Do not use!)

Golden section

- Start with $a < b < c$ so that $f(a), f(c) > f(b)$
- Choose a point in $x \in [a, b]$ or $x \in [b, c]$, say you picked $x \in [b, c]$
- Keep x if $f(c), f(b) > f(x)$, x is the new candidate for a min

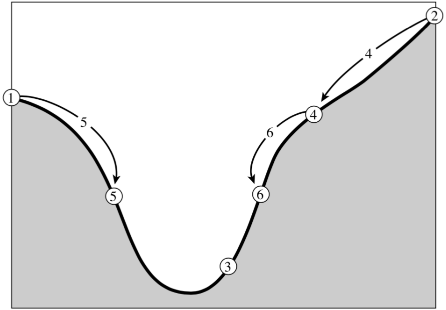


Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points. After the steps shown, the minimum is bracketed by points 5,3,6.

Golden section - How to pick intermediate point x?

- We can write b as a convex combination of a and b

$$b = (1-\gamma)\,a + \gamma c$$

Note: ${}^{b-a}/_{c-a} = \gamma$ and ${}^{c-b}/_{c-a} = 1-\gamma$

Golden section - How to pick intermediate point x?

- We can write b as a convex combination of a and b

$$b = (1-\gamma)\,a + \gamma c$$

  Note: ${}^{b-a}/_{c-a} = \gamma$ and ${}^{c-b}/_{c-a} = 1-\gamma$
- We want to place x an additional $\eta\,(c-a)$ beyond b: ${}^{x-b}/_{c-a} = \eta$

Golden section - How to pick intermediate point x?

- We can write b as a convex combination of a and b

$$b = (1-\gamma)\, a + \gamma c$$

  Note: ${}^{b-a}/_{c-a} = \gamma$ and ${}^{c-b}/_{c-a} = 1-\gamma$
- We want to place x an additional $\eta\,(c-a)$ beyond b: ${}^{x-b}/_{c-a} = \eta$
- Two options for new segment: $[a, x]$ or $[b, c]$
  - First segment will have length $(\gamma+\eta)\,|c-a|$, second $(1-\gamma)\,|c-a|$
  - If we want to min worst case (having too large of an interval)

$$\gamma + \eta = 1 - \gamma \longrightarrow \eta = 1 - 2\gamma$$

Golden section - How to pick intermediate point x?

- We can write b as a convex combination of a and b

$$b = (1-\gamma)\, a + \gamma c$$

  Note: ${}^{b-a}/_{c-a} = \gamma$ and ${}^{c-b}/_{c-a} = 1-\gamma$
- We want to place x an additional $\eta\,(c-a)$ beyond b: ${}^{x-b}/_{c-a} = \eta$
- Two options for new segment: $[a, x]$ or $[b, c]$
  - First segment will have length $(\gamma+\eta)\,|c-a|$, second $(1-\gamma)\,|c-a|$
  - If we want to min worst case (having too large of an interval)

$$\gamma + \eta = 1 - \gamma \longrightarrow \eta = 1 - 2\gamma$$

- Magic! x is symmetric to b! Note: $|b-a| = |c-x|$
  - For $\eta > 0$ we need to place x in the longest of $[a, b]$ or $[b, c]$

Golden section - How to pick ratio γ?

- If we use the same γ for all iterations this imposes *scale similarity:*

$$\frac{x-b}{c-b} = \frac{b-a}{c-a} \longrightarrow \frac{{}^{x-b}\!/_{c-a}}{{}^{c-b}\!/_{c-a}} = \frac{b-a}{c-a} \longrightarrow \frac{\eta}{1-\gamma} = \gamma$$

Golden section - How to pick ratio γ?

- If we use the same γ for all iterations this imposes *scale similarity:*

$$\frac{x-b}{c-b} = \frac{b-a}{c-a} \longrightarrow \frac{{x-b}/{c-a}}{{c-b}/{c-a}} = \frac{b-a}{c-a} \longrightarrow \frac{\eta}{1-\gamma} = \gamma$$

- Solution gives the golden ratio:

$$\gamma = \frac{3-\sqrt{5}}{2} \qquad 1-\gamma = \frac{1+\sqrt{5}}{2} - 1 = \frac{1}{\text{Golden Ratio}} \approx 0.618$$

recall that golden ratio is $\varphi \equiv {1+\sqrt{5}}/{2}$ and that it is the only number that satisfies ${1}/{\varphi} = \varphi - 1$

Golden section - How to pick ratio γ?

- If we use the same γ for all iterations this imposes *scale similarity:*

$$\frac{x-b}{c-b} = \frac{b-a}{c-a} \longrightarrow \frac{{}^{x-b}/_{c-a}}{{}^{c-b}/_{c-a}} = \frac{b-a}{c-a} \longrightarrow \frac{\eta}{1-\gamma} = \gamma$$

- Solution gives the golden ratio:

$$\gamma = \frac{3-\sqrt{5}}{2} \qquad 1-\gamma = \frac{1+\sqrt{5}}{2} - 1 = \frac{1}{\text{Golden Ratio}} \approx 0.618$$

recall that golden ratio is $\varphi \equiv {}^{1+\sqrt{5}}/_{2}$ and that it is the only number that satisfies ${}^{1}/_{\varphi} = \varphi - 1$

- Golden section guarantees that each new function evaluation will bracket min to an interval $\frac{1}{\varphi}$ the size of the preceding interval.

Why (and why not) golden section

- Guaranteed convergence to (local) minimum

Why (and why not) golden section

- Guaranteed convergence to (local) minimum
- Interval always includes minimum

Why (and why not) golden section

- Guaranteed convergence to (local) minimum
- Interval always includes minimum
- But convergence is constant: n^{th}−interval is $\left(1/\varphi\right)^n |a_0 - c_0|$
  - Typically takes more iterations than competing methods

Why (and why not) golden section

- Guaranteed convergence to (local) minimum
- Interval always includes minimum
- But convergence is constant: n^{th}−interval is $\left({}^{1}/_{\varphi}\right)^{n} |a_0 - c_0|$
  - Typically takes more iterations than competing methods

Question is not why, but when!

- When facing complicated problems is good to start with robust methods
- Start with golden section and then move to more complex method
- Stop early when $|a - c| < tol$ for some "large" tol.

(Inverse) Parabolic interpolation - Brent

- Basic idea is that a function can be locally parabolic (quadratic)
- We can use the formula for the abscissa x that is the minimum of a parabola to guess our new point (if function is actually quadratic we are done in one step!)

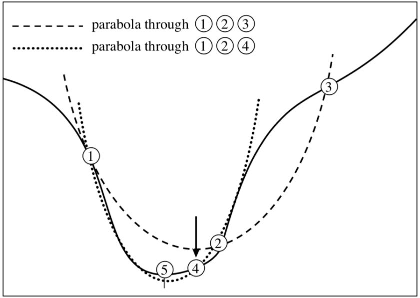


Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

(Inverse) Parabolic interpolation - Brent

- Brent uses the key step of inverse parabolic interpolation when it can
- Formula gives critical points only... you could be jumping to a max...
- Points can be collinear (an odd problem but it destroys the method)

(Inverse) Parabolic interpolation - Brent

- Brent uses the key step of inverse parabolic interpolation when it can
- Formula gives critical points only... you could be jumping to a max...
- Points can be collinear (an odd problem but it destroys the method)

Brent uses a combination of parabolic interpolation and golden search

- Preferred method in practice

(Inverse) Parabolic interpolation - Brent

- Brent uses the key step of inverse parabolic interpolation when it can
- Formula gives critical points only... you could be jumping to a max...
- Points can be collinear (an odd problem but it destroys the method)

Brent uses a combination of parabolic interpolation and golden search

- Preferred method in practice

Brent can be improved with derivatives:

- The sign of $f^{'}(b)$ indicates only whether the next test point should be taken in the interval (a, b) or in the interval (b, c).
- Avoids using derivatives in problematic ways

Problems with derivatives

One can use first and second derivatives to fit higher order polynomial to approximate (locally) and find critical points

Problems with derivatives

One can use first and second derivatives to fit higher order polynomial to approximate (locally) and find critical points

- New intervals are not guaranteed to contain the minimum

Problems with derivatives

One can use first and second derivatives to fit higher order polynomial to approximate (locally) and find critical points

- New intervals are not guaranteed to contain the minimum
- Small gains of speed in practice:

  "Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. [...] most function evaluations are spent in getting globally close enough to the minimum [...] we are more worried about all the funny "stiff" things that high-order polynomials can do, and about their sensitivities to roundoff error."

Problems with derivatives

One can use first and second derivatives to fit higher order polynomial to approximate (locally) and find critical points

- New intervals are not guaranteed to contain the minimum
- Small gains of speed in practice:

  "Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. [...] most function evaluations are spent in getting globally close enough to the minimum [...] we are more worried about all the funny "stiff" things that high-order polynomials can do, and about their sensitivities to roundoff error."
- Also general problems with derivatives in the computer:

  "too many functions whose computed "derivatives" don't integrate up to the function value and don't accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation"

Multi-dimensional optimizers

Try not to do it!

- Problems get significantly harder
- Algorithms get significantly less reliable

Multi-dimensional optimizers

Try not to do it!

- Problems get significantly harder
- Algorithms get significantly less reliable

Alternatives:

- Try to reduce your problem to a one-dimensional problem
    - Easy to do for consumption leisure
    - Analytical solutions are best

Multi-dimensional optimizers

Try not to do it!

- Problems get significantly harder
- Algorithms get significantly less reliable

Alternatives:

- Try to reduce your problem to a one-dimensional problem
  - Easy to do for consumption leisure
  - Analytical solutions are best
- Sequential solution
  - You might have to solve an auxiliary problem for each guess

Multi-dimensional optimizers - If you must

1. **Quasi-Newton:** Fast but only locally convergent (can backfire)
    - Use the BFGS algorithm
    - Key idea: Find an approximation of the Hessian, use that to get direction of improvement

Multi-dimensional optimizers - If you must

1. **Quasi-Newton:** Fast but only locally convergent (can backfire)
    - Use the BFGS algorithm
    - Key idea: Find an approximation of the Hessian, use that to get direction of improvement
2. **Nelder-Mead:** Slow but super reliable (also global properties)
    - Should be your first choice
    - Depends only on function evaluations
    - Use it to get you near a minimum then try other methods

Multi-dimensional optimizers - If you must

1. **Quasi-Newton:** Fast but only locally convergent (can backfire)
   - Use the BFGS algorithm
   - Key idea: Find an approximation of the Hessian, use that to get direction of improvement
2. **Nelder-Mead:** Slow but super reliable (also global properties)
   - Should be your first choice
   - Depends only on function evaluations
   - Use it to get you near a minimum then try other methods
3. **Derivative-Free Nonlinear-Least-Squares (DFNLS):** Potentially very good
   - I don't know much about these
   - Use BOVYQA algorithm - Powell-like algorithm without derivatives

Global Optimizers

Overview

- For (agent) optimization problems better to use local optimizers

Overview

- For (agent) optimization problems better to use local optimizers
- Global optimizers useful for estimation/calibration

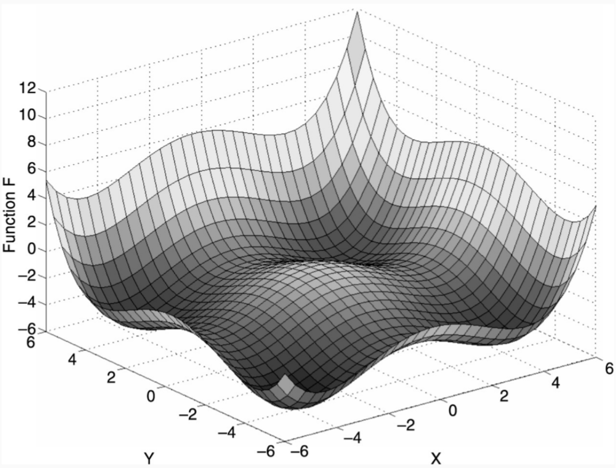
Overview

- For (agent) optimization problems better to use local optimizers
- Global optimizers useful for estimation/calibration
- Best initial algorithm: Nelder-Mead
    - Couple with **Simulated Annealing** to search for global solution

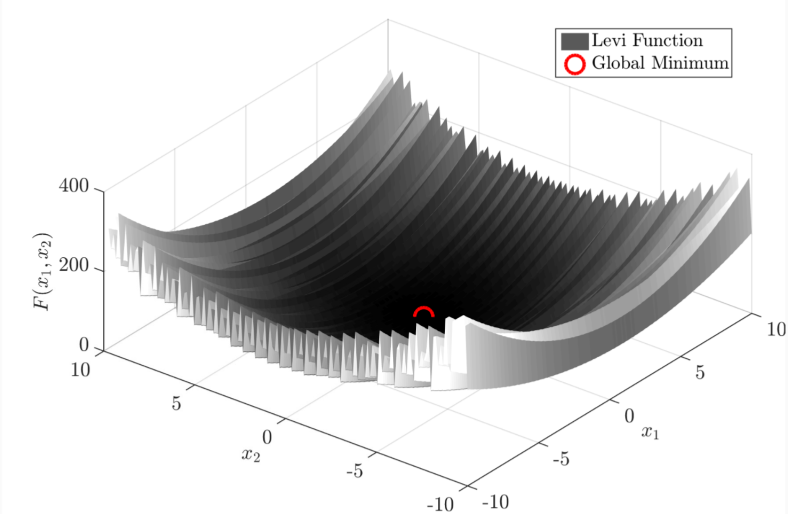
Overview

- For (agent) optimization problems better to use local optimizers
- Global optimizers useful for estimation/calibration
- Best initial algorithm: Nelder-Mead
  - Couple with **Simulated Annealing** to search for global solution
- Nice alternative: TikTak algorithm, see Arnaud, Guvenen & Kleineberg (2020)

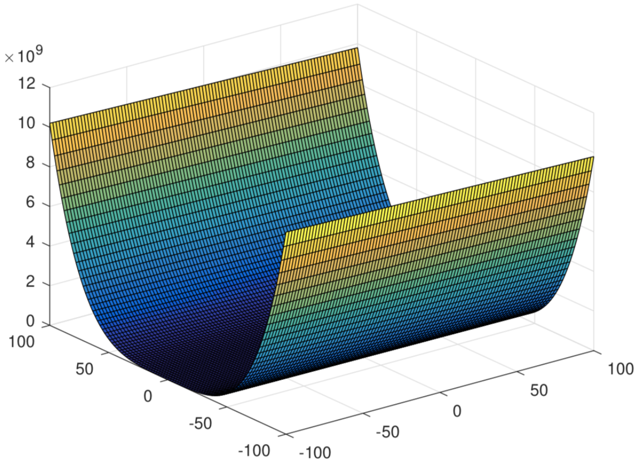
General problem: Functions are ugly



General problem: Functions are ugly



General problem: Functions are ugly



What do do about it?

- Back to analytical results
  - Try to show you actually have a nice looking function
  - For example with Bellman equation and consumption savings or portfolio choice under linear constraints

What do do about it?

- Back to analytical results
  - Try to show you actually have a nice looking function
  - For example with Bellman equation and consumption savings or portfolio choice under linear constraints
- Plot! You need to know how your function looks like
  - Level curves, heat maps

What do do about it?

- Back to analytical results
  - Try to show you actually have a nice looking function
  - For example with Bellman equation and consumption savings or portfolio choice under linear constraints
- Plot! You need to know how your function looks like
  - Level curves, heat maps
- Re-start your optimizer from "solution"
  - It will often move away

What do do about it?

- Back to analytical results
  - Try to show you actually have a nice looking function
  - For example with Bellman equation and consumption savings or portfolio choice under linear constraints
- Plot! You need to know how your function looks like
  - Level curves, heat maps
- Re-start your optimizer from "solution"
  - It will often move away
- Use various starting points for your optimization routine (costly)

Nelder-Mead

- Most robust method: The tractor to Newton's Ferrari
- Only requires function evaluations (but very many of them)

Nelder-Mead

- Most robust method: The tractor to Newton's Ferrari
- Only requires function evaluations (but very many of them)

Basic idea for N-dimensions:

- Start with a simplex described by $N+1$ vertices
  - Say x_0, $x_0 + \lambda e_1$,...,$x_0 + \lambda e_N$

Nelder-Mead

- Most robust method: The tractor to Newton's Ferrari
- Only requires function evaluations (but very many of them)

Basic idea for N-dimensions:

- Start with a simplex described by $N + 1$ vertices
  - Say x_0, $x_0 + \lambda e_1, \ldots, x_0 + \lambda e_N$
- The objective is to move the simplex to approach the minimum

Nelder-Mead

- Most robust method: The tractor to Newton's Ferrari
- Only requires function evaluations (but very many of them)

Basic idea for N-dimensions:

- Start with a simplex described by $N+1$ vertices
  - Say x_0, $x_0 + \lambda e_1, \ldots, x_0 + \lambda e_N$
- The objective is to move the simplex to approach the minimum
- Three ways to do it: reflections, expansions and contractions
  - Reflections change one vertex and send it in the direction of the lowest evaluation
  - Expansions and contractions can be in a given direction or in all directions

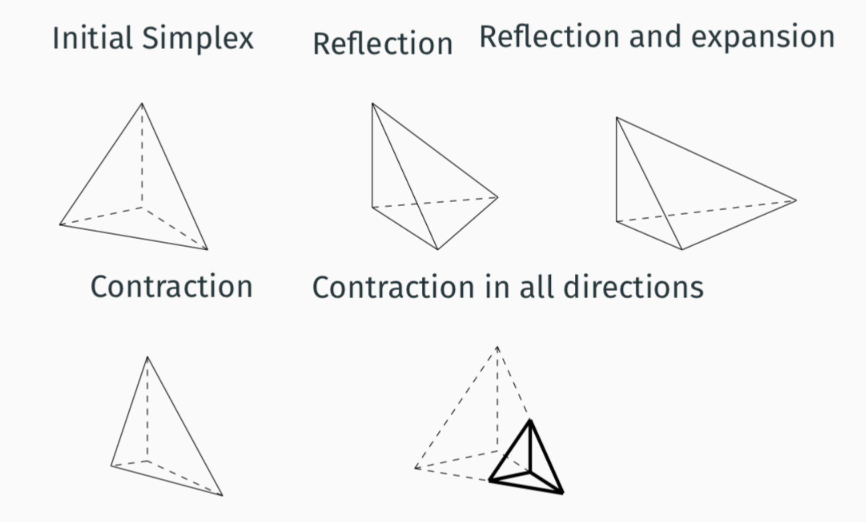
Nelder-Mead

- Most robust method: The tractor to Newton's Ferrari
- Only requires function evaluations (but very many of them)

Basic idea for N-dimensions:

- Start with a simplex described by $N+1$ vertices
  - Say x_0, $x_0 + \lambda e_1, \ldots, x_0 + \lambda e_N$
- The objective is to move the simplex to approach the minimum
- Three ways to do it: reflections, expansions and contractions
  - Reflections change one vertex and send it in the direction of the lowest evaluation
  - Expansions and contractions can be in a given direction or in all directions
- Easy to restart from a potential minimum

Nelder-Mead



Initial Simplex
Reflection
Reflection and expansion
Contraction
Contraction in all directions

TikTak

Algorithm 1: TikTak Global Optimizer

input : Number of seeds (N_0) and number of candidates ($N^\star$)
output: Global optimum of function F

1. Generate a sequence of N_0 quasi-random **Sobol numbers** ;
2. Evaluate the function in N_0 points. Keep the best $N^\star$

   $x_1, x_2, \ldots x_{N^\star}$ s.t. $F(x_1) \leq F(x_2) \leq \ldots \leq F(x_{N^\star})$;
3. Set $x^\star = x_1$ and $y^\star = F(x_1)$;

for *i=1:*$N^\star$ **do**

- 4.1. Let $\tilde{x}_0 = (1-\theta_i)x_i + \theta_i x^\star$ with $\theta_i \in [0, \bar{\theta}]$ is increasing in i, $\bar{\theta} < 1$;
- 4.2. Get local optimum: $\tilde{x} = \text{Optim}(F, \tilde{x}_0)$;
- 4.3. Update: $y^\star = \max\{y^\star, F(\tilde{x})\}$ and $x^\star$;

5. Return best result $(x^\star, F(x^\star))$

Random vs Quasi-Random Numbers

- We often have to evaluate functions “covering the space”
- This is pretty hard when we have many dimensions

Random vs Quasi-Random Numbers

- We often have to evaluate functions "covering the space"
- This is pretty hard when we have many dimensions

Options:

1. Make a (Hyper-)Cartesian grid
    - Costliest solution of all, and still leaves too much space empty

Random vs Quasi-Random Numbers

- We often have to evaluate functions "covering the space"
- This is pretty hard when we have many dimensions

Options:

1. Make a (Hyper-)Cartesian grid
   - Costliest solution of all, and still leaves too much space empty
2. Draw random numbers from some distribution (usually uniform)
   - Sounds better... but in practice has bad "space coverage"
   - Capricious behavior (where points end up located)

Random vs Quasi-Random Numbers

- We often have to evaluate functions "covering the space"
- This is pretty hard when we have many dimensions

Options:

1. Make a (Hyper-)Cartesian grid
   - Costliest solution of all, and still leaves too much space empty
2. Draw random numbers from some distribution (usually uniform)
   - Sounds better... but in practice has bad "space coverage"
   - Capricious behavior (where points end up located)
3. Quasi-random numbers: Deterministic sequences of numbers
   - Designed to spread maximally on a space
   - Build iteratively (next point in sequence fills out a portion of the space with less point density)

Other uses: Successive approximation

- Iterative nature of the sequence is really useful
- Lets you expand your grid in an orderly manner

Other uses: Successive approximation

- Iterative nature of the sequence is really useful
- Lets you expand your grid in an orderly manner

Example: Approximating a function (say to compute an integral)

- Start approximation with n_0 nodes
- Compute approximation with $n_0 + \Delta n$ nodes
- Compare results (say the difference in the integral)
- Increase nodes until result is stable

Other uses: Successive approximation

- Iterative nature of the sequence is really useful
- Lets you expand your grid in an orderly manner

Example: Approximating a function (say to compute an integral)

- Start approximation with n_0 nodes
- Compute approximation with $n_0 + \Delta n$ nodes
- Compare results (say the difference in the integral)
- Increase nodes until result is stable

Note: same idea as in Gauss-Kronrod quadrature integration

Other uses: (Quasi-)Monte Carlo integration

- Computing integrals by the Monte-Carlo method
  - Evaluate the function at many points and taking “expectations”

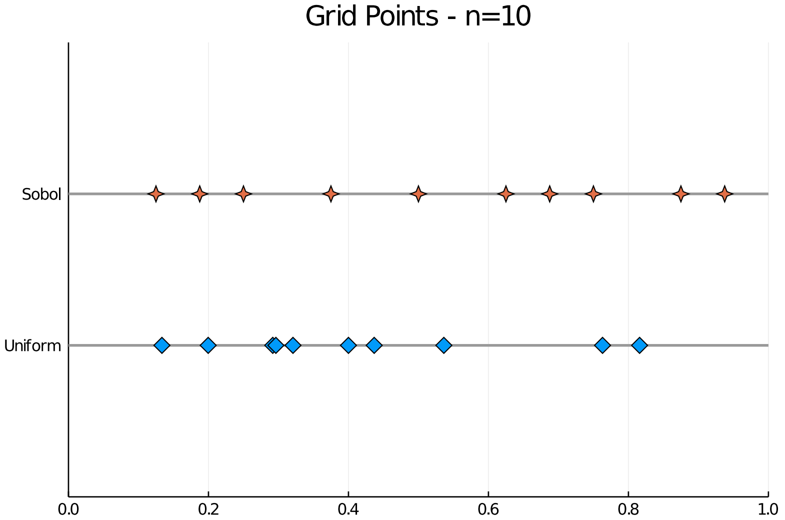
Other uses: (Quasi-)Monte Carlo integration

- Computing integrals by the Monte-Carlo method
  - Evaluate the function at many points and taking "expectations"
- Good integration requires points to "cover the space"
  - Monte Carlo is very robust but very demanding
  - Needs lots of points to get good evaluation

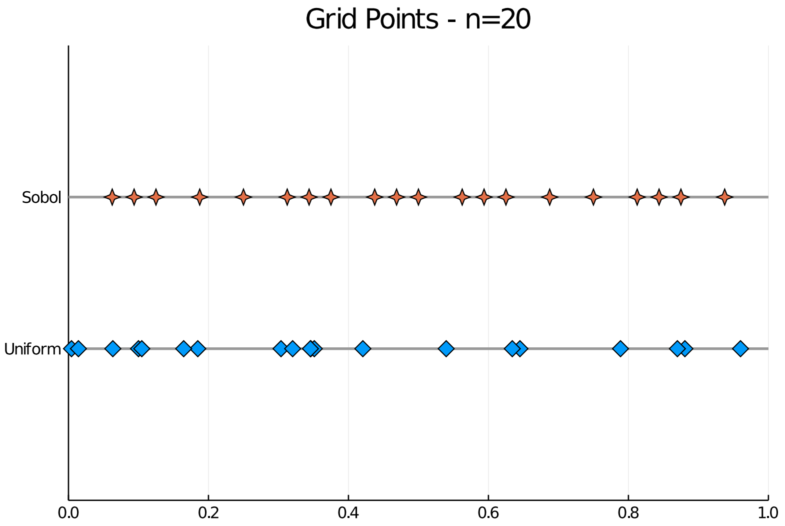
Other uses: (Quasi-)Monte Carlo integration

- Computing integrals by the Monte-Carlo method
  - Evaluate the function at many points and taking "expectations"
- Good integration requires points to "cover the space"
  - Monte Carlo is very robust but very demanding
  - Needs lots of points to get good evaluation
- Quasi-Monte Carlo integration uses quasi-random numbers
  - Increases convergence of integral
- Sobol numbers are particularly good

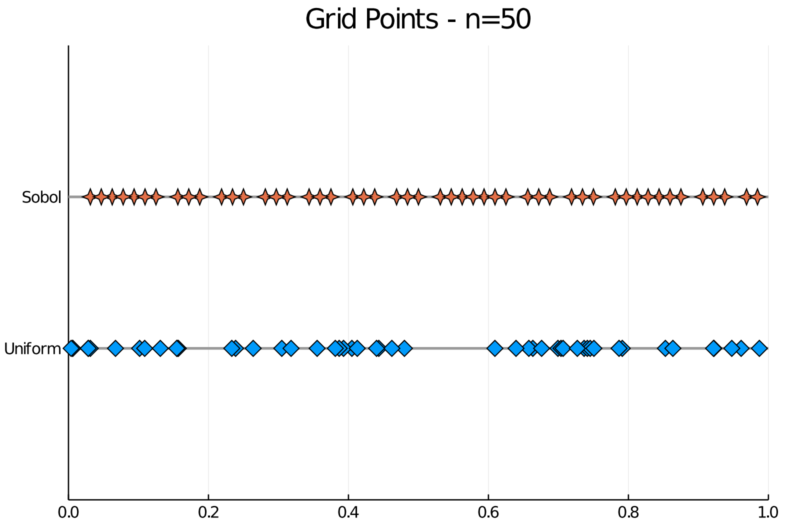
Sobol vs Uniform (1D)



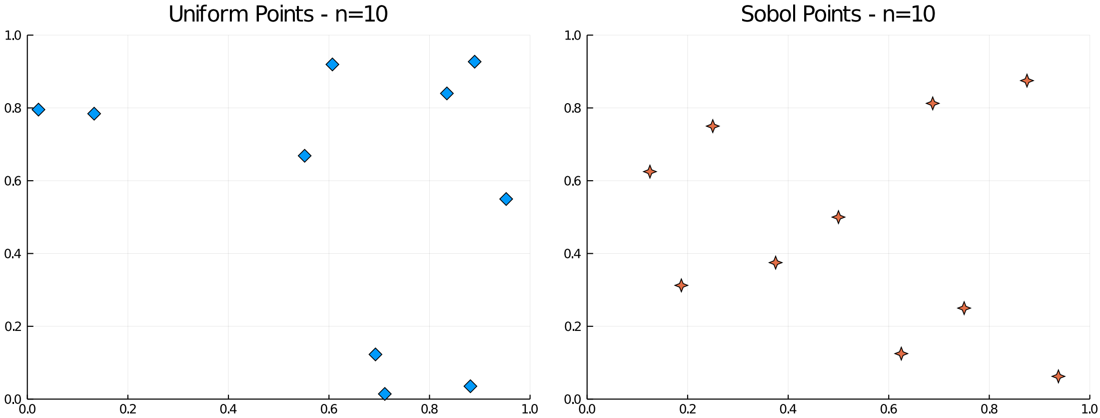
Sobol vs Uniform (1D)



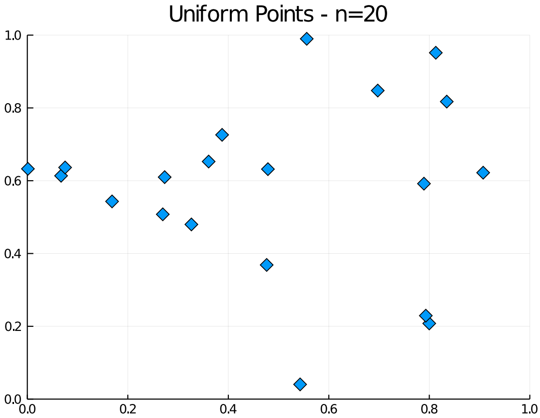
Sobol vs Uniform (1D)

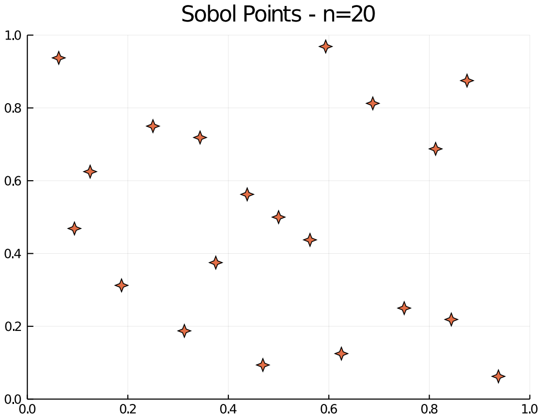


Sobol vs Uniform (2D) - Covering all the space

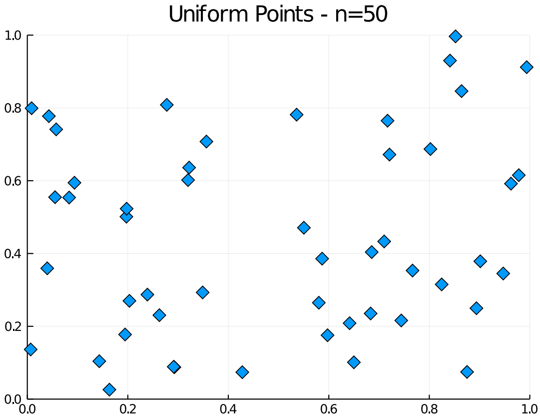


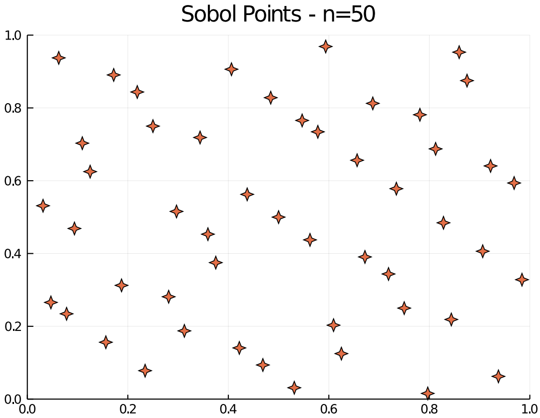
Sobol vs Uniform (2D) - Covering all the space



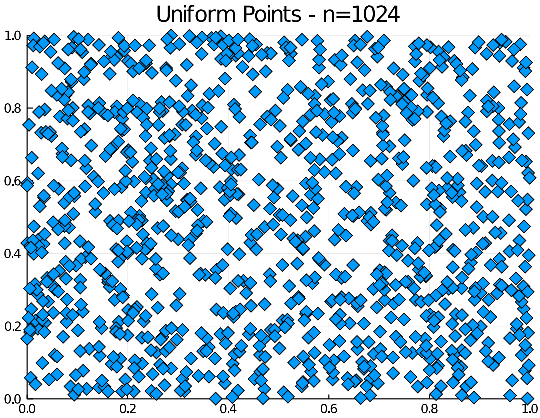


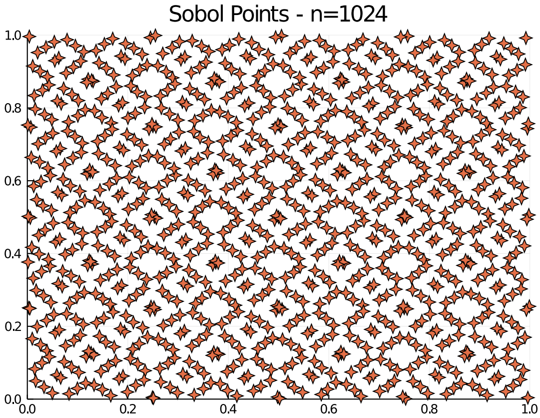
Sobol vs Uniform (2D) - Covering all the space





Sobol vs Uniform (2D) - Covering all the space





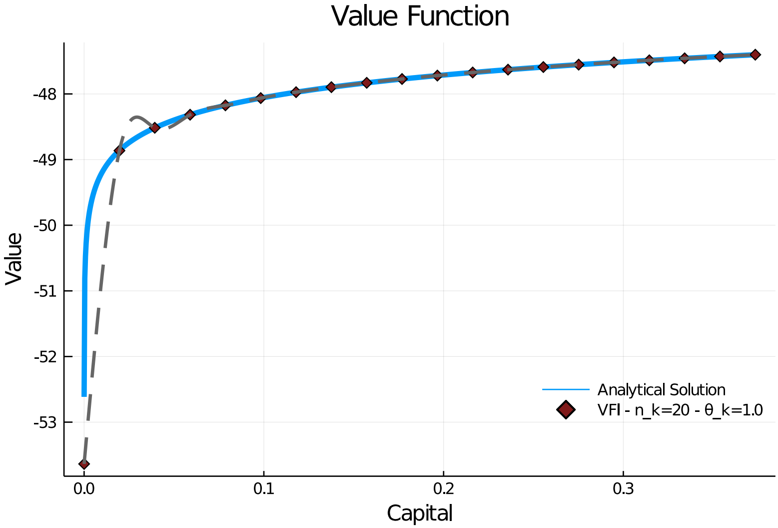
VFI with Continuous Optimization

VFI - Algorithm

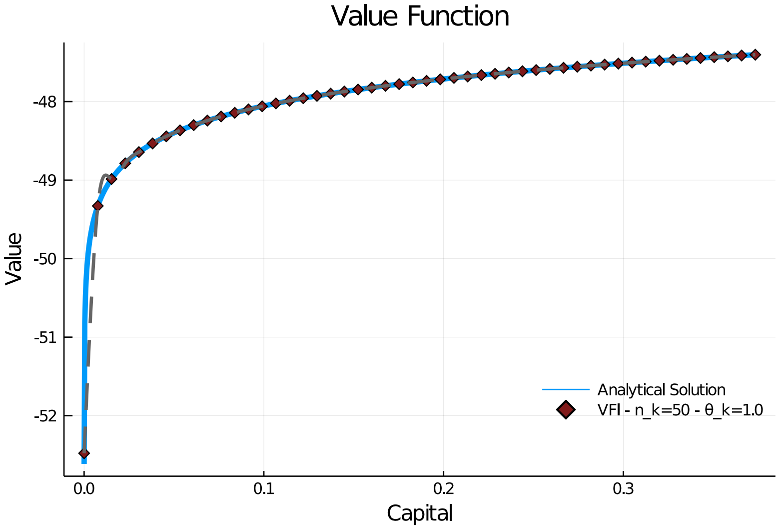
Algorithm 2: Bellman Operator: Continuous choice

```
Function T(V_old,k_grid,n_k,parameters):
    for i = 1:n_k do
        # Define objective function
        F(kp) = -U(k_grid[i],kp) - β*Vp(kp)
              Vp(kp) = Interpolation(k_grid,V_old,kp)
        # Find feasible range of kp
        k_min= 0;     k_max = z*k_grid[i]^α - c_min
        #Check for corner solutions with derivative at bounds
        kp = k_min if -dF(k_min)<0;     kp = k_max if -dF(k_max)>0;
        # Solve min (Optional: Further bracket min before minimizing)
        G_kp[i] = Optimize(F,k_min,k_max);    V_new[i] = -F(kp)
    return V_new, G_kp
```

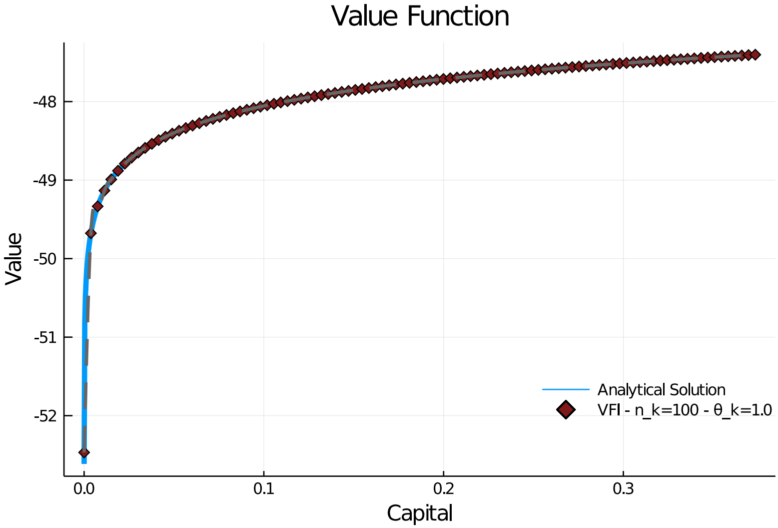
Value functions ($n_k = 20, 50, 100$)



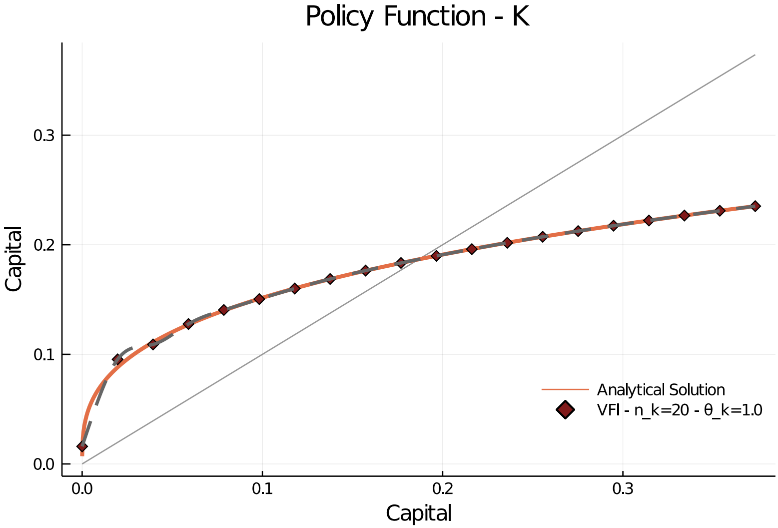
Value functions ($n_k = 20, 50, 100$)



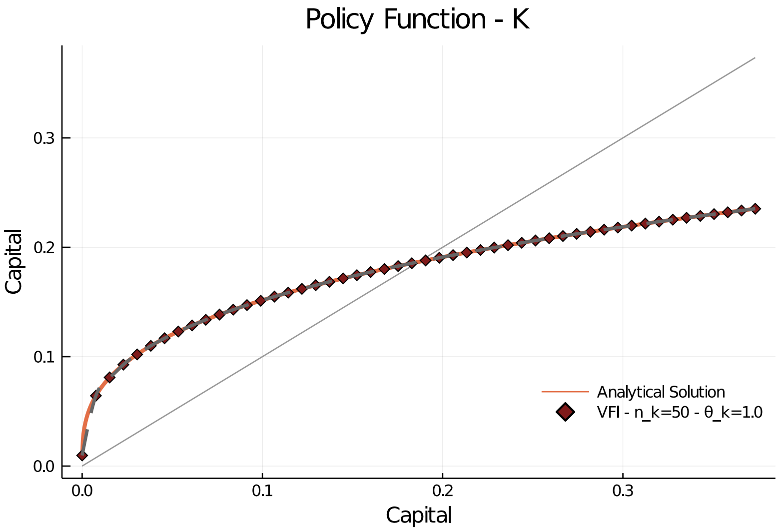
Value functions ($n_k = 20, 50, 100$)



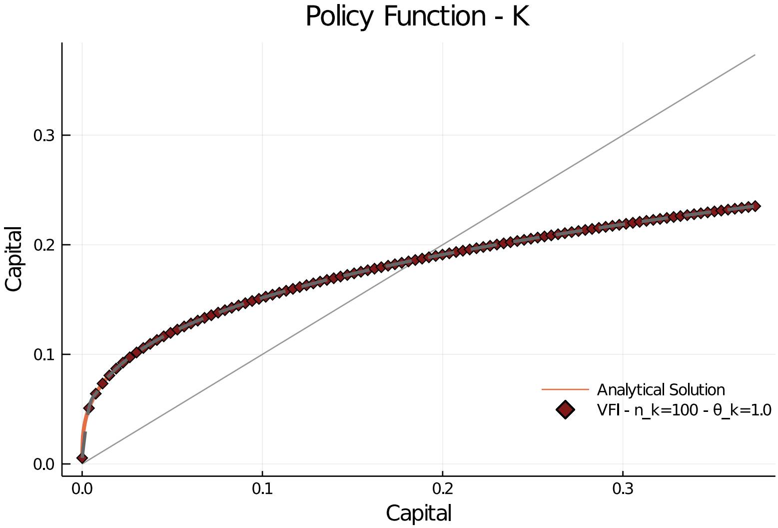
Policy functions ($n_k = 20, 50, 100$)



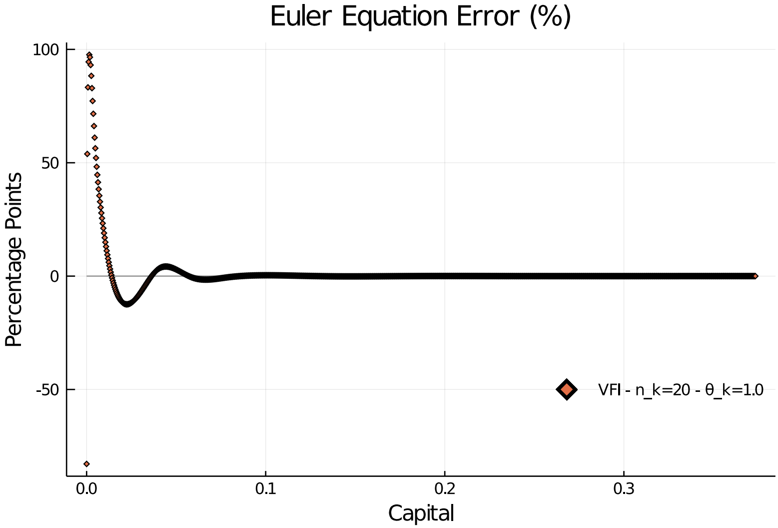
Policy functions ($n_k = 20, 50, 100$)



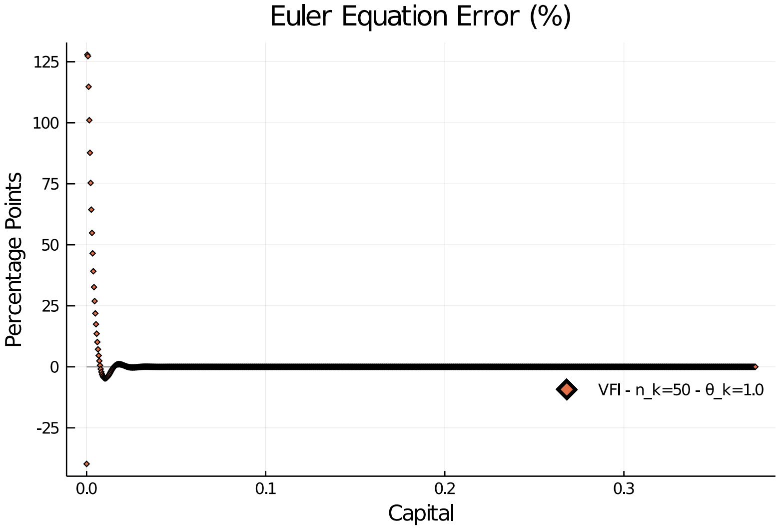
Policy functions ($n_k = 20, 50, 100$)



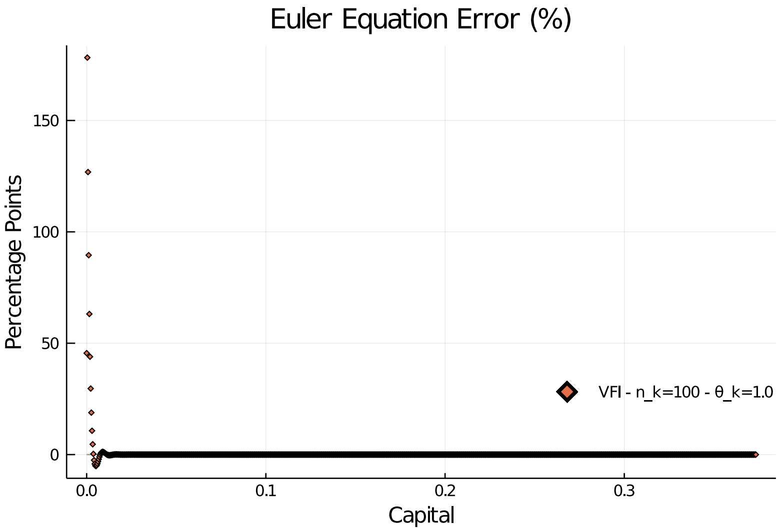
Euler Equation - Problems at the bottom



Euler Equation - Problems at the bottom

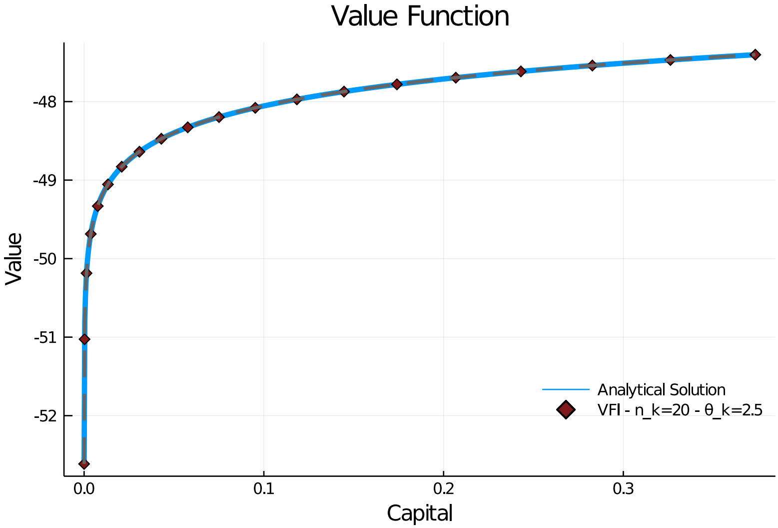


Euler Equation - Problems at the bottom

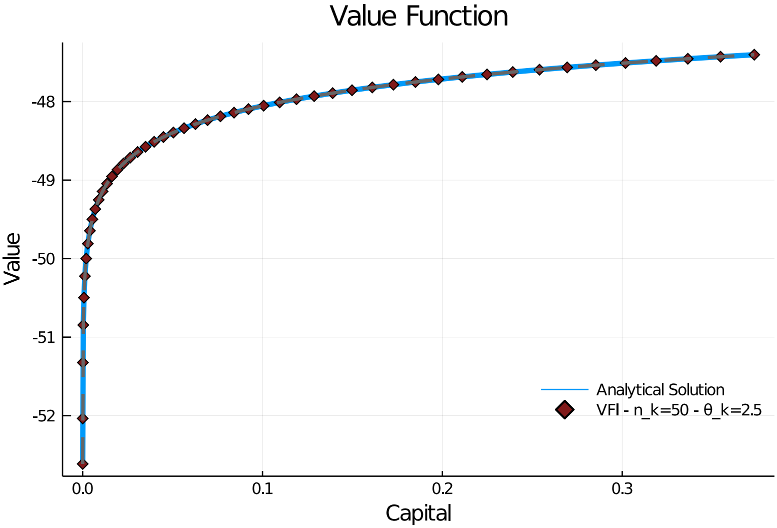


Revisiting Grid Spacing

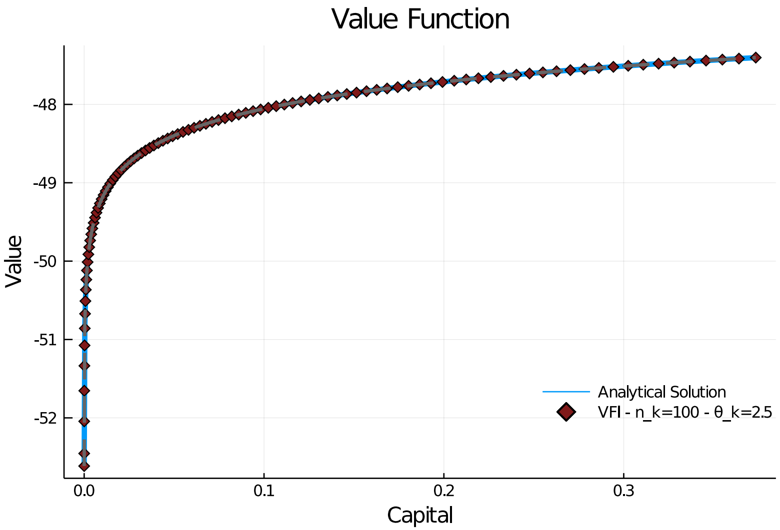
Value functions ($n_k = 20, 50, 100$)



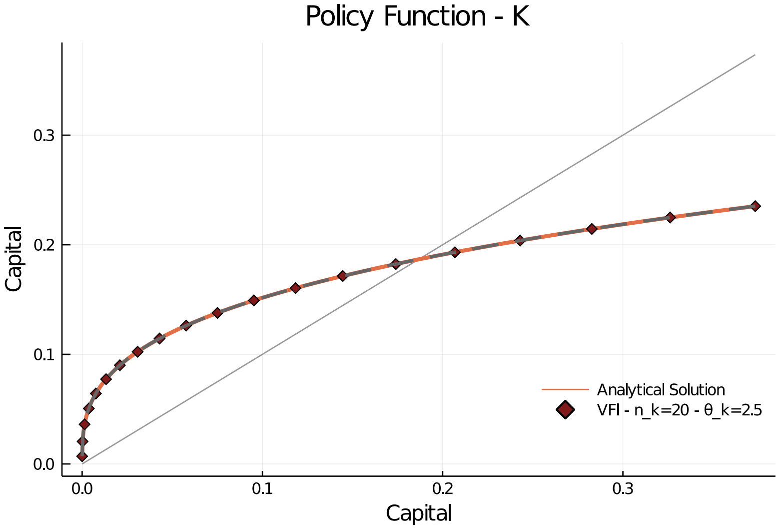
Value functions ($n_k = 20, 50, 100$)



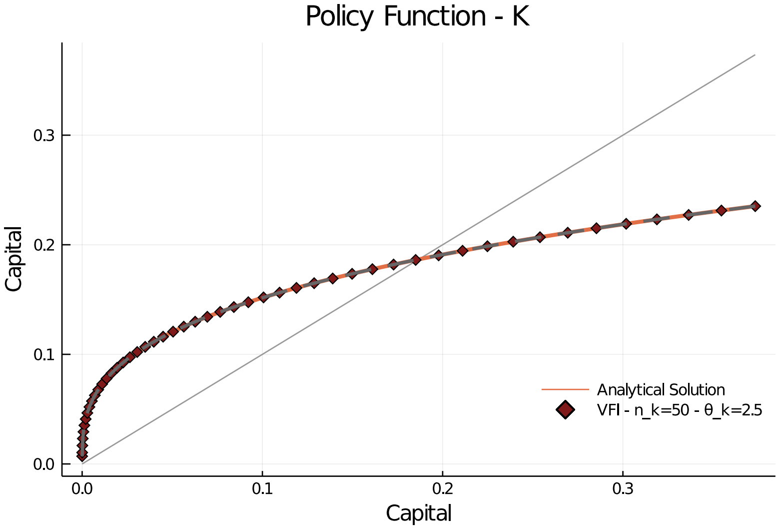
Value functions ($n_k = 20, 50, 100$)



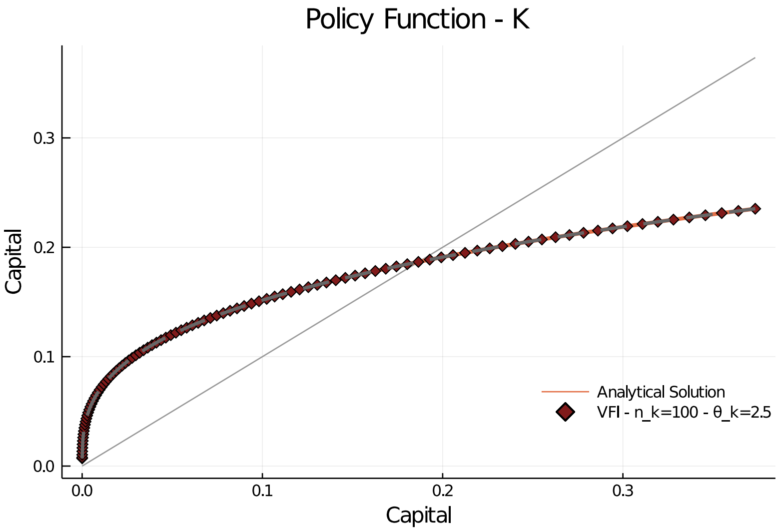
Policy functions ($n_k = 20, 50, 100$)



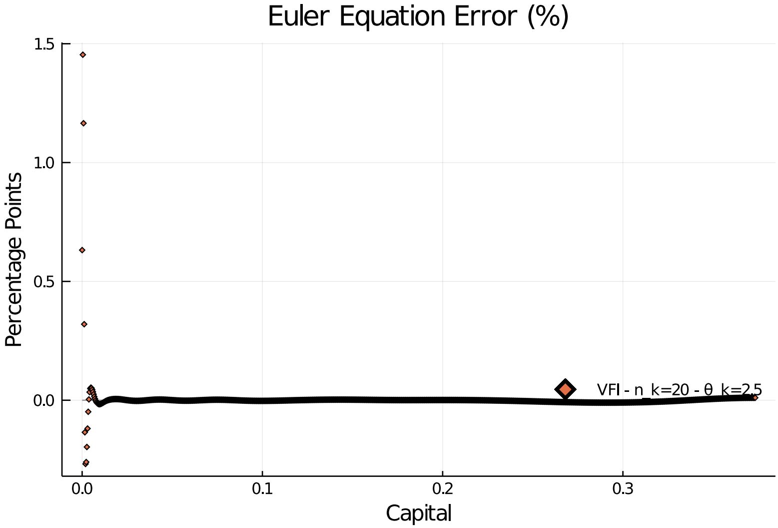
Policy functions ($n_k = 20, 50, 100$)



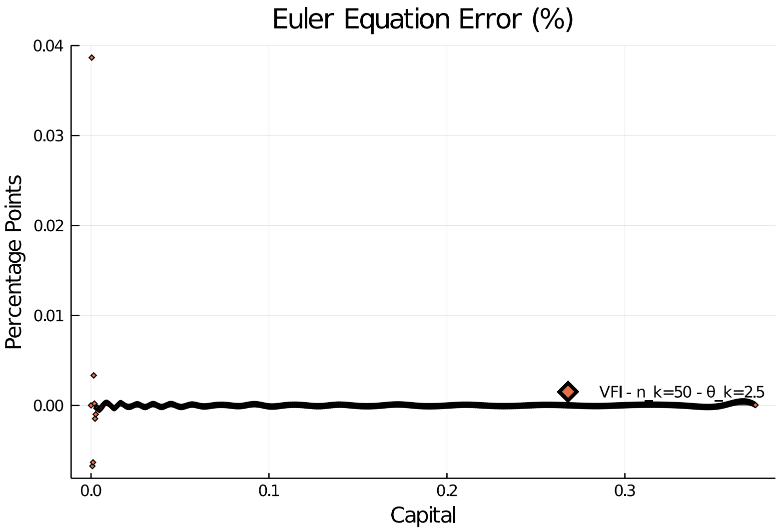
Policy functions ($n_k = 20, 50, 100$)



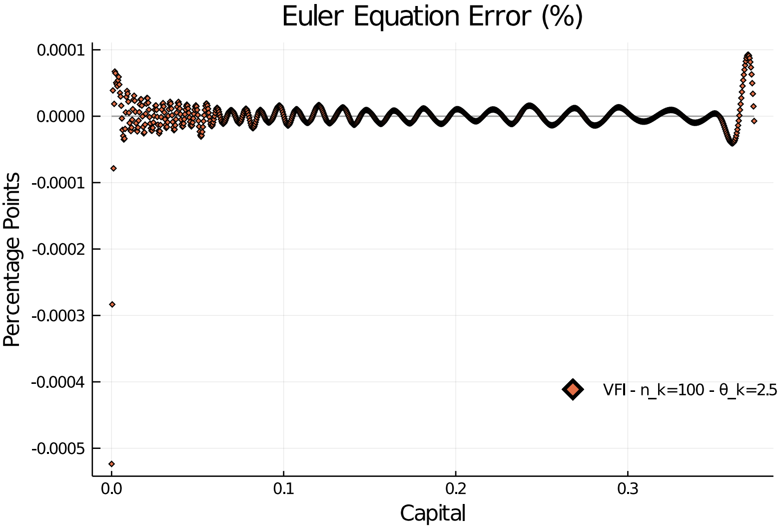
Euler Equation - No issues at the bottom



Euler Equation - No issues at the bottom



Euler Equation - No issues at the bottom



Root Finding

Overview

- Everything is very similar to what we have discussed
- Instead of maximizing the objective function we solve FOC
  - Find root of Euler equation

Overview

- Everything is very similar to what we have discussed
- Instead of maximizing the objective function we solve FOC
  - Find root of Euler equation
- In fact is common to use minimization methods for FOC
  - Minimize square of residual

Overview

- Everything is very similar to what we have discussed
- Instead of maximizing the objective function we solve FOC
  - Find root of Euler equation
- In fact is common to use minimization methods for FOC
  - Minimize square of residual
- Root finding is particularly useful to find equilibrium prices
  - Clear markets

Algorithms

- Bracketing: a, b such that $f(a) \cdot f(b) < 0$
  - You always want to make sure you bracketed a zero

Algorithms

- Bracketing: a, b such that $f(a) \cdot f(b) < 0$
  - You always want to make sure you bracketed a zero
- Bisection (use Golden Section):
  - Most robust method, preferred method for complex market clearing

Algorithms

- Bracketing: a, b such that $f(a) \cdot f(b) < 0$
  - You always want to make sure you bracketed a zero
- Bisection (use Golden Section):
  - Most robust method, preferred method for complex market clearing
- Brent (variants of the secant method):
  - Brent is the best (unless your problem is easy)

Algorithms

- Bracketing: a, b such that $f(a) \cdot f(b) < 0$
  - You always want to make sure you bracketed a zero
- Bisection (use Golden Section):
  - Most robust method, preferred method for complex market clearing
- Brent (variants of the secant method):
  - Brent is the best (unless your problem is easy)
- There are many other methods... depends on what you are doing

VFI - Algorithm

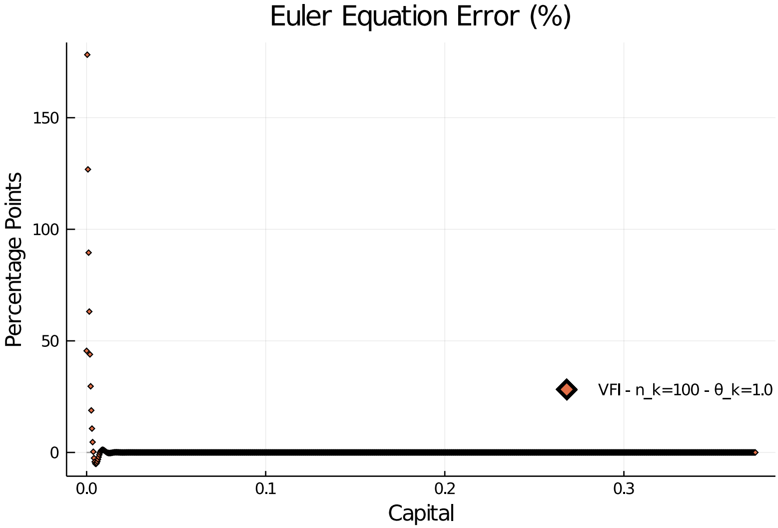
Algorithm 3: Bellman Operator: Continuous choice - Root finding

```
Function T(V_old,k_grid,n_k,parameters):
    for i = 1:n_k do
        # Define objective function
        F(kp) = -U(k_grid[i],kp) - β*Vp(kp)
        dF(kp) = dU(k_grid[i],kp) + β*dVp(kp)
            Vp(kp) = Interpolation(k_grid,V_old,kp)
        # Find feasible range of kp
        k_min= 0;    k_max = z*k_grid[i]^α - c_min
        #Check for corner solutions with derivative at bounds
        kp = k_min if -dF(k_min)<0;    kp = k_max if -dF(k_max)>0;
        # Solve min (Optional: Further bracket zero before minimizing)
        G_kp[i] = Roots(F,k_min,k_max);   V_new[i] = -F(kp)
```

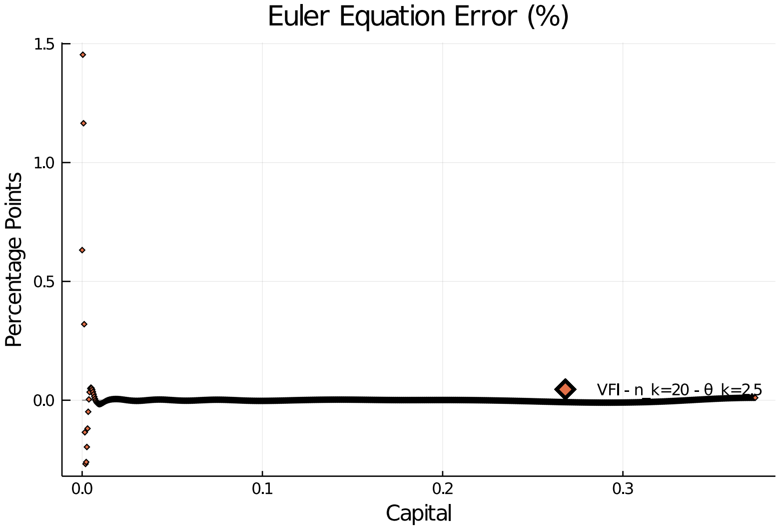
Euler Error ($n_k = 20, 50, 100$)

No Convergence for $n_k = 20, 50$ due to bad approximation to first derivative

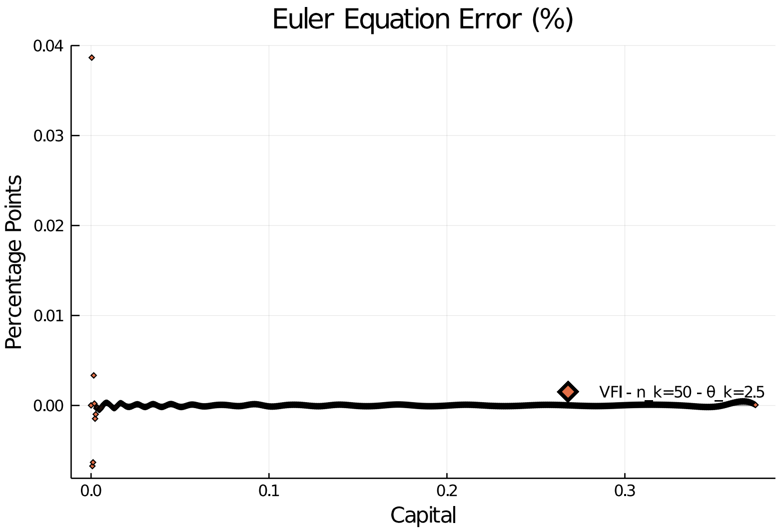
Euler Error ($n_k = 20, 50, 100$)



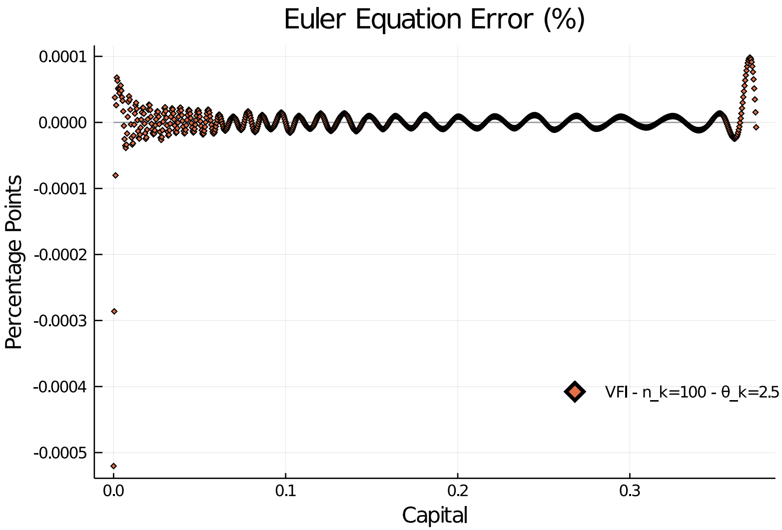
Euler Equation - Curved Grid



Euler Equation - Curved Grid



Euler Equation - Curved Grid



Two Choice Variables: Dealing with Labor Supply

Two choice variables

- Dealing with more than one choice variable complicates things

Two choice variables

- Dealing with more than one choice variable complicates things
- Part of the complication comes from the many options available

Two choice variables

- Dealing with more than one choice variable complicates things
- Part of the complication comes from the many options available

There are three main options:

1. Multi-dimensional maximization
2. Multi-dimensional root-finding
3. Nested problems

The problem

$$
\begin{aligned}
V(k) = \max_{\{c,k',\ell\}} \; & u(c,\ell) + \beta V\left(k'\right) \\
\text{s.t.} \quad & c + k' = f(k,\ell) \\
& \ell \in [0,1]
\end{aligned}
$$

First order conditions:

$$
\begin{aligned}
u_c(c,\ell) &= \beta V_k\left(k'\right) \\
-u_\ell(c,\ell) &= u_c(c,\ell)\, f_\ell(k,\ell) \\
c + k' &= f(k,\ell)
\end{aligned}
$$

The problem

$$
\begin{aligned}
V(k) = \max_{\{c,k',\ell\}} \; & u(c,\ell) + \beta V\left(k'\right) \\
\text{s.t.} \qquad & c + k' = f(k,\ell) \\
& \ell \in [0,1]
\end{aligned}
$$

First order conditions:

$$
\begin{aligned}
u_c(c,\ell) &= \beta V_k\left(k'\right) \\
-u_\ell(c,\ell) &= u_c(c,\ell)\, f_\ell(k,\ell) \\
c + k' &= f(k,\ell)
\end{aligned}
$$

Obviously eliminate consumption: $c = f(k,\ell) - k'$

Multi-dimensional maximization

Algorithm 4: Objective Function for Multi-Dimensinal Maximization

Function `Bellman_Objective`($k^{'}, \ell; k, V\_old, parameters$):

Compute implied consumption

$c = f(k, \ell) - k^{'}$

Check that consumption is allowable. If c is too low, use penalty

if $c < c\_min$ **then**

return F = penalty(c)

Evaluate Bellman objective

return $F = u(c, \ell) + \beta V\_old(k^{'})$

Note: You can pass V_old as a function (interpolation) rather than as a vector of values

Penalty Functions

- If the penalty is **too big**, it can completely throw off derivative-based methods and will at least confuse other methods.
  - Do not change the value of the objective function by orders of magnitude at the bound
- Always check for corner solutions before passing to a solver
  - Is $c = \underline{c}$ optimal if working at boundary?
- Avoid flat penalties, when function is equal to a "bad" value if violating constraint
  - Gives no information to solver... creates valleys

Penalty Functions

Better option: **Differentiable penalty functions**

1. Imagine the objective function is still defined at boundary, then:

$$P(c) = F(\underline{c}) + a(c - \underline{c})^2$$

   - Quadratic penalties are often enough, you can use something stronger
   - You can also use this if function is undefined at boundary by setting a base value close to the boundary

2. Alternative: Logarithmic barrier functions ()
   - See Boyd and Vandenberghe, 2013, Ch 11, Interior point methods
   - Approximate constrained optimization by adding a penalty
   - Instead of maximizing $F(x)$, maximize:

$$F(x) + \sum_{i=1}^{N} \frac{1}{t} \log(x_i)$$

   - Approximation improves with larger t

Multi-dimensional root-finding

Algorithm 5: Objective Function for Multi-Dimensinal Root-Finding

Function `FOC`($k', \ell; k, V\_old, parameters$)**:**

Compute implied consumption

$c = f(k, \ell) - k'$

Evaluate first order conditions

$F[1] = -u_c(c, \ell) + \beta V\_old_k(k')$

$F[2] = -u_\ell(c, \ell) + u_c(c, \ell) f_\ell(k, \ell)$

Note: You can pass V_old_k as a function

Return objective

return F (or return F.$\hat{2}$ for minimzation)

Penalties in root-finding

- It is harder to determine penalties in this case
- I am not aware of a standard practice
- You can always use the derivative from your differentiable penalty
  - Careful! Objective is differentiable but derivative is no longer continuous
- Best practice is still to check boundaries by hand

Nested problem - Outer function

Algorithm 6: Objective Function for Nested Optimization Problem

Function `Bellman_Objective`($k^{'}$;k,V_old,$parameters$):

  # Check consumption limits and apply penalty

  $c = f(k, 1) - k^{'}$

  if c<c_min **then**

    return F = penalty(c)

  # Solve for labor (inside bounds)

  $\ell^{\star} = Root(FOC_{\ell}; \underline{\ell}, 1, k^{'}, k)$ where $\underline{\ell} = max(\,0\,,\ f^{-1}(c\_min + k^{'}; k)\,)$

  # Compute implied consumption

  $c = f(k, \ell^{\star}) - k^{'}$

  # Evaluate Bellman objective

  return $F = u(c, \ell^{\star})) + \beta V\_old(k^{'})$

Nested problem - Inner function

Algorithm 7: Objective Function for Labor FOC

Function $FOC_\ell(\ell; k', k, V\_old, parameters)$**:**

Compute implied consumption
$c = f(k, \ell) - k'$

Evaluate first order condition
$F = -u_\ell(c, \ell) + u_c(c, \ell) f_\ell(k, \ell)$

Return objective
return F (or return F.2̂ for minimzation)

Why nest labor inside savings choice?

- Nesting makes the problem easier to solve
- But it is expensive because we solve too many inner problems
- Savings problem is harder to solve
  - Requires interpolation and potentially expectations
- Better to have the simpler problem be the inner problem