Lecture 04

Optimization

Ivan Rudik
AEM 7130

Software and stuff

Necessary things to do:

- Install the `QuantEcon` Julia package
- Install the `Optim` Julia package

Optimization

All econ problems are optimization problems

Optimization

All econ problems are optimization problems

- Min costs

Optimization

All econ problems are optimization problems

- Min costs
- Max PV E[welfare]

Optimization

Some are harder than others:

Optimization

Some are harder than others:

- Individual utility max: easy

Optimization

Some are harder than others:

- Individual utility max: easy
- Decentralized electricity market with nodal pricing and market power: hard

Optimization

Some are harder than others:

- Individual utility max: easy
- Decentralized electricity market with nodal pricing and market power: hard
- One input profit maximization problem: easy

Optimization

Some are harder than others:

- Individual utility max: easy
- Decentralized electricity market with nodal pricing and market power: hard
- One input profit maximization problem: easy
- N-input profit maximization with learning and forecasts: hard

Things we will do

1. Linear rootfinding
2. Non-linear rootfinding
3. Complementarity problems
4. Non-linear unconstrained maximization/minimization
5. Non-linear constrained maximization/minimization

Linear rootfinding

How do we solve these?

Linear rootfinding

How do we solve these?

Consider a simple generic problem:

$Ax = b$

Linear rootfinding

How do we solve these?

Consider a simple generic problem:

$Ax = b$

Invert A

$x = A^{-1}b$

Linear rootfinding

How do we solve these?

Consider a simple generic problem:

$Ax = b$

Invert A

$x = A^{-1}b$

THE END

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$$f(x) = 0, f : \mathbb{R} \to \mathbb{R}^n$$

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$f(x) = 0, f : \mathbb{R} \to \mathbb{R}^n$

What's a common rootfinding problem?

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$f(x) = 0, f : \mathbb{R} \rightarrow \mathbb{R}^n$

What's a common rootfinding problem?

Can we reframe a common economic problem as rootfinding?

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$$f(x) = 0, f : \mathbb{R} \to \mathbb{R}^n$$

What's a common rootfinding problem?

Can we reframe a common economic problem as rootfinding?

Yes!

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$f(x) = 0, f : \mathbb{R} \to \mathbb{R}^n$

What's a common rootfinding problem?

Can we reframe a common economic problem as rootfinding?

Yes!

Fixed point problems are rootfinding problems:

Non-linear rootfinding

With non-linear rootfinding problems we want to solve:

$$f(x) = 0, f : \mathbb{R} \to \mathbb{R}^n$$

What's a common rootfinding problem?

Can we reframe a common economic problem as rootfinding?

Yes!

Fixed point problems are rootfinding problems:

$$g(x) = x \Rightarrow f(x) \equiv g(x) - x = 0$$

Basic non-linear rootfinders: Bisection method

What does the intermediate value theorem tell us?

Basic non-linear rootfinders: Bisection method

What does the intermediate value theorem tell us?

If a continuous real-valued function on a given interval takes on two values a and b, it achieves all values in the set $[a, b]$ somewhere in its domain

Basic non-linear rootfinders: Bisection method

What does the intermediate value theorem tell us?

If a continuous real-valued function on a given interval takes on two values a and b, it achieves all values in the set $[a, b]$ somewhere in its domain

How can this motivate an algorithm to find the root of a function?

Basic non-linear rootfinders: Bisection method

If we have a continuous, 1 variable function that is positive at some value and negative at another, a root must fall in between those values

Basic non-linear rootfinders: Bisection method

If we have a continuous, 1 variable function that is positive at some value and negative at another, a root must fall in between those values

We know a root exists by IVT, what's an efficient way to find it?

Basic non-linear rootfinders: Bisection method

If we have a continuous, 1 variable function that is positive at some value and negative at another, a root must fall in between those values

We know a root exists by IVT, what's an efficient way to find it?

Continually bisect the interval!

The bisection method

The bisection method works by continually bisecting the interval and only keeping the half interval with a zero until "convergence"

1. Select the midpoint of $[a, b]$, $(a + b)/2$

The bisection method

The bisection method works by continually bisecting the interval and only keeping the half interval with a zero until "convergence"

1. Select the midpoint of $[a, b]$, $(a + b)/2$
2. Zero must be in the lower or upper half

The bisection method

The bisection method works by continually bisecting the interval and only keeping the half interval with a zero until "convergence"

1. Select the midpoint of $[a, b]$, $(a + b)/2$
2. Zero must be in the lower or upper half
3. Check the sign of the midpoint, if it has the same sign as the lower bound a root must be the right subinterval

The bisection method

The bisection method works by continually bisecting the interval and only keeping the half interval with a zero until "convergence"

1. Select the midpoint of $[a, b]$, $(a + b)/2$
2. Zero must be in the lower or upper half
3. Check the sign of the midpoint, if it has the same sign as the lower bound a root must be the right subinterval
4. Select the midpoint of $[(a + b)/2, b]$...

Write out the code to do it

The bisection algorithm

```julia
function bisection(f, lower_bound, upper_bound)

    tolerance = 1e-3                                # tolerance for solution
    guess = 0.5*(upper_bound + lower_bound)         # initial guess, bisect the interval
    difference = (upper_bound - lower_bound)/2      # initialize bound difference

    while difference > tolerance                    # loop until convergence
        println("Intermediate guess of $guess.")
        difference = difference/2
        if sign(f(lower_bound)) == sign(f(guess))   # if the guess has the same sign as the lowe
            lower_bound = guess                     # solution is in the upper half of the inte
            guess = guess + difference
        else                                        # else the solution is in the lower half of
            upper_bound = guess
            guess = guess - difference
        end

    end
    println("The root of f(x) is at $guess.")
end
```

The bisection method

```
f(x) = x^3;
bisection(f, -4, 1)
```

```
## Intermediate guess of -1.5.
## Intermediate guess of -0.25.
## Intermediate guess of 0.375.
## Intermediate guess of 0.0625.
## Intermediate guess of -0.09375.
## Intermediate guess of -0.015625.
## Intermediate guess of 0.0234375.
## Intermediate guess of 0.00390625.
## Intermediate guess of -0.005859375.
## Intermediate guess of -0.0009765625.
## Intermediate guess of 0.00146484375.
## Intermediate guess of 0.000244140625.
## The root of f(x) is at -0.0003662109375.
```

The bisection method

```
g(x) = 3x^3 + 2x -4;
bisection(g, -6, 4)
```

```
## Intermediate guess of -1.0.
## Intermediate guess of 1.5.
## Intermediate guess of 0.25.
## Intermediate guess of 0.875.
## Intermediate guess of 1.1875.
## Intermediate guess of 1.03125.
## Intermediate guess of 0.953125.
## Intermediate guess of 0.9140625.
## Intermediate guess of 0.89453125.
## Intermediate guess of 0.904296875.
## Intermediate guess of 0.8994140625.
## Intermediate guess of 0.90185546875.
## Intermediate guess of 0.900634765625.
## The root of f(x) is at 0.9012451171875.
```

The bisection method

```
h(x) = cos(x);
bisection(h, -pi, pi)
```

```
## Intermediate guess of 0.0.
## Intermediate guess of -1.5707963267948966.
## Intermediate guess of -2.356194490192345.
## Intermediate guess of -1.9634954084936207.
## Intermediate guess of -1.7671458676442586.
## Intermediate guess of -1.6689710972195777.
## Intermediate guess of -1.6198837120072371.
## Intermediate guess of -1.595340019401067.
## Intermediate guess of -1.5830681730979819.
## Intermediate guess of -1.5769322499464393.
## Intermediate guess of -1.573864288370668.
## Intermediate guess of -1.5723303075827824.
## The root of f(x) is at -1.5715633171888395.
```

The bisection method

The bisection method is incredibly robust: if a function f satisfies the IVT, it is **guaranteed to converge in a specific number of iterations**

The bisection method

The bisection method is incredibly robust: if a function f satisfies the IVT, it is **guaranteed to converge in a specific number of iterations**

A root can be calculated to arbitrary precision ϵ
in a maximum of $log([b-a]/\epsilon)/log(2)$ iterations

Robustness comes with drawbacks:

The bisection method

The bisection method is incredibly robust: if a function f satisfies the IVT, it is **guaranteed to converge in a specific number of iterations**

A root can be calculated to arbitrary precision ϵ
in a maximum of $log([b-a]/\epsilon)/log(2)$ iterations

Robustness comes with drawbacks:

1. It only works in one dimension
2. It is slow because it only uses information about the function's level

Function iteration

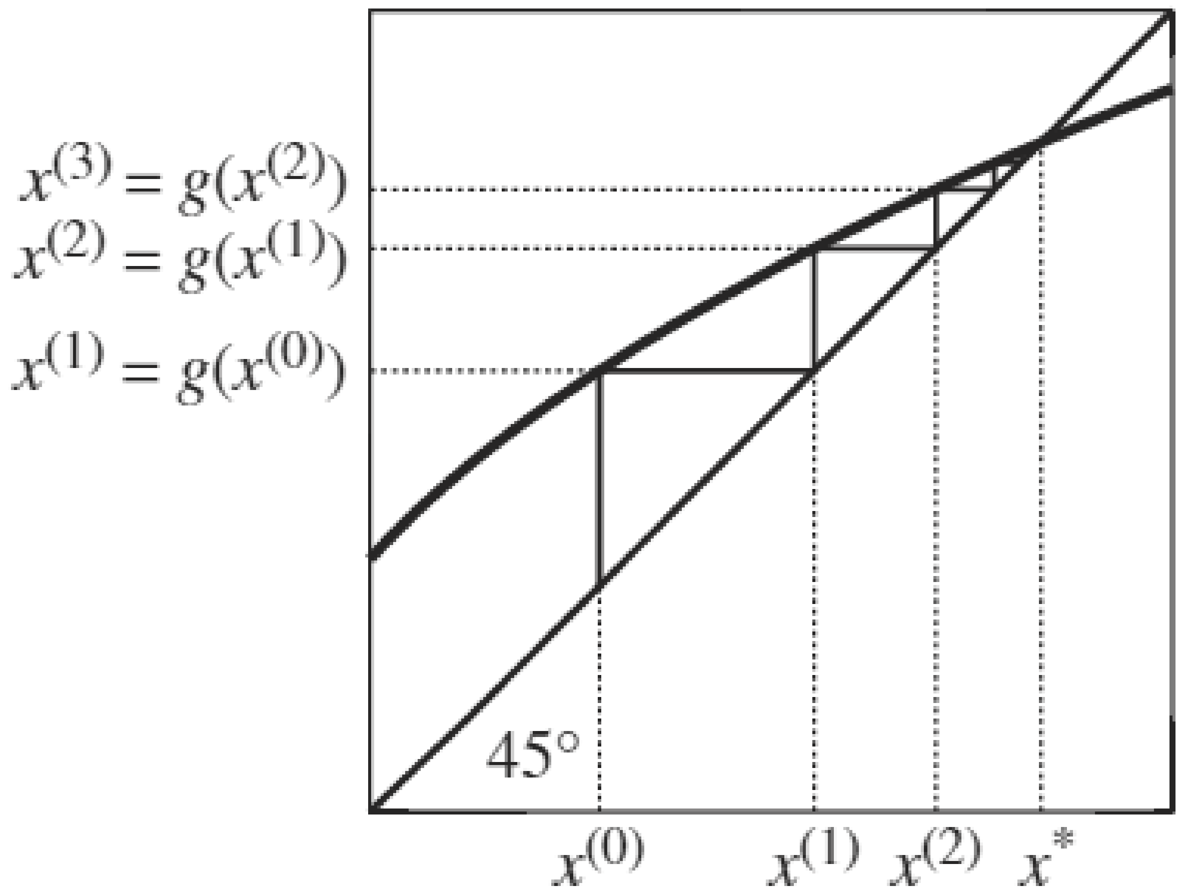
Fixed points can be computed using function iteration

Function iteration

Fixed points can be computed using function iteration

Since we can recast fixed points as rootfinding problems we can use function iteration to find roots too

Function iteration



Function iteration

Function iteration can be quick, but is not always guaranteed to converge

Function iteration

Function iteration can be quick, but is not always guaranteed to converge

In general, it can be quite unstable as we will see

Function iteration

Function iteration can be quick, but is not always guaranteed to converge

In general, it can be quite unstable as we will see

Code up a function iteration algorithm to find a fixed point of an arbitrary function f

Function iteration

Function iteration is pretty simple to implement

```
function function_iteration(f, guess)
    tolerance = 1e-2                                     # tolerance for solution
    max_it = 10                                          # maximum number of iterations
    x_old = guess                                        # initialize old x value
    x = guess                                            # initialize current x
    error = 1e10                                         # initialize error
    it = 1

    while abs(error) > tolerance && it < max_it
        println("Intermediate guess of $x.")
        x = f(x_old)                                     # new x = f(old x)
        error = x - x_old                                # error
        x_old = x
        it = it + 1
    end
    println("The fixed point of f(x) is at $x.")
end;
```

Function iteration

Analytic solution: 1

```
f(x) = x^(-0.5);
function_iteration(f, 2.)
```

```
## Intermediate guess of 2.0.
## Intermediate guess of 0.7071067811865476.
## Intermediate guess of 1.189207115002721.
## Intermediate guess of 0.9170040432046712.
## Intermediate guess of 1.0442737824274138.
## Intermediate guess of 0.9785720620877002.
## Intermediate guess of 1.0108892860517005.
## Intermediate guess of 0.9945994234836332.
## The fixed point of f(x) is at 1.0027112750502025.
```

Works!

Function iteration

Analytic solution: $\sqrt{3} \approx 1.73$

```
f(x) = 3 + x - x^2;
function_iteration(f, 2.)
```

```
## Intermediate guess of 2.0.
## Intermediate guess of 1.0.
## Intermediate guess of 3.0.
## Intermediate guess of -3.0.
## Intermediate guess of -9.0.
## Intermediate guess of -87.0.
## Intermediate guess of -7653.0.
## Intermediate guess of -5.8576059e7.
## Intermediate guess of -3.431154746547537e15.
## The fixed point of f(x) is at -1.1772822894755698e31.
```

=(

Function iteration

Analytic solution: 1.5

```
f(x) = 3 - x;
function_iteration(f, 2.)
```

```
## Intermediate guess of 2.0.
## Intermediate guess of 1.0.
## Intermediate guess of 2.0.
## Intermediate guess of 1.0.
## Intermediate guess of 2.0.
## Intermediate guess of 1.0.
## Intermediate guess of 2.0.
## Intermediate guess of 1.0.
## Intermediate guess of 2.0.
## The fixed point of f(x) is at 1.0.
```

=(

Function iteration

Analytic solution: 1 or 0

```
f(x) = x^2;
function_iteration(f, 1.01)
```

```
## Intermediate guess of 1.01.
## Intermediate guess of 1.0201.
## Intermediate guess of 1.04060401.
## Intermediate guess of 1.0828567056280802.
## Intermediate guess of 1.1725786449236988.
## Intermediate guess of 1.3749406785310976.
## Intermediate guess of 1.890461869479555.
## Intermediate guess of 3.573846079956134.
## Intermediate guess of 12.772375803217825.
## The fixed point of f(x) is at 163.1335836586242.
```

=(

Function iteration

Is function iteration fundamentally flawed?

Function iteration

Is function iteration fundamentally flawed? Not quite

Function iteration

Is function iteration fundamentally flawed? Not quite

Some of these issues can be solved by **damping**

Function iteration

Is function iteration fundamentally flawed? Not quite

Some of these issues can be solved by **damping**

Damping is where you do not do a full update of x, but a convex combination of the new value $f(x)$ and the old value x: $x_{new} = \alpha f(x_{old}) + (1 - \alpha)x_{old}$

Function iteration

Is function iteration fundamentally flawed? Not quite

Some of these issues can be solved by **damping**

Damping is where you do not do a full update of x, but a convex combination of the new value $f(x)$ and the old value x: $x_{new} = \alpha f(x_{old}) + (1 - \alpha)x_{old}$

Damping improves the stability of iterative algorithms

Function iteration

Is function iteration fundamentally flawed? Not quite

Some of these issues can be solved by **damping**

Damping is where you do not do a full update of x, but a convex combination of the new value $f(x)$ and the old value x: $x_{new} = \alpha f(x_{old}) + (1-\alpha)x_{old}$

Damping improves the stability of iterative algorithms

Rewrite your algorithm with damping and try again

Function iteration

Is function iteration fundamentally flawed? Not quite

Some of these issues can be solved by **damping**

Damping is where you do not do a full update of x, but a convex combination of the new value $f(x)$ and the old value x: $x_{new} = \alpha f(x_{old}) + (1 - \alpha)x_{old}$

Damping improves the stability of iterative algorithms

Rewrite your algorithm with damping and try again

For some α, you need to decrease your tolerance by a factor of $1/\alpha$ to account for how the damped error will be smaller by the same factor

Function iteration

Function iteration is pretty simple to implement

```
function function_iteration_damped(f, guess)
    tolerance = 1e-4                                        # tolerance for solution
    max_it = 1000                                           # maximum number of iterations
    x_old = guess                                           # initialize old x value
    x = guess                                               # initialize current x
    error = 1e10                                            # initialize error
    it = 1

    while abs(error) > tolerance && it < max_it
        x = 0.1 * f(x_old) + 0.9 * x_old
        error = x - x_old                                   # error
        x_old = x
        it = it + 1
    end
    println("The fixed point of f(x) is at $x.")
end;
```

Function iteration

Analytic solution: 1

```
f(x) = x^(-0.5);
function_iteration_damped(f, 2.)
```

```
## The fixed point of f(x) is at 1.0005141871702672.
```

Works!

Function iteration

Analytic solution: $\sqrt{3} \approx 1.73$

```
f(x) = 3 + x - x^2;
function_iteration_damped(f, 2.)
```

```
## The fixed point of f(x) is at 1.7322240086832341.
```

Works!

Function iteration

Analytic solution: 1.5

```
f(x) = 3 - x;
function_iteration_damped(f, 2.)
```

```
## The fixed point of f(x) is at 1.5003961408125717.
```

Works!

Function iteration

Analytic solution: 1 or 0

```
f(x) = x^2;
function_iteration_damped(f, 1.01)
```

```
## The fixed point of f(x) is at Inf.
```

=(

Function iteration does struggle with some functions even with damping

Newton's method

Newton's method and variants are the workhorses of solving n-dimensional non-linear problems

Newton's method

Newton's method and variants are the workhorses of solving n-dimensional non-linear problems

What's the idea?

Newton's method

Newton's method and variants are the workhorses of solving n-dimensional non-linear problems

What's the idea?

Take a hard non-linear problem and replace it with a sequence of linear problems

Newton's method

Newton's method and variants are the workhorses of solving n-dimensional non-linear problems

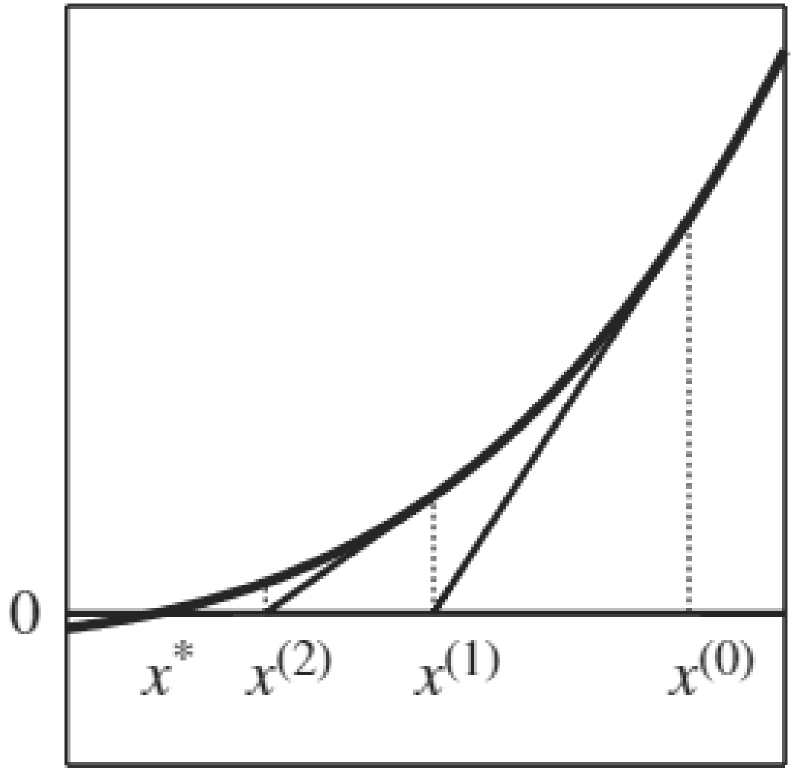
What's the idea?

Take a hard non-linear problem and replace it with a sequence of linear problems

Under certain conditions the sequence of solutions will converge to the true solution

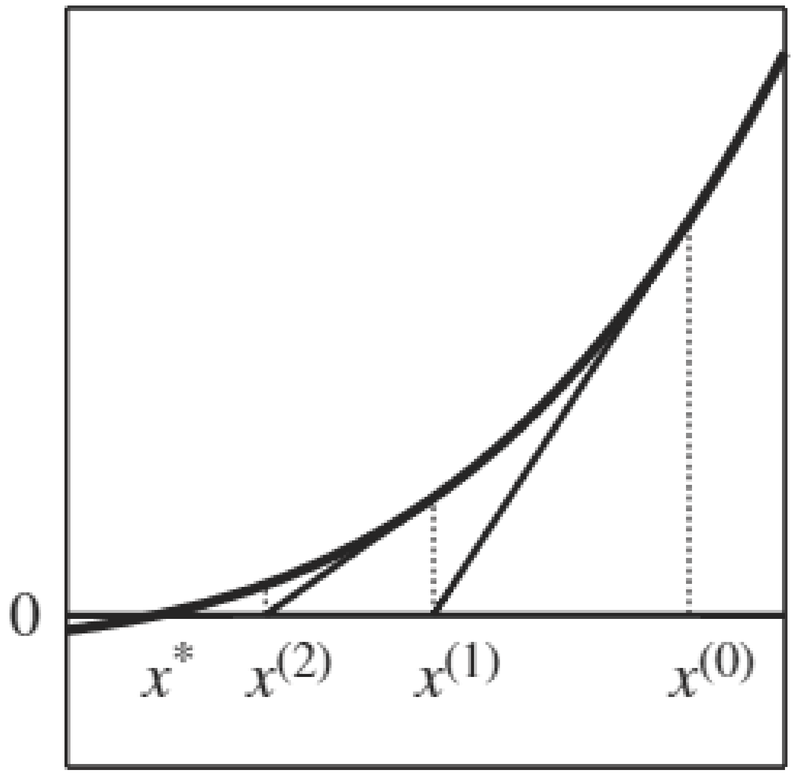
Newton's method

Here's a graphical depiction of Newton's method:



Newton's method

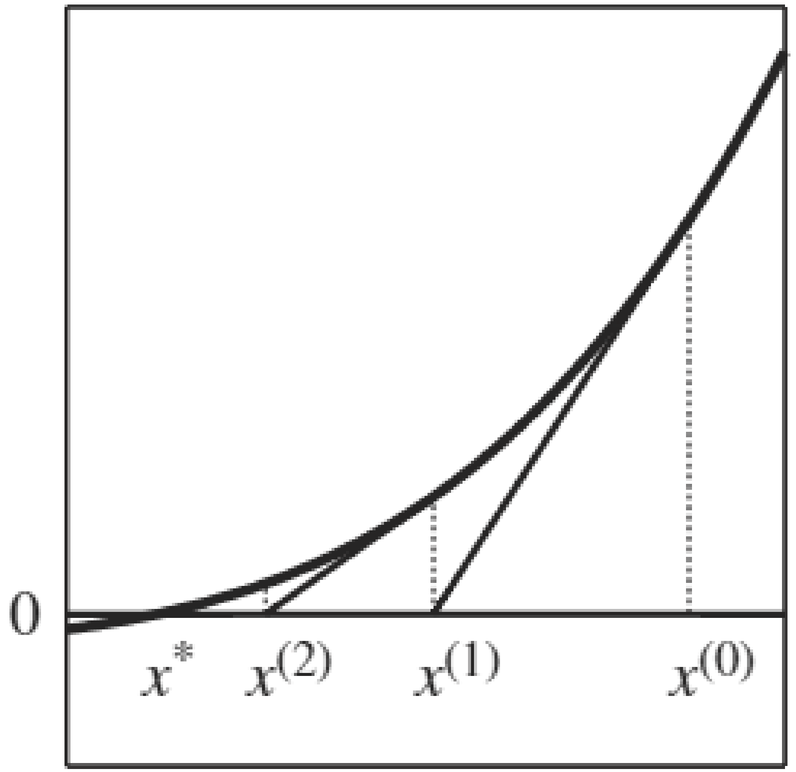
Start with an initial guess of the root at $x^{(0)}$



Newton's method

Start with an initial guess of the root at $x^{(0)}$

Approximate the non-linear function with its first-order Taylor expansion about $x^{(0)}$

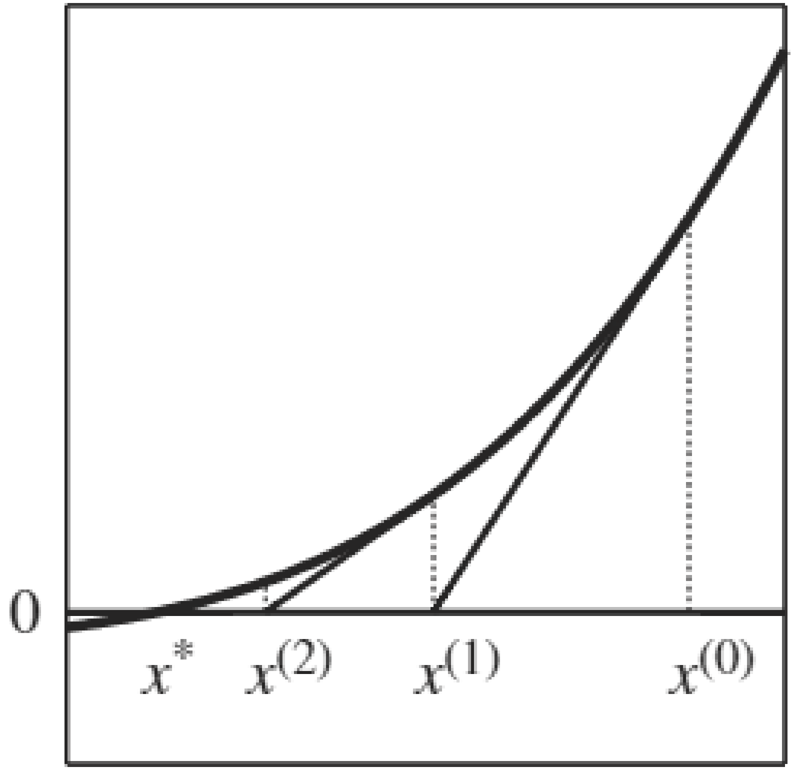


Newton's method

Start with an initial guess of the root at $x^{(0)}$

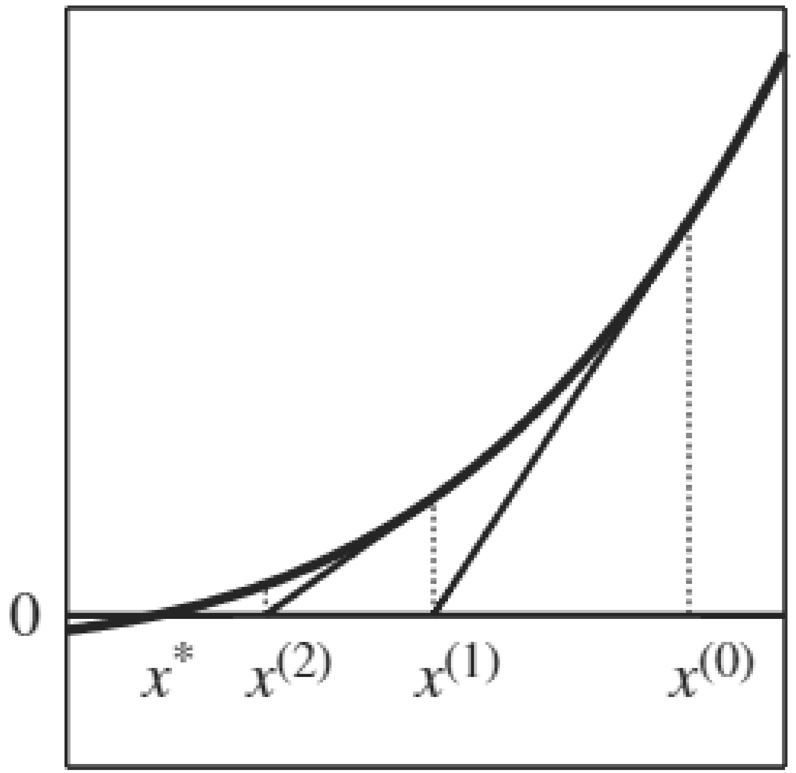
Approximate the non-linear function with its first-order Taylor expansion about $x^{(0)}$

This is just the tangent line at x^0, solve for the root of this linear approximation, call it $x^{(1)}$



Newton's method

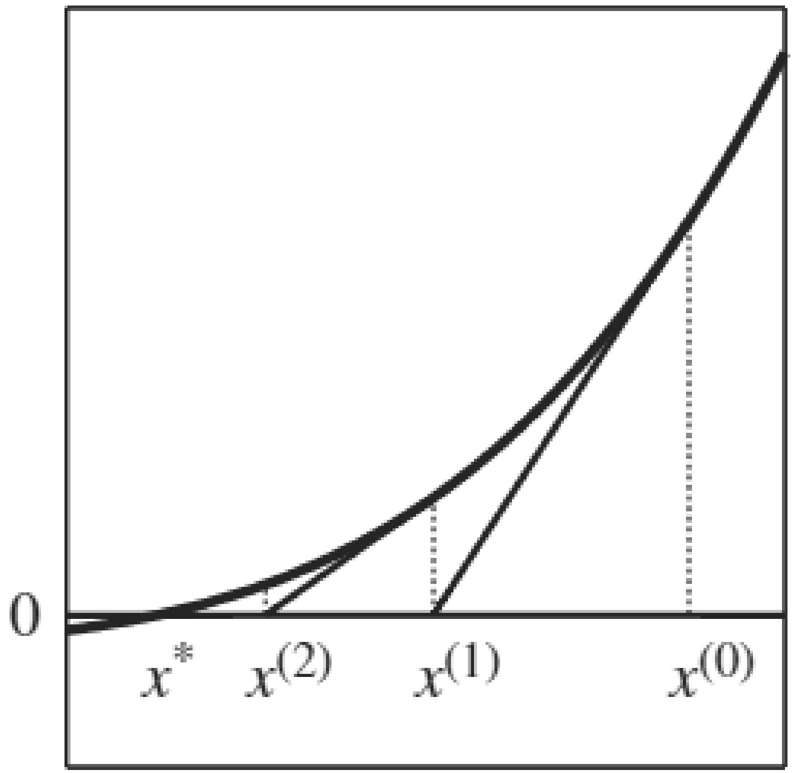
Repeat starting at $x^{(1)}$ until we converge to x^*



Newton's method

Repeat starting at $x^{(1)}$ until we converge to x^*

This can be applied to a function with an arbitrary number of dimensions



Newton's method

Begin with some initial guess of the root vector $\mathbf{x}^{(\mathbf{0})}$

Newton's method

Begin with some initial guess of the root vector $\mathbf{x^{(0)}}$

Our new guess $\mathbf{x^{(k+1)}}$ given some arbitrary point in the algorithm, $\mathbf{x^{(k)}}$, is obtained by approximating $f(\mathbf{x})$ using a first-order Taylor expansion about $\mathbf{x^{(k)}}$ and solving for $\mathbf{x}$:

$$f(\mathbf{x}) \approx f(\mathbf{x^{(k)}}) + f'(\mathbf{x^{(k)}})(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}) = 0$$
$$\Rightarrow \mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} - \left[f'(\mathbf{x^{(k)}})\right]^{-1} f(\mathbf{x^{(k)}})$$

Newton's method

Code up a one variable Newton's method algorithm for an arbitrary function f

Newton's method

Code up a one variable Newton's method algorithm for an arbitrary function f

```
function newtons_method(f, f_prime, guess)
    diff = Inf     # Initialize problem
    tol = 1e-5
    x_old = guess
    x = 1e10

    while abs(diff) > tol
        x = f(x_old) - f(x_old)/f_prime(x_old) # Root of linear approximation
        diff = x - x_old
        x_old = x
    end
    println("The root of f(x) is at $x.")
end;
```

Newton's method

```julia
f(x) = x^3;
f_prime(x) = 3x^2;
newtons_method(f, f_prime, 1.)
```

```
## The root of f(x) is at 1.231347218094855e-6.
```

Newton's method

```
f(x) = x^3;
f_prime(x) = 3x^2;
newtons_method(f, f_prime, 1.)
```

```
## The root of f(x) is at 1.231347218094855e-6.
```

```
f(x) = sin(x);
f_prime(x) = cos(x);
newtons_method(f, f_prime, pi/4)
```

```
## The root of f(x) is at 5.941936124988917e-19.
```

Newton's method

Newton's method has nice properties regarding convergence and speed:

If $f(x)$ is continuously differentiable, the initial guess is "sufficiently close" to the root, and $f(x)$ is invertible near the root, then Newton's method converges to the root

Newton's method

Newton's method has nice properties regarding convergence and speed:

If $f(x)$ is continuously differentiable, the initial guess is "sufficiently close" to the root, and $f(x)$ is invertible near the root, then Newton's method converges to the root

What is "sufficiently close"?

Newton's method

Newton's method has nice properties regarding convergence and speed:

If $f(x)$ is continuously differentiable, the initial guess is "sufficiently close" to the root, and $f(x)$ is invertible near the root, then Newton's method converges to the root

What is "sufficiently close"?

We need $f(x)$ to be invertible so the algorithm above is well defined

Newton's method

Newton's method has nice properties regarding convergence and speed:

If $f(x)$ is continuously differentiable, the initial guess is "sufficiently close" to the root, and $f(x)$ is invertible near the root, then Newton's method converges to the root

What is "sufficiently close"?

We need $f(x)$ to be invertible so the algorithm above is well defined

If $f'(x)$ is ill-conditioned we can run into problems with rounding error

Quasi-Newton: Secant method

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Quasi-Newton: Secant method

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Why?

Quasi-Newton: Secant method

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Why?

1. Coding error / time
2. Can actually be slower to evaluate than finite differences for a nonlinear problem, see Ken Judd's notes

Quasi-Newton: Secant method

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Why?

1. Coding error / time
2. Can actually be slower to evaluate than finite differences for a nonlinear problem, see Ken Judd's notes

Quasi-Newton: Secant method

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Why?

1. Coding error / time
2. Can actually be slower to evaluate than finite differences for a nonlinear problem, see Ken Judd's notes

Using our current root guess $x^{(k)}$ and our previous root guess $x^{(k-1)}$:

$$f'(x^{(k)}) \approx \frac{f(x^{(k)})-f(x^{(k-1)})}{x^{(k)}-x^{(k-1)}}$$

Quasi-Newton: Secant method

Our new iteration rule then becomes

Quasi-Newton: Secant method

Our new iteration rule then becomes

$$x^{(k+1)} = x^{(k)} - \tfrac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})} f(x^{(k)})$$

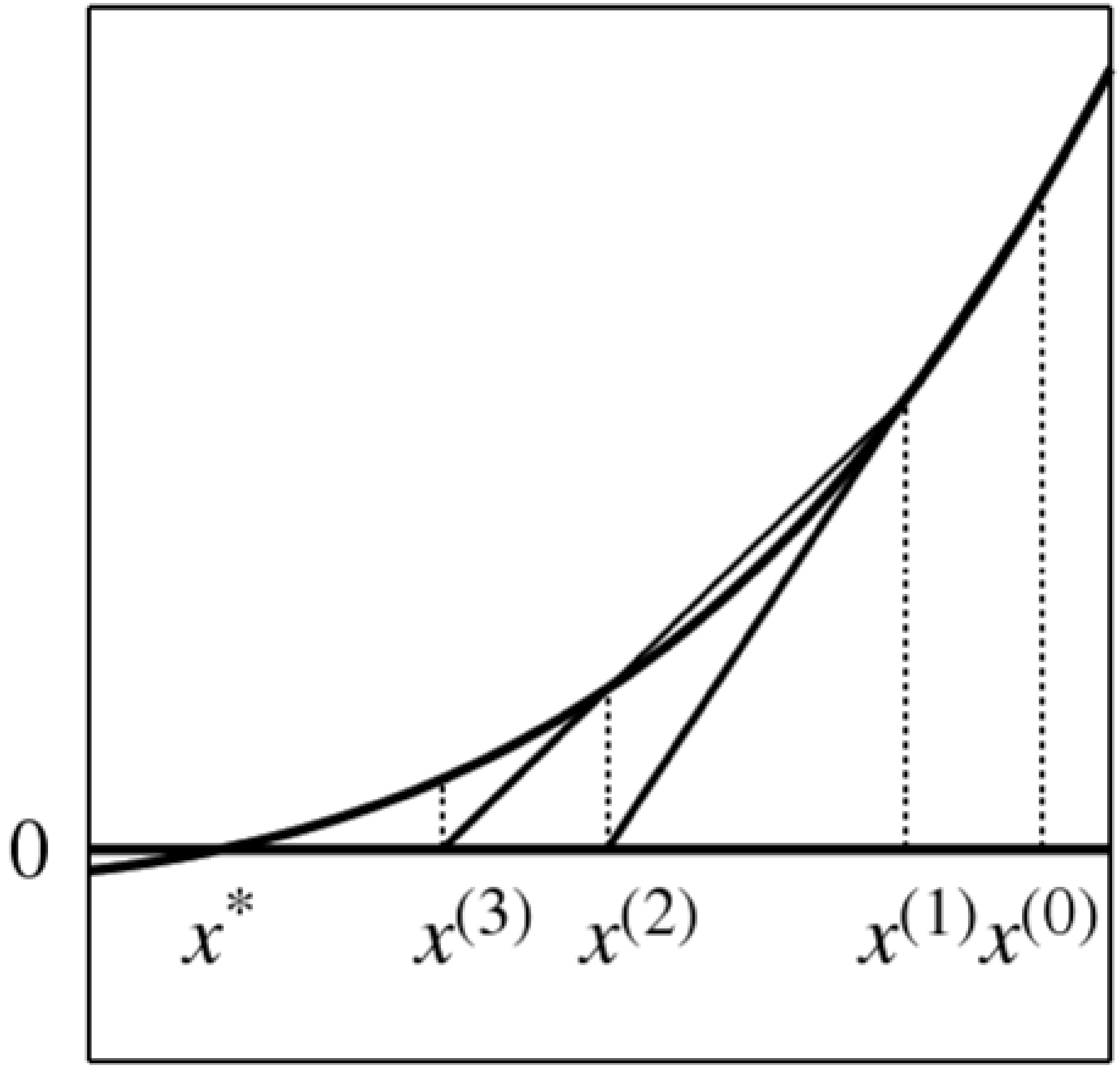
Quasi-Newton: Secant method

Our new iteration rule then becomes

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})} f(x^{(k)})$$

Now we require two initial guesses so that we have an initial approximation of the derivative

Quasi-Newton: Secant method



Quasi-Newton: Broyden's method

Broyden's method is the most widely used rootfinding method for n-dimensional problems

Quasi-Newton: Broyden's method

Broyden's method is the most widely used rootfinding method for n-dimensional problems

It is a generalization of the secant method where have a sequence of guesses of the Jacobian at the root

Quasi-Newton: Broyden's method

Broyden's method is the most widely used rootfinding method for n-dimensional problems

It is a generalization of the secant method where have a sequence of guesses of the Jacobian at the root

We must initially provide a guess of the root, $x^{(0)}$, but also a guess of the Jacobian, $A_{(0)}$

Quasi-Newton: Broyden's method

Root guess update is the same as before but with our guess of the Jacobian substituted in for the actual Jacobian or the finite difference approximation

$$\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} - A_{(k)}^{-1}\, f(\mathbf{x^{(k)}}).$$

Quasi-Newton: Broyden's method

Root guess update is the same as before but with our guess of the Jacobian substituted in for the actual Jacobian or the finite difference approximation

$$\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} - A_{(k)}^{-1} f(\mathbf{x^{(k)}}).$$

we still need to update $A_{(k)}$: we do this update is performed by making the smallest change, in terms of the Frobenius matrix norm, that satisfies what is called the *secant condition* (under determined if $n > 1$):

$$f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}}) = A_{(k+1)} \left(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}\right)$$

Quasi-Newton: Broyden's method

The updated differences in root guesses, and the function value at those root guesses, should align with our estimate of the Jacobian at that point

Quasi-Newton: Broyden's method

The updated differences in root guesses, and the function value at those root guesses, should align with our estimate of the Jacobian at that point

$$A_{(k+1)} = A_{(k)} +$$

$$\left[f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}}) - A_{(k+1)}\left(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}\right)\right] \times$$

$$\frac{\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}}{(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}})^T(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}})}$$

Accelerating Broyden

Why update the Jacobian and then invert when we can just update an inverted Jacobian $B = A^{-1}$

$$B_{(k+1)} = B_{(k)} + \frac{[d^{(k)} - u^{(k)}]{d^{(k)}}^T B_{(k)}}{{d^{(k)}}^T u^{(k)}}$$

where $d^{(k)} = (\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}})$, and $u^{(k)} = B_{(k)}\left[f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}})\right]$.

Accelerating Broyden

Broyden converges under relatively weak conditions:

Accelerating Broyden

Broyden converges under relatively weak conditions:

1. f is continuously differentiable,
2. $x^{(0)}$ is close to the root of f
3. f' is invertible around the root
4. A_0 is sufficiently close to the Jacobian

Convergence speed

Rootfinding algorithms will converge at different speeds in terms of the number of operations

Convergence speed

Rootfinding algorithms will converge at different speeds in terms of the number of operations

A sequence of iterates $x^{(k)}$ is said to converge to x^* at a rate of order p if there is a constant C such that

$$||x^{(k+1)} - x^*|| \leq C||x^{(k)} - x^*||^p$$

for sufficiently large k

Convergence speed

$$||x^{(k+1)} - x^*|| \leq C||x^{(k)} - x^*||^p$$

If $C < 1$ and $p = 1$, the rate of convergence is linear

If $1 < p < 2$, convergence is superlinear, and if $p = 2$ convergence is quadratic.
The higher order the convergence rate, the faster it converges

Convergence speed

How fast do the methods we've seen converge?

Convergence speed

How fast do the methods we've seen converge?

- Bisection: linear rate with $C = 0.5$ (kind of obvious once you see it)

Convergence speed

How fast do the methods we've seen converge?

- Bisection: linear rate with $C = 0.5$ (kind of obvious once you see it)
- Function iteration: linear rate with $C = ||f'(x^*)||$

Convergence speed

How fast do the methods we've seen converge?

- Bisection: linear rate with $C = 0.5$ (kind of obvious once you see it)
- Function iteration: linear rate with $C = ||f'(x^*)||$
- Secant and Broyden: superlinear rate with $p \approx 1.62$

Convergence speed

How fast do the methods we've seen converge?

- Bisection: linear rate with $C = 0.5$ (kind of obvious once you see it)
- Function iteration: linear rate with $C = ||f'(x^*)||$
- Secant and Broyden: superlinear rate with $p \approx 1.62$
- Newton: $p = 2$

Convergence speed

Convergence rates only account for the number of **iterations** of the method

The steps taken in a given iteration of each solution method may vary in computational cost because of differences in the number of arithmetic operations

Although an algorithm may take more iterations to solve,
each iteration may be solved faster and the overall algorithm takes less time

Convergence speed

Ex:

- Bisection method only requires a single function evaluation during each iteration
- Function iteration only requires a single function evaluation during each iteration
- Broyden's method requires both a function evaluation and matrix multiplication
- Newton's method requires a function evaluation, a derivative evaluation, and solving a linear system

Convergence speed

Ex:

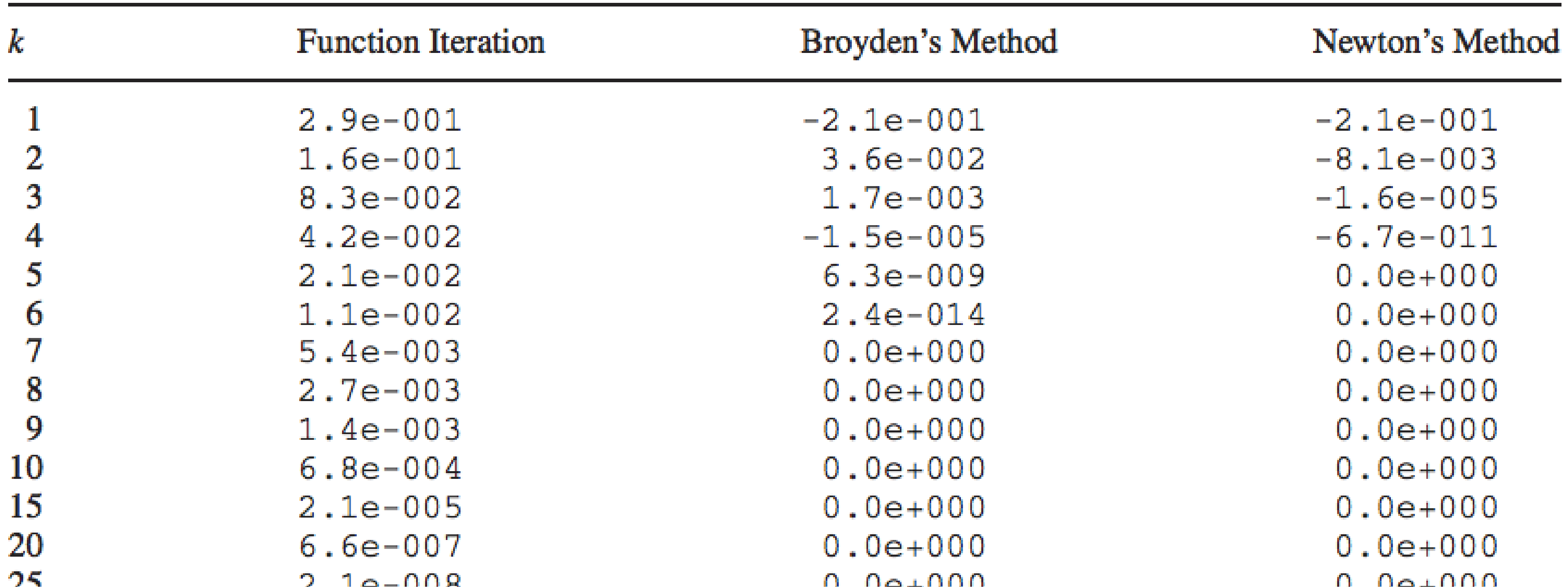
- Bisection method only requires a single function evaluation during each iteration
- Function iteration only requires a single function evaluation during each iteration
- Broyden's method requires both a function evaluation and matrix multiplication
- Newton's method requires a function evaluation, a derivative evaluation, and solving a linear system

Bisection and function iteration are usually slow

Convergence speed

Consider an example where $f(x) = x - \sqrt{}(x) = 0$

What does convergence look like across our main approaches in terms of the L^1-norm if all guesses start at $x^{(0)} = 0.5$?



k	Function Iteration	Broyden's Method	Newton's Method
1	2.9e-001	-2.1e-001	-2.1e-001
2	1.6e-001	3.6e-002	-8.1e-003
3	8.3e-002	1.7e-003	-1.6e-005
4	4.2e-002	-1.5e-005	-6.7e-011
5	2.1e-002	6.3e-009	0.0e+000
6	1.1e-002	2.4e-014	0.0e+000
7	5.4e-003	0.0e+000	0.0e+000
8	2.7e-003	0.0e+000	0.0e+000
9	1.4e-003	0.0e+000	0.0e+000
10	6.8e-004	0.0e+000	0.0e+000
15	2.1e-005	0.0e+000	0.0e+000
20	6.6e-007	0.0e+000	0.0e+000
25	2.1e-008	0.0e+000	0.0e+000

Maximization (minimization) methods

How we solve maximization problems has many similarities to rootfinding and complementarity problems

Maximization (minimization) methods

How we solve maximization problems has many similarities to rootfinding and complementarity problems

I'll tend to frame problems as minimization problems because it is the convention the optimization literature

Maximization (minimization) methods

How we solve maximization problems has many similarities to rootfinding and complementarity problems

I'll tend to frame problems as minimization problems because it is the convention the optimization literature

We make two distinctions:

Maximization (minimization) methods

How we solve maximization problems has many similarities to rootfinding and complementarity problems

I'll tend to frame problems as minimization problems because it is the convention the optimization literature

We make two distinctions:

Local vs global: are we finding an optimum in a local region, or globally?

Maximization (minimization) methods

How we solve maximization problems has many similarities to rootfinding and complementarity problems

I'll tend to frame problems as minimization problems because it is the convention the optimization literature

We make two distinctions:

Local vs global: are we finding an optimum in a local region, or globally?

Derivative-using vs derivative-free: Do we want to use higher-order information?

Maximization (minimization) methods

I'll focus on local solvers, common global solvers I won't cover:

1. Genetic algorithms
2. Simulated annealing
3. DIRECT

Derivative free optimization: Golden search

Similar to bisection, **golden search** looks for a solution of a one-dimensional problem over smaller and smaller brackets

Derivative free optimization: Golden search

Similar to bisection, **golden search** looks for a solution of a one-dimensional problem over smaller and smaller brackets

If we have a continuous one dimensional function, $f(x)$, and we want to find a local minimum in some interval $[a, b]$

Derivative free optimization: Golden search

1. Select points $x_1, x_2 \in [a, b]$ where $x_2 > x_1$

Derivative free optimization: Golden search

1. Select points $x_1, x_2 \in [a, b]$ where $x_2 > x_1$

2. If $f(x_1) < f(x_2)$ replace $[a, b]$ with $[a, x_2]$, else replace $[a, b]$ with $[x_1, b]$

Derivative free optimization: Golden search

1. Select points $x_1, x_2 \in [a, b]$ where $x_2 > x_1$
2. If $f(x_1) < f(x_2)$ replace $[a, b]$ with $[a, x_2]$, else replace $[a, b]$ with $[x_1, b]$
3. Repeat until convergence criterion is met

Derivative free optimization: Golden search

1. Select points $x_1, x_2 \in [a, b]$ where $x_2 > x_1$
2. If $f(x_1) < f(x_2)$ replace $[a, b]$ with $[a, x_2]$, else replace $[a, b]$ with $[x_1, b]$
3. Repeat until convergence criterion is met

Replace the endpoint of the interval next to the evaluated point with the highest value $\rightarrow$ keep the lower evaluated point in the interval $\rightarrow$ guarantees that a local minimum still exists

Derivative free optimization: Golden search

How do we pick x_1 and x_2?

Derivative free optimization: Golden search

How do we pick x_1 and x_2?

Achievable goal for selection process:

- New interval is independent of whether the upper or lower bound is replaced
- Only requires one function evaluation per iteration

Derivative free optimization: Golden search

How do we pick x_1 and x_2?

Achievable goal for selection process:

- New interval is independent of whether the upper or lower bound is replaced
- Only requires one function evaluation per iteration

There's one algorithm that satisfies this

Derivative free optimization: Golden search

Golden search algorithm for point selection:

$$x_i = a + \alpha_i(b - a)$$

$$\alpha_1 = \frac{3 - \sqrt{5}}{2} \qquad \alpha_2 = \frac{\sqrt{5} - 1}{2}$$

Derivative free optimization: Golden search

Golden search algorithm for point selection:

$$x_i = a + \alpha_i(b - a)$$
$$\alpha_1 = \frac{3 - \sqrt{5}}{2} \qquad \alpha_2 = \frac{\sqrt{5} - 1}{2}$$

The value of α_2 is called the golden ratio and is where the algorithm gets its name

Derivative free optimization: Golden search

Golden search algorithm for point selection:

$$x_i = a + \alpha_i(b - a)$$

$$\alpha_1 = \frac{3 - \sqrt{5}}{2} \qquad \alpha_2 = \frac{\sqrt{5} - 1}{2}$$

The value of α_2 is called the golden ratio and is where the algorithm gets its name

Write out a golden search algorithm

Golden search

```julia
function golden_search(f, lower_bound, upper_bound)
    alpha_1 = (3 - sqrt(5))/2  # GS parameter 1
    alpha_2 = (sqrt(5) - 1)/2  # GS parameter 2
    tolerance = 1e-2           # tolerance for convergence
    difference = 1e10
    while difference > tolerance
        x_1 = lower_bound + alpha_1*(upper_bound - lower_bound)  # new x_1
        x_2 = lower_bound + alpha_2*(upper_bound - lower_bound)  # new x_2
        if f(x_1) < f(x_2)     # reset bounds
            upper_bound = x_2
        else
            lower_bound = x_1
        end
        difference = x_2 - x_1
    end
    println("Minimum is at x = $((lower_bound+upper_bound)/2).")
end;
```

Golden search

```
f(x) = 2x^2 + 9x;
golden_search(f, -10, 10)
```

```
## Minimum is at x = -2.2483173872886444.
```

```
f(x) = x^4;
golden_search(f, -5, 3)
```

```
## Minimum is at x = -0.003105620015141938.
```

```
f(x) = sin(x);
golden_search(f, 0, 1)
```

```
## Minimum is at x = 0.010643118126104103.
```

Nelder-Mead: Simplex

Golden search is nice and simple but only works in one dimension

There are several derivative free methods for minimization that work in multiple dimensions, the most commonly used one is **Nelder-Mead (NM)**

Nelder-Mead: Simplex

Golden search is nice and simple but only works in one dimension

There are several derivative free methods for minimization that work in multiple dimensions, the most commonly used one is **Nelder-Mead (NM)**

NM works by first constructing a simplex: we evaluate the function at $n+1$ points in an n dimensional problem

It then manipulates the highest value point, similar to golden search

Nelder-Mead: Simplex

There are six operations:

Nelder-Mead: Simplex

There are six operations:

- **Order:** order the value at the vertices of the simplex $f(x_1) \leq \ldots \leq f(x_{n+1})$

Nelder-Mead: Simplex

There are six operations:

- **Order:** order the value at the vertices of the simplex $f(x_1) \leq \ldots \leq f(x_{n+1})$
- **Centroid:** calculate x_0, the centroid of the non - x_{n+1} points

Nelder-Mead: Simplex

There are six operations:

- **Order:** order the value at the vertices of the simplex $f(x_1) \leq \ldots \leq f(x_{n+1})$
- **Centroid:** calculate x_0, the centroid of the non - x_{n+1} points
- **Reflection:** reflect x_{n+1} through the opposite face of the simplex and evaluate the new point: $x_r = x_0 + \alpha(x_0 - x_{n+1}), \alpha > 0$
    - If this improves upon the second-highest (e.g. its lower) but is not the lowest value point, replace x_{n+1} with x_r and restart
    - If this is the lowest value point so far, go to step 4
    - If $f(x_r) > f(x_n)$ go to step 5

Nelder-Mead: Simplex

- **Expansion:** push the reflected point further in the same direction

Nelder-Mead: Simplex

- **Expansion:** push the reflected point further in the same direction
- **Contract:** Contract the highest value point toward the middle
  - Compute $x_c = x_0 + \gamma(x_0 - x_{n+1}), 0 < \gamma \leq 0.5$
  - If x_c is better than the worst point replace x_{n+1} with x_c and restart
  - Else go to step 6

Nelder-Mead: Simplex

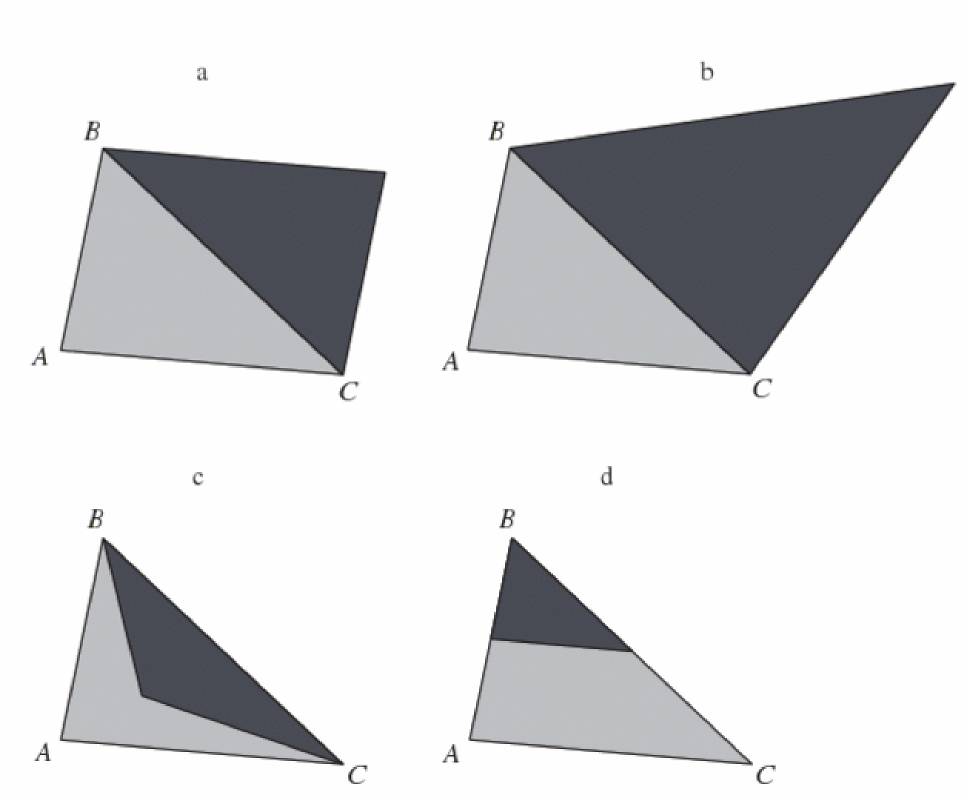
- **Expansion:** push the reflected point further in the same direction
- **Contract:** Contract the highest value point toward the middle
  - Compute $x_c = x_0 + \gamma(x_0 - x_{n+1}), 0 < \gamma \leq 0.5$
  - If x_c is better than the worst point replace x_{n+1} with x_c and restart
  - Else go to step 6
- **Shrink:** shrink the simplex toward the best point
  - Replace all points but the best one with $x_i = x_1 + \sigma(x_i - x_1)$

Nelder-Mead: Simplex

- **Expansion:** push the reflected point further in the same direction
- **Contract:** Contract the highest value point toward the middle
  - Compute $x_c = x_0 + \gamma(x_0 - x_{n+1}), 0 < \gamma \leq 0.5$
  - If x_c is better than the worst point replace x_{n+1} with x_c and restart
  - Else go to step 6
- **Shrink:** shrink the simplex toward the best point
  - Replace all points but the best one with $x_i = x_1 + \sigma(x_i - x_1)$

Nelder-Mead is a pain to code efficiently (i.e. don't spend the time doing it yourself) but is in the `Optim.jl` package

Nelder-Mead: Simplex



Nelder-Mead is commonly used but slow and unreliable, no real useful convergence properties, avoid using it

What is a solution?

We typically want to find a global extremum, here a minimum, of our objective function f

What is a solution?

We typically want to find a global extremum, here a minimum, of our objective function f

x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x over the domain of the function

What is a solution?

We typically want to find a global extremum, here a minimum, of our objective function f

x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x over the domain of the function

Problem: most algorithms are *local* minimizers that find a point x^* such that $f(x^*) \leq f(x)$ for all $x \in N$, where N is a neighborhood of x^*

What is a solution?

We typically want to find a global extremum, here a minimum, of our objective function f

x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x over the domain of the function

Problem: most algorithms are *local* minimizers that find a point x^* such that $f(x^*) \leq f(x)$ for all $x \in N$, where N is a neighborhood of x^*

Typically analytical problems are set up to have a unique minimum so any local solver can generally find the global optimum

What is a solution?

Lots of problems have properties that don't satisfy the typical sufficiency conditions for a unique minimum (strictly decreasing and convex), like

What is a solution?

Lots of problems have properties that don't satisfy the typical sufficiency conditions for a unique minimum (strictly decreasing and convex), like

- Concave transitions
- Games with multiple equilibria
- Etc

What is a solution?

Lots of problems have properties that don't satisfy the typical sufficiency conditions for a unique minimum (strictly decreasing and convex), like

- Concave transitions
- Games with multiple equilibria
- Etc

How do we find a local minimum?

What is a solution?

Lots of problems have properties that don't satisfy the typical sufficiency conditions for a unique minimum (strictly decreasing and convex), like

- Concave transitions
- Games with multiple equilibria
- Etc

How do we find a local minimum?

Do we need to evaluate every single point?

The general unconstrained approach

Optimization algorithms typically have the following set up:

1. Start at some x_0
2. Work through a series of iterates $\{x_k\}_{k=1}^{\infty}$ until we have "converged" with sufficient accuracy

The general unconstrained approach

Optimization algorithms typically have the following set up:

1. Start at some x_0
2. Work through a series of iterates $\{x_k\}_{k=1}^{\infty}$ until we have "converged" with sufficient accuracy

If the function is smooth, we can take advantage of that information about the function's shape to figure out which direction to move in next

The general unconstrained approach

Optimization algorithms typically have the following set up:

1. Start at some x_0
2. Work through a series of iterates $\{x_k\}_{k=1}^{\infty}$ until we have "converged" with sufficient accuracy

If the function is smooth, we can take advantage of that information about the function's shape to figure out which direction to move in next

If f is twice continuously differentiable, we can use the gradient ∇f and Hessian $\nabla^2 f$ to figure out if x^* is a local minimizer

The general unconstrained approach

Taylor's Theorem tells us that if f is twice differentiable, then there exists a $t \in (0, 1)$ such that

$$f(x^* + p) = f(x^*) + \nabla\, f(x^*)^T\, p + \frac{1}{2!}\, p^T\, \nabla^2\, f(x^* + tp)\, p$$

This is an exact equality

The general unconstrained approach

Taylor's Theorem tells us that if f is twice differentiable, then there exists a $t \in (0, 1)$ such that

$$f(x^* + p) = f(x^*) + \nabla f(x^*)^T p + \frac{1}{2!} p^T \nabla^2 f(x^* + tp) p$$

This is an exact equality

From here we can prove the usual necessary and sufficient conditions for a local optimum

Two large classes of algorithms

All modern algorithms have that general set up but may go about it in different ways

Two large classes of algorithms

All modern algorithms have that general set up but may go about it in different ways

Most modern optimization problems fall into one of two classes:

1. Line search
2. Trust region

Two large classes of algorithms

All modern algorithms have that general set up but may go about it in different ways

Most modern optimization problems fall into one of two classes:

1. Line search
2. Trust region

The relationship between these two approaches has a lot of similiarities to the relationship between the constrained problem and the dual Lagrange problem

Line search algorithms

General idea:

1. Start at some current iterate x_k
2. Select a direction to move in p_k
3. Figure out how far along p_k to move

Line search algorithms

How do we figure out how far to move?

Line search algorithms

How do we figure out how far to move?

"Approximately" solve this problem to figure out the **step length** α

$$\min_{\alpha>0} f(x_k + \alpha p_k)$$

Line search algorithms

How do we figure out how far to move?

"Approximately" solve this problem to figure out the **step length** α

$$\min_{\alpha>0} f(x_k + \alpha p_k)$$

We are finding the distance to move, α in direction p_k that minimizes our objective f

Line search algorithms

Typically do not perform the full minimization problem since it is costly

We only try a limited number of step lengths α before picking the best one and moving onto our next iterate x_{k+1}

Line search algorithms

Typically do not perform the full minimization problem since it is costly

We only try a limited number of step lengths α before picking the best one and moving onto our next iterate x_{k+1}

We still haven't answered, what direction p_k do we decide to move in?

Line search: direction choice

What's an obvious choice for p_k?

Line search: direction choice

What's an obvious choice for p_k?

The direction that yields the *steepest descent*

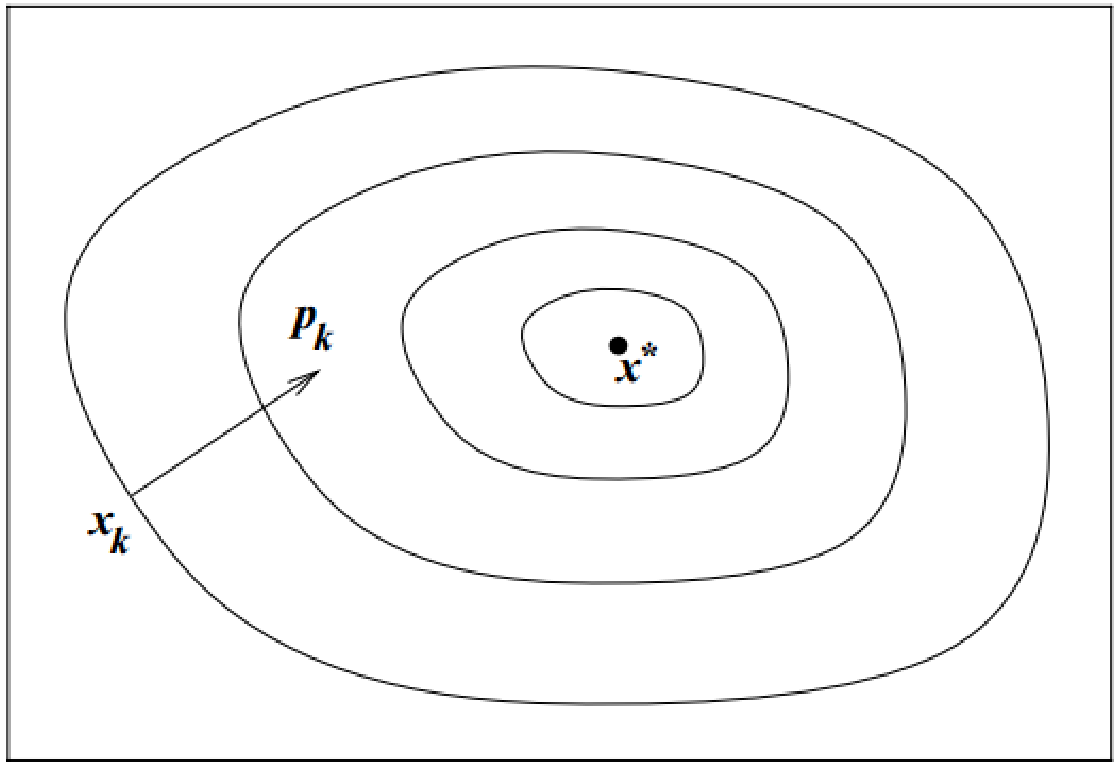
Line search: direction choice

What's an obvious choice for p_k?

The direction that yields the *steepest descent*

$-\nabla f_k$ is the direction that makes f decrease most rapidly, k indicates we are evaluating f at iteration k

Line search algorithms



Line search: steepest descent

We can verify this is the direction of steepest descent by referring to Taylor's theorem

Line search: steepest descent

We can verify this is the direction of steepest descent by referring to Taylor's theorem

For any direction p and step length α, we have that

$$f(x_k + \alpha p) = f(x_k) + \alpha\, p^T \nabla f_k + \frac{1}{2!}\, \alpha^2 p^T \nabla^2 f(x_k + tp)\, p$$

Line search: steepest descent

We can verify this is the direction of steepest descent by referring to Taylor's theorem

For any direction p and step length α, we have that

$$f(x_k + \alpha p) = f(x_k) + \alpha\, p^T \nabla f_k + \frac{1}{2!}\, \alpha^2 p^T \nabla^2 f(x_k + tp)\, p$$

The rate of change in f along p at x_k $(\alpha = 0)$ is $p^T \nabla f_k$

Line search: steepest descent

The the unit vector of quickest descent solves

$$\min_p p^T \nabla f_k \quad \text{subject to: } ||p|| = 1$$

Line search: steepest descent

The the unit vector of quickest descent solves

$$\min_p p^T \nabla f_k \quad \text{subject to: } ||p|| = 1$$

Re-express the objective as $\min_{\theta,p} ||p|| \, ||\nabla f_k|| cos \, \theta$, where θ is the angle between p and ∇f_k

Line search: steepest descent

The the unit vector of quickest descent solves

$$\min_p p^T \nabla f_k \quad \text{subject to: } ||p|| = 1$$

Re-express the objective as $\min_{\theta,p} ||p||\,||\nabla f_k|| cos\,\theta$, where θ is the angle between p and ∇f_k

The minimum is attained when $cos\,\theta = -1$ and $p = -\frac{\nabla f_k}{||\nabla f_k||}$, so the direction of steepest descent is simply $-\nabla f_k$

Line search: steepest descent

The *steepest descent method* searches along this direction at every iteration k

It may select the step length α_k in several different ways

A benefit of the algorithm is that we only require the gradient of the function, and no Hessian

However it can be very slow

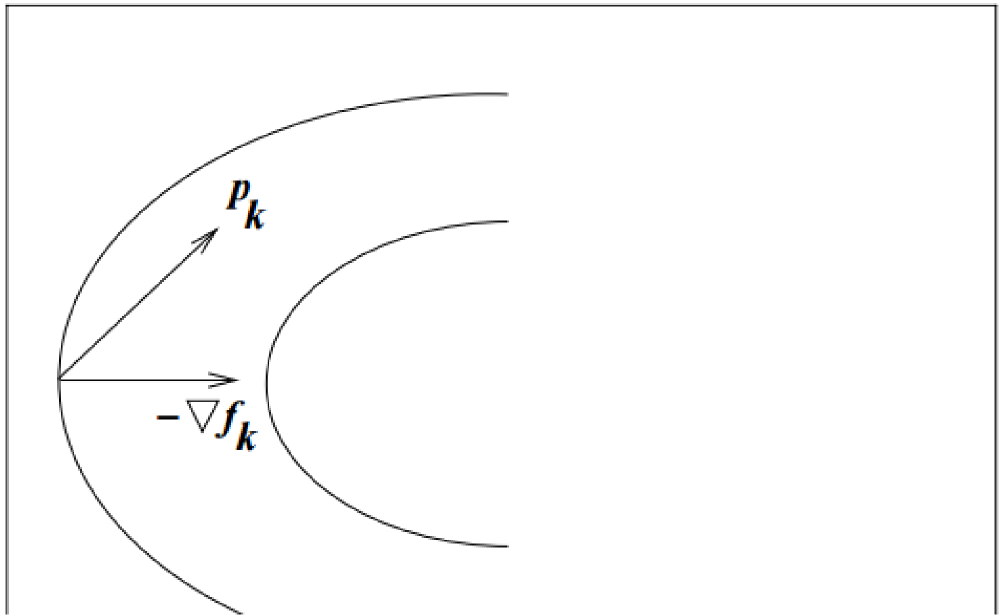
Line search: alternative directions

We can always use search directions other than the steepest descent

Line search: alternative directions

We can always use search directions other than the steepest descent

Any descent direction, i.e. one that is within 45° of $-\nabla f_k$,
is *guaranteed* to produce a decrease in f as long as the step size is sufficiently small



Line search: alternative directions

We can actually verify this with Taylor's theorem

Line search: alternative directions

We can actually verify this with Taylor's theorem

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \,\nabla\, f_k + O(\epsilon^2)$$

Line search: alternative directions

We can actually verify this with Taylor's theorem

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \nabla f_k + O(\epsilon^2)$$

If p_k is in a descending direction, θ_k will be of an angle such that $cos\, \theta_k < 0$

This gives us

Line search: alternative directions

We can actually verify this with Taylor's theorem

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \nabla f_k + O(\epsilon^2)$$

If p_k is in a descending direction, θ_k will be of an angle such that $cos\, \theta_k < 0$

This gives us

$$p_k^T \nabla f_k = ||p_k||\, ||\nabla f_k|| cos\, \theta_k < 0$$

Line search: alternative directions

We can actually verify this with Taylor's theorem

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \nabla f_k + O(\epsilon^2)$$

If p_k is in a descending direction, θ_k will be of an angle such that $cos\, \theta_k < 0$

This gives us

$$p_k^T \nabla f_k = ||p_k||\, ||\nabla f_k|| cos\, \theta_k < 0$$

Therefore $f(x_k + \epsilon p_k) < f(x_k)$ for positive but sufficiently small ϵ

Line search: alternative directions

We can actually verify this with Taylor's theorem

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \nabla f_k + O(\epsilon^2)$$

If p_k is in a descending direction, θ_k will be of an angle such that $cos\, \theta_k < 0$

This gives us

$$p_k^T \nabla f_k = ||p_k||\, ||\nabla f_k|| cos\, \theta_k < 0$$

Therefore $f(x_k + \epsilon p_k) < f(x_k)$ for positive but sufficiently small ϵ

Is $-\nabla f_k$ always the best search direction?

Newton's method

The most important search direction is not steepest descent but **Newton's direction**

Newton's method

The most important search direction is not steepest descent but **Newton's direction**

Newton's direction comes out of the second order Taylor series approximation to $f(x_k + p)$

$$f(x_k + p) \approx f_k + p^T \nabla f_k + \frac{1}{2!} p^T \nabla^2 f_k \, p$$

Define this as $\mathbf{m_k(p)}$

Newton's method

We find the Newton direction by selecting the vector p that minimizes $f(x_k + p)$

Newton's method

We find the Newton direction by selecting the vector p that minimizes $f(x_k + p)$

This ends up being

$$p_k^N = -\frac{\nabla f_k}{\nabla^2 f_k}$$

Newton's method

Newton's method

This approximation to the function we are trying to solve has error of $O(||p||^3)$

,

so if p is small, the quadratic approximation is very accurate

Newton's method

This approximation to the function we are trying to solve has error of $O(||p||^3)$

,

so if p is small, the quadratic approximation is very accurate

Drawback: requires explicit computation of the Hessian, $\nabla^2 f(x)$

- Quasi-Newton solvers also exist (e.g. BFGS, L-BFGS, etc)

Trust region methods

Trust region methods construct an approximating model, m_k whose behavior near the current iterate x_k is close to that of the actual function f

Trust region methods

Trust region methods construct an approximating model, m_k whose behavior near the current iterate x_k is close to that of the actual function f

We then search for a minimizer of m_k

Trust region methods

Trust region methods construct an approximating model, m_k
whose behavior near the current iterate x_k is close to that of the actual function f

We then search for a minimizer of m_k

Issue: m_k may not represent f well when far away from the current iterate x_k

Trust region methods

Trust region methods construct an approximating model, m_k whose behavior near the current iterate x_k is close to that of the actual function f

We then search for a minimizer of m_k

Issue: m_k may not represent f well when far away from the current iterate x_k

Solution: Restrict the search for a minimizer to be within some region of x_k, called a **trust region**

Trust region methods

Trust region problems can be formulated as

$$\min_p m_k(x_k + p)$$

where

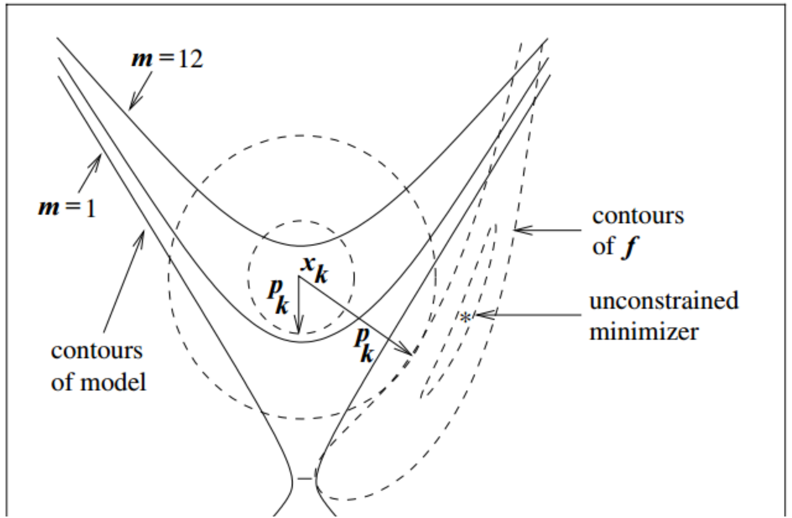
- $x_k + p \in \Gamma$
- Γ is a ball defined by $||p||_2 \leq \Delta$
- Δ is called the trust region radius

Trust region Methods

Typically the approximating model m_k is a quadratic function (i.e. a second-order Taylor approximation)

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2!} p^T B_k p$$

where B_k is the Hessian or an approximation to the Hessian



Line search vs trust region

Whats the fundamental difference between line search and trust region?

Line search vs trust region

Whats the fundamental difference between line search and trust region?

Line search first picks a direction then searches along that direction for the optimal step length

Trust region first defines our step length via the trust region radius, then searches for the optimal direction

Line search vs trust region

There is a special case of the trust region where if we set B_k, the approximate Hessian, to zero, the solution to the problem is

$$p_k = -\frac{\Delta_k \nabla f_k}{||\nabla f_k||}$$

This is just the steepest descent solution for the line search problem

Problem scaling

The **scaling** of a problem matters for optimization performance

Problem scaling

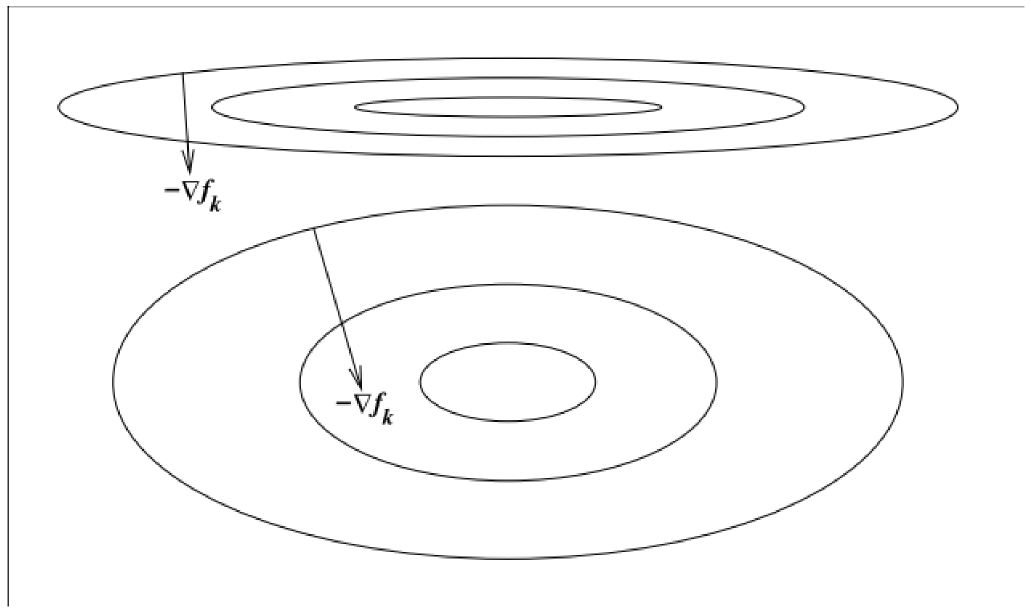
The **scaling** of a problem matters for optimization performance

A problem is **poorly scaled** if changes to x in a certain direction
produce much bigger changes in f than changes to in x in another direction

Problem scaling

The scaling of a problem matters for optimization performance

A problem is poorly scaled if changes to x in a certain direction produce much bigger changes in f than changes to in x in another direction



Problem scaling

Ex: $f(x) = 10^9 x_1^2 + x_2^2$ is poorly scaled

Problem scaling

Ex: $f(x) = 10^9 x_1^2 + x_2^2$ is poorly scaled

This happens when things change at different rates:

- Investment rates between 0 and 1, but global consumption is in dollars

Problem scaling

Ex: $f(x) = 10^9 x_1^2 + x_2^2$ is poorly scaled

This happens when things change at different rates:

- Investment rates between 0 and 1, but global consumption is in dollars

How do we solve this issue?

Problem scaling

Ex: $f(x) = 10^9 x_1^2 + x_2^2$ is poorly scaled

This happens when things change at different rates:

- Investment rates between 0 and 1, but global consumption is in dollars

How do we solve this issue?

Rescale the problem: put them in units that are generally within an order of magnitude of 1

- Investment rate in percentage terms: $0\% - 100\%$
- Consumption in units of trillion dollars instead of dollars

Constrained optimization

How do we solve constrained optimization problems?

Constrained optimization

How do we solve constrained optimization problems?

Typically as a variant of unconstrained optimization techniques

Constrained optimization

How do we solve constrained optimization problems?

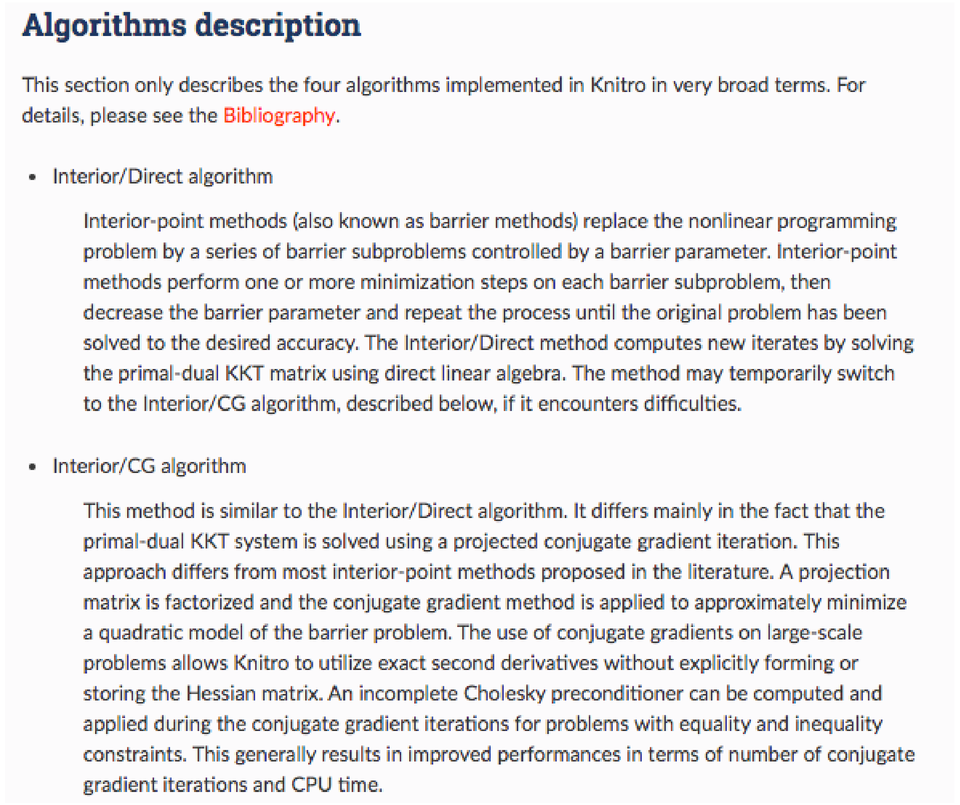
Typically as a variant of unconstrained optimization techniques

We will discuss three types of constrained optimization algorithms

- Penalty methods
- Active set methods
- Interior point methods

Constrained optimization

These are the algorithms in workhorse commercial solvers: KNITRO



Algorithms description

This section only describes the four algorithms implemented in Knitro in very broad terms. For details, please see the Bibliography.

- Interior/Direct algorithm

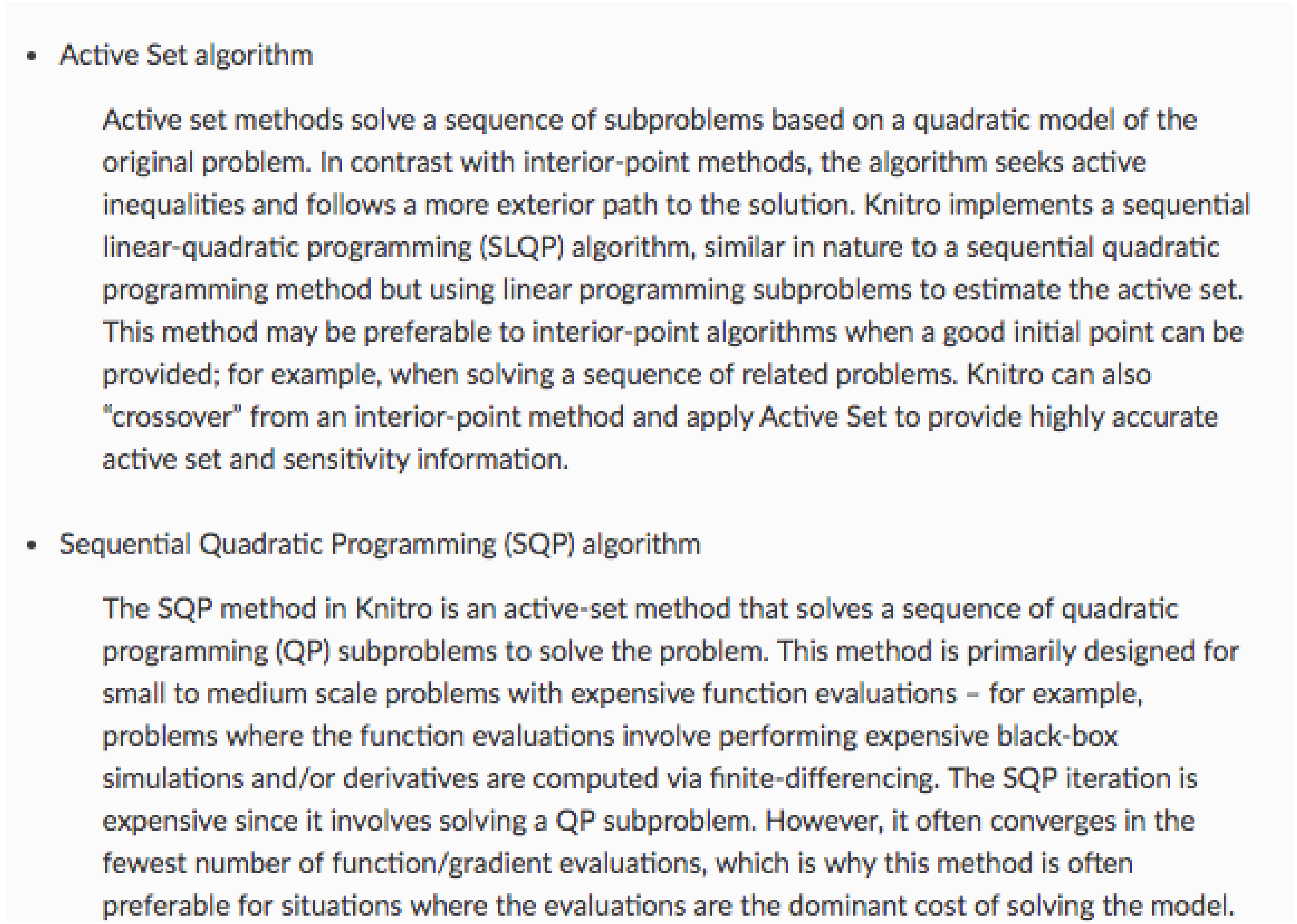
  Interior-point methods (also known as barrier methods) replace the nonlinear programming problem by a series of barrier subproblems controlled by a barrier parameter. Interior-point methods perform one or more minimization steps on each barrier subproblem, then decrease the barrier parameter and repeat the process until the original problem has been solved to the desired accuracy. The Interior/Direct method computes new iterates by solving the primal-dual KKT matrix using direct linear algebra. The method may temporarily switch to the Interior/CG algorithm, described below, if it encounters difficulties.

- Interior/CG algorithm

  This method is similar to the Interior/Direct algorithm. It differs mainly in the fact that the primal-dual KKT system is solved using a projected conjugate gradient iteration. This approach differs from most interior-point methods proposed in the literature. A projection matrix is factorized and the conjugate gradient method is applied to approximately minimize a quadratic model of the barrier problem. The use of conjugate gradients on large-scale problems allows Knitro to utilize exact second derivatives without explicitly forming or storing the Hessian matrix. An incomplete Cholesky preconditioner can be computed and applied during the conjugate gradient iterations for problems with equality and inequality constraints. This generally results in improved performances in terms of number of conjugate gradient iterations and CPU time.

Constrained optimization

These are the algorithms in workhorse commercial solvers: KNITRO



- Active Set algorithm

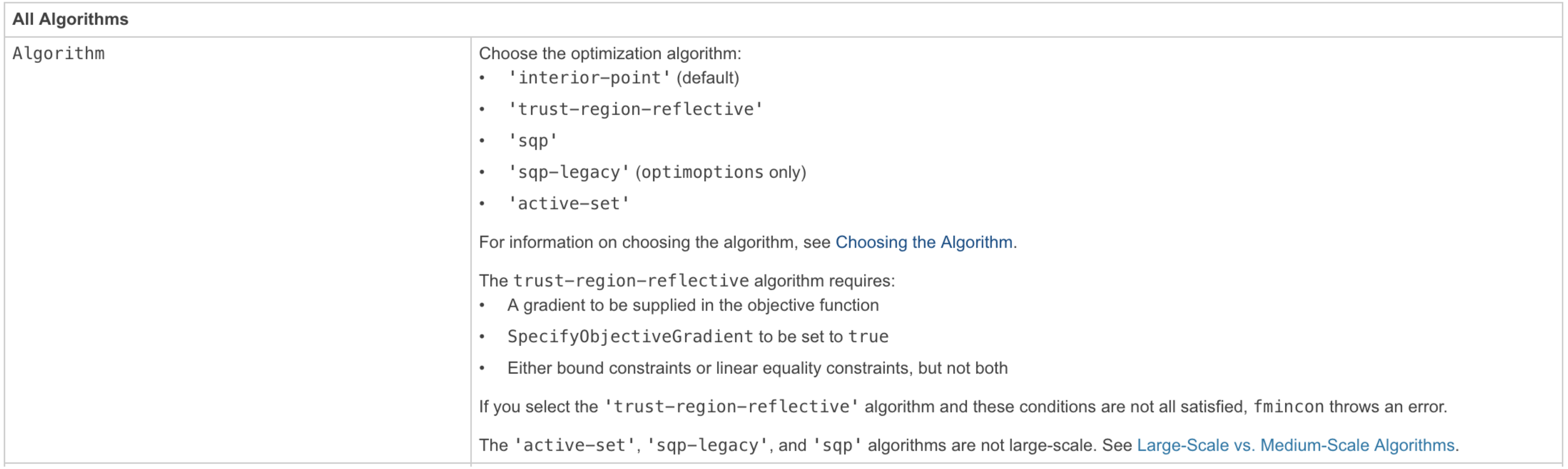
  Active set methods solve a sequence of subproblems based on a quadratic model of the original problem. In contrast with interior-point methods, the algorithm seeks active inequalities and follows a more exterior path to the solution. Knitro implements a sequential linear-quadratic programming (SLQP) algorithm, similar in nature to a sequential quadratic programming method but using linear programming subproblems to estimate the active set. This method may be preferable to interior-point algorithms when a good initial point can be provided; for example, when solving a sequence of related problems. Knitro can also "crossover" from an interior-point method and apply Active Set to provide highly accurate active set and sensitivity information.

- Sequential Quadratic Programming (SQP) algorithm

  The SQP method in Knitro is an active-set method that solves a sequence of quadratic programming (QP) subproblems to solve the problem. This method is primarily designed for small to medium scale problems with expensive function evaluations – for example, problems where the function evaluations involve performing expensive black-box simulations and/or derivatives are computed via finite-differencing. The SQP iteration is expensive since it involves solving a QP subproblem. However, it often converges in the fewest number of function/gradient evaluations, which is why this method is often preferable for situations where the evaluations are the dominant cost of solving the model.

Constrained optimization

These are the algorithms in workhorse commercial solvers: fmincon/MATLAB



All Algorithms	
`Algorithm`	Choose the optimization algorithm: • `'interior-point'` (default) • `'trust-region-reflective'` • `'sqp'` • `'sqp-legacy'` (`optimoptions` only) • `'active-set'` For information on choosing the algorithm, see Choosing the Algorithm. The `trust-region-reflective` algorithm requires: • A gradient to be supplied in the objective function • `SpecifyObjectiveGradient` to be set to `true` • Either bound constraints or linear equality constraints, but not both If you select the `'trust-region-reflective'` algorithm and these conditions are not all satisfied, `fmincon` throws an error. The `'active-set'`, `'sqp-legacy'`, and `'sqp'` algorithms are not large-scale. See Large-Scale vs. Medium-Scale Algorithms.

Constrained optimization: Penalty methods

Suppose we wish to minimize some function subject to equality constraints (easily generalizes to inequality)

$$\min_x f(x) \ \text{ subject to: } c_i(x) = 0$$

Constrained optimization: Penalty methods

Suppose we wish to minimize some function subject to equality constraints (easily generalizes to inequality)

$$\min_x f(x) \;\; \text{subject to: } c_i(x) = 0$$

How does an algorithm know to not violate the constraint?

Constrained optimization: Penalty methods

Suppose we wish to minimize some function subject to equality constraints (easily generalizes to inequality)

$$\min_x f(x) \ \text{ subject to: } c_i(x) = 0$$

How does an algorithm know to not violate the constraint?

One way is to introduce a **penalty function** into our objective and remove the constraint:

$$Q(x;\mu) = f(x) + \frac{\mu}{2}\sum_i c_i^2(x)$$

where μ is the penalty parameter

Constrained optimization: Penalty methods

$$Q(x;\mu) = f(x) + \frac{\mu}{2}\sum_i c_i^2(x)$$

Constrained optimization: Penalty methods

$$Q(x;\mu) = f(x) + \frac{\mu}{2} \sum_i c_i^2(x)$$

The second term increases the value of the function, bigger $\mu \rightarrow$ bigger penalty from violating the constraint

Constrained optimization: Penalty methods

$$Q(x;\mu) = f(x) + \frac{\mu}{2} \sum_i c_i^2(x)$$

The second term increases the value of the function, bigger $\mu \rightarrow$ bigger penalty from violating the constraint

The penalty terms are smooth $\rightarrow$ use unconstrained optimization techniques to solve the problem by searching for iterates of x_k

Constrained optimization: Penalty methods

Also generally iterate on sequences of $\mu_k \to \infty$ as $k \to \infty$, to require satisfying the constraints as we close in

Constrained optimization: Penalty methods

Also generally iterate on sequences of $\mu_k \to \infty$ as $k \to \infty$, to require satisfying the constraints as we close in

There are also augmented Lagrangian methods that take the quadratic penalty method and add in explicit estimates of Lagrange multipliers to help force binding constraints to bind precisely

Constrained optimization: Penalty method example

Example:

$$\min x_1 + x_2 \quad \text{subject to:} \quad x_1^2 + x_2^2 - 2 = 0$$

Constrained optimization: Penalty method example

Example:

$$\min x_1 + x_2 \quad \text{subject to:} \quad x_1^2 + x_2^2 - 2 = 0$$

Solution is pretty easy to show to be $(-1, -1)$

Constrained optimization: Penalty method example

Example:

$$\min x_1 + x_2 \quad \text{subject to:} \quad x_1^2 + x_2^2 - 2 = 0$$

Solution is pretty easy to show to be $(-1, -1)$

The penalty method function $Q(x_1, x_2; \mu)$ is

$$Q(x_1, x_2; \mu) = x_1 + x_2 + \frac{\mu}{2}(x_1^2 + x_2^2 - 2)^2$$

Constrained optimization: Penalty method example

Example:

$$\min x_1 + x_2 \quad \text{subject to:} \quad x_1^2 + x_2^2 - 2 = 0$$

Solution is pretty easy to show to be $(-1, -1)$

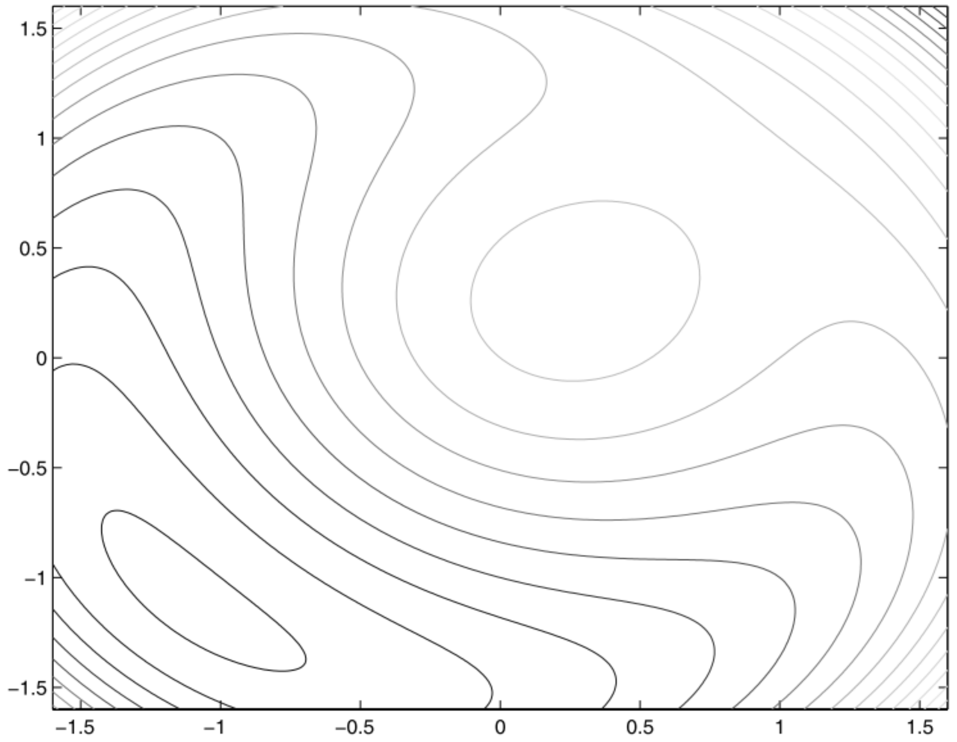
The penalty method function $Q(x_1, x_2; \mu)$ is

$$Q(x_1, x_2; \mu) = x_1 + x_2 + \frac{\mu}{2}(x_1^2 + x_2^2 - 2)^2$$

Let's ramp up μ and see what happens to how the function looks

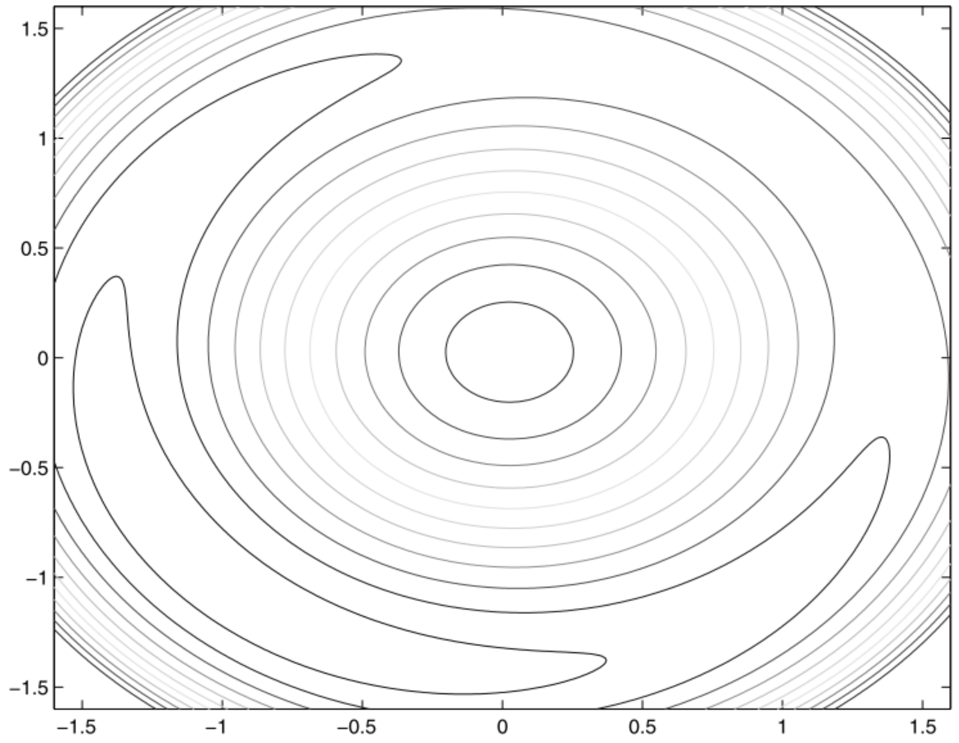
Constrained optimization: Penalty method example

$\mu = 1$, solution is around $(-1.1, -1.1)$



Constrained optimization: Penalty method example

$\mu = 10$, solution is very close to $(-1, -1)$, can easily see trough, and rapid value increase outside $x_1^2 + x_2^2 = 2$



Constrained optimization: Active set methods

Active set methods encapsulate sequential quadratic programming (SQP) methods

Constrained optimization: Active set methods

Active set methods encapsulate sequential quadratic programming (SQP) methods

Main idea:

1. Replace the large non-linear constrained problem with a constrained quadratic programming problem
2. Use Newton's method to solve the sequence of simpler quadratic problems

Constrained optimization: Active set methods

The Lagrangian is

$$L(x, \lambda) = f(x) - \lambda^T c(x)$$

Constrained optimization: Active set methods

The Lagrangian is

$$L(x, \lambda) = f(x) - \lambda^T c(x)$$

Denote $A(x)^T$ as the Jacobian of the constraints

$$A(x)^T = [\nabla c_1(x), \dots, \nabla c_m(x)]$$

Constrained optimization: Active set methods

The first-order conditions $F(x, \lambda)$ can be written as,

$$\nabla f(x) - A(x)^T \lambda = 0$$
$$c(x) = 0$$

Any solution to the equality constrained problem, where $A(x^*)$ has full rank also satisfies the first-order necessary conditions

Constrained optimization: Active set methods

The first-order conditions $F(x, \lambda)$ can be written as,

$$\nabla f(x) - A(x)^T \lambda = 0$$
$$c(x) = 0$$

Any solution to the equality constrained problem, where $A(x^*)$ has full rank also satisfies the first-order necessary conditions

Active set methods use Newton's method to find the solution (x^*, λ^*) of $F(x, \lambda)$

Constrained optimization: Active set methods

Issue: if we have many constraints, keeping track of all of them can be expensive

Constrained optimization: Active set methods

Issue: if we have many constraints, keeping track of all of them can be expensive

Main idea: recognize that if an inequality constraint is not binding, or **active**, then it has no influence on the solution

$\rightarrow$ in the iteration procedure we can effectively ignore it

Constrained optimization: Active set methods

Issue: if we have many constraints, keeping track of all of them can be expensive

Main idea: recognize that if an inequality constraint is not binding, or **active**, then it has no influence on the solution

$\rightarrow$ in the iteration procedure we can effectively ignore it

Active set methods find ways to reduce the complexity of the optimization routine
by selectively ignoring constraints that are not active (i.e. non-positive Lagrange multipliers) or close to being active

Constrained optimization: Interior point methods

Interior point methods are also called barrier methods

Constrained optimization: Interior point methods

Interior point methods are also called barrier methods

These are typically used for inequality constrained problems

Constrained optimization: Interior point methods

Interior point methods are also called barrier methods

These are typically used for inequality constrained problems

The name **interior point** comes from the algorithm traversing the domain along the interior of the inequality constraints

Constrained optimization: Interior point methods

Interior point methods are also called barrier methods

These are typically used for inequality constrained problems

The name **interior point** comes from the algorithm traversing the domain along the interior of the inequality constraints

Issue: how do we ensure we are on the interior of the feasible set?

Constrained optimization: Interior point methods

Interior point methods are also called barrier methods

These are typically used for inequality constrained problems

The name **interior point** comes from the algorithm traversing the domain along the interior of the inequality constraints

Issue: how do we ensure we are on the interior of the feasible set?

Main idea: impose a **barrier** to stop the solver from letting a constraint bind

Constrained optimization: Interior point methods

Consider the following constrained optimization problem

$$\min_x f(x)$$
$$\text{subject to: } c_E(x) = 0, c_I(x) \geq 0$$

Constrained optimization: Interior point methods

Consider the following constrained optimization problem

$$\min_{x} f(x)$$
$$\text{subject to: } c_E(x) = 0, c_I(x) \geq 0$$

Reformulate this problem as

$$\min_{x,s} f(x)$$
$$\text{subject to: } c_E(x) = 0, c_I(x) - s = 0, s \geq 0$$

where s is a vector of slack variables for the constraints

Constrained optimization: Interior point methods

Final step: introduce a **barrier function** to eliminate the inequality constraint,

$$\min_{x,s} f(x) - \mu \sum_{i=1}^{m} log(s_i)$$
$$\text{subject to: } c_E(x) = 0, c_I(x) - s = 0$$

where μ is a positive barrier parameter

Constrained optimization: Interior point methods

The barrier function prevents the components of s from approaching zero by imposing a logarithmic barrier $\rightarrow$ it maintains slack in the constraints

Constrained optimization: Interior point methods

The barrier function prevents the components of s from approaching zero by imposing a logarithmic barrier $\rightarrow$ it maintains slack in the constraints

Interior point methods solve a sequence of barrier problems until $\{\mu_k\}$ converges to zero

Constrained optimization: Interior point methods

The barrier function prevents the components of s from approaching zero by imposing a logarithmic barrier $\rightarrow$ it maintains slack in the constraints

Interior point methods solve a sequence of barrier problems until $\{\mu_k\}$ converges to zero

The solution to the barrier problem converges to that of the original problem

Best practices for optimization

Plug in your guess, let the solver go, and you're done right?

Best practices for optimization

Plug in your guess, let the solver go, and you're done right?

WRONG

Best practices for optimization

Plug in your guess, let the solver go, and you're done right?

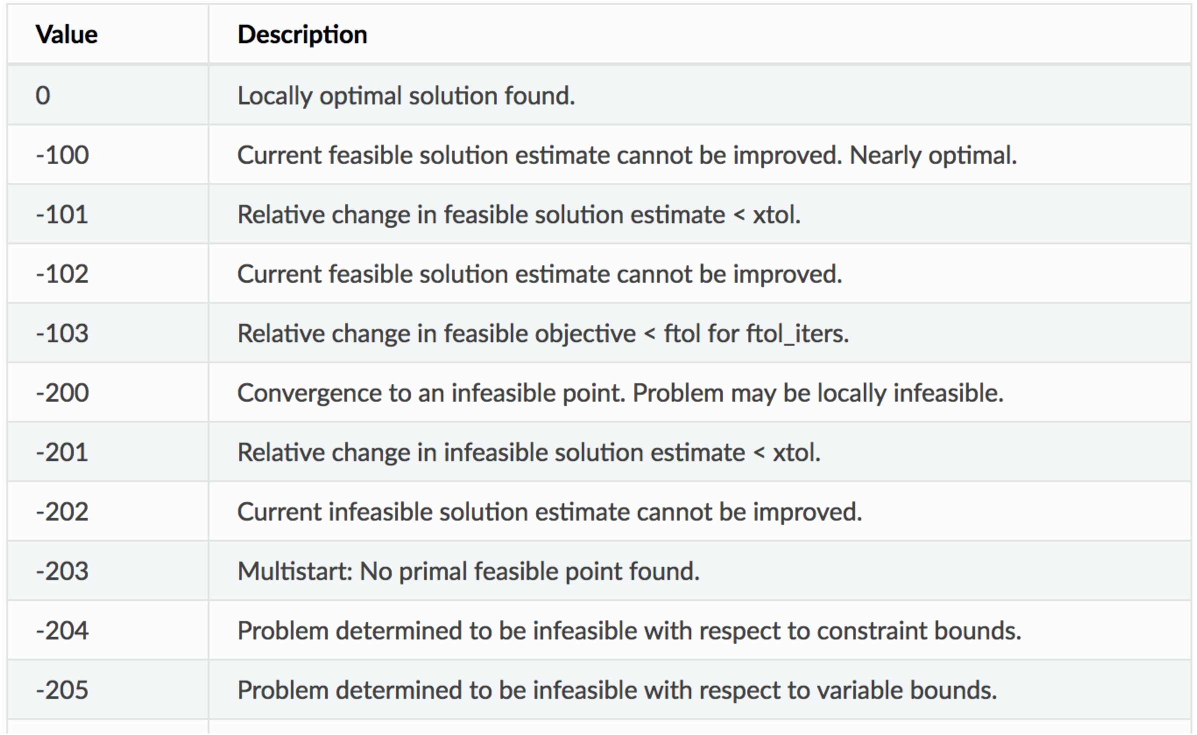
WRONG

These algorithms are not guaranteed to always find even a local solution, you need to test and make sure you are converging correctly

Check exitflags: KNITRO-specific numbers here

Exitflags tell you why the solver stopped, exit flags of 0 or -10X are generally good, anything else is bad

-10X can indicate bad scaling, ill-conditioning, etc



Value	Description
0	Locally optimal solution found.
-100	Current feasible solution estimate cannot be improved. Nearly optimal.
-101	Relative change in feasible solution estimate < xtol.
-102	Current feasible solution estimate cannot be improved.
-103	Relative change in feasible objective < ftol for ftol_iters.
-200	Convergence to an infeasible point. Problem may be locally infeasible.
-201	Relative change in infeasible solution estimate < xtol.
-202	Current infeasible solution estimate cannot be improved.
-203	Multistart: No primal feasible point found.
-204	Problem determined to be infeasible with respect to constraint bounds.
-205	Problem determined to be infeasible with respect to variable bounds.

Try alternative algorithms

Optimization is approximately 53% art

Try alternative algorithms

Optimization is approximately 53% art

Not all algorithms are suited for every problem $\rightarrow$ it is useful to check how different algorithms perform

Try alternative algorithms

Optimization is approximately 53% art

Not all algorithms are suited for every problem $\rightarrow$ it is useful to check how different algorithms perform

Interior-point is usually the default in constrained optimization solvers (low memory usage, fast), but try other algorithms and see if the solution generally remains the same

Be aware of tolerances

Two main tolerances in optimization:

1. `ftol` is the tolerance for the change in the function value (absolute and relative)
2. `xtol` is the tolerance for the change in the input values (absolute and relative)

Be aware of tolerances

Two main tolerances in optimization:

1. `ftol` is the tolerance for the change in the function value (absolute and relative)
2. `xtol` is the tolerance for the change in the input values (absolute and relative)

What is a suitable tolerance?

Be aware of tolerances

It depends

Be aware of tolerances

It depends

Explore sensitivity to tolerance, typically pick a conservative (small) number

- Defaults in solvers are usually `1e-6`

Be aware of tolerances

May be a substantial tradeoff between accuracy of your solution and speed

Be aware of tolerances

May be a substantial tradeoff between accuracy of your solution and speed

Common bad practice is to pick a larger tolerance (e.g. `1e-3`) so the problem "works" (e.g. so your big MLE converges)

Be aware of tolerances

May be a substantial tradeoff between accuracy of your solution and speed

Common bad practice is to pick a larger tolerance (e.g. `1e-3`) so the problem "works" (e.g. so your big MLE converges)

Issue is that 1e-3 might be pretty big for your problem
if you haven't checked that your solution is not sensitive to the tolerance

Perturb your initial guesses

Initial guesses matter

Perturb your initial guesses

Initial guesses matter

Good ones can improve performance

- e.g. initial guess for next iteration of coefficient estimates should be current iteration estimates

Perturb your initial guesses

Initial guesses matter

Good ones can improve performance

- e.g. initial guess for next iteration of coefficient estimates should be current iteration estimates

Bad ones can give you terrible performance, or wrong answers if your problem isn't perfect

- e.g. bad scaling, not well-conditioned, multiple equilibria