AGEC 652 - Lecture 4.2

Constrained optimization

Diego S. Cardoso

Spring 2022

Course roadmap

1. Intro to Scientific Computing
2. Numerical operations and representations
3. Systems of equations
4. **Optimization**
    1. Unconstrained optimization
    2. **Constrained optimization** $\leftarrow$ You are here
5. Structural estimation

*These slides are based on Miranda & Fackler (2002), Nocedal & Wright (2006), Judd (1998), and course materials by Ivan Rudik and Florian Oswald.

Constrained optimization setup

We want to solve

$$\min_x f(x)$$

subject to

$$g(x) = 0$$
$$h(x) \leq 0$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, $h : \mathbb{R}^n \to \mathbb{R}^l$, and f, g, and h are twice continuously differentiable

- We have m *equality constraints* and l *inequality constraints*

Constraint types

Constraints come in two types: equality or inequality

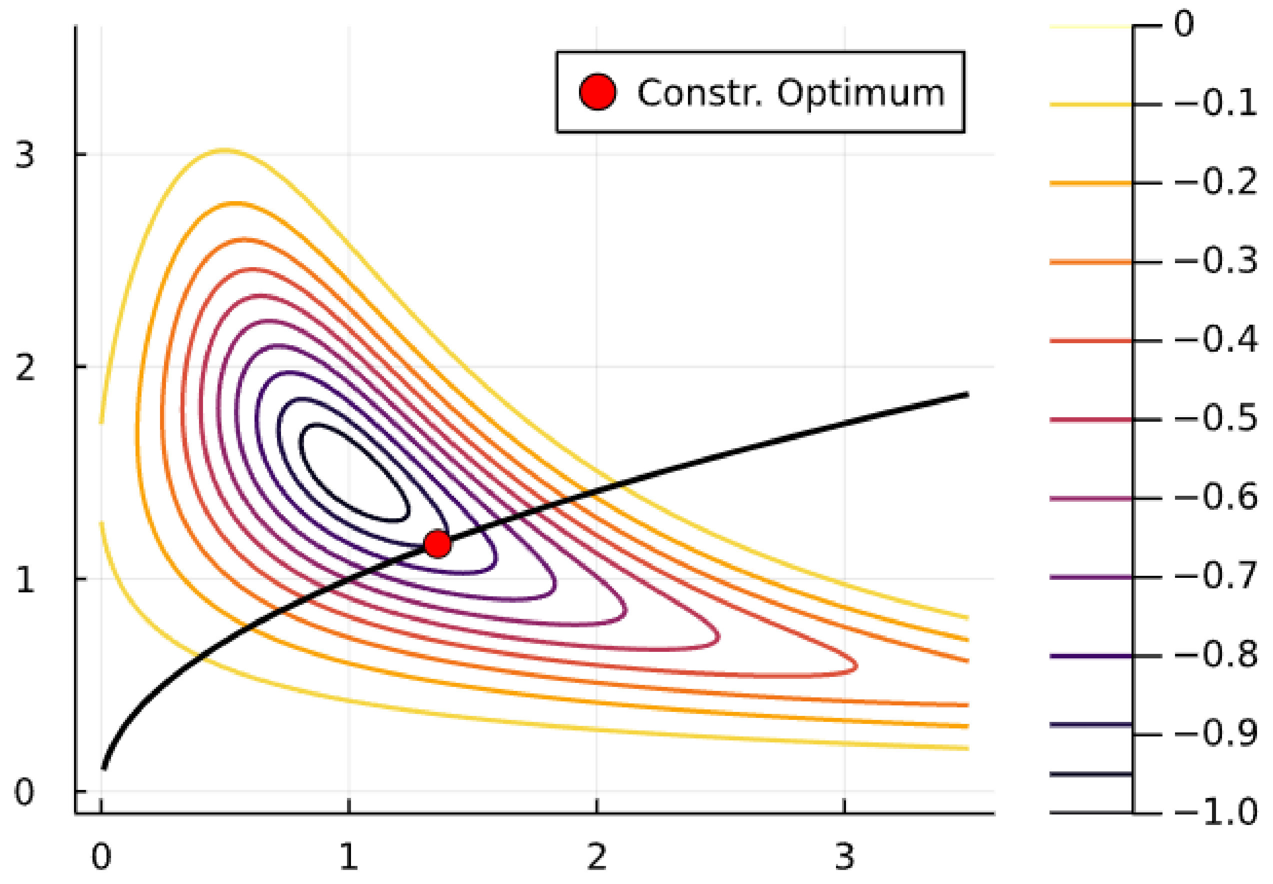
Let's see a an illustration with a single constraint. Consider the optimization problem

$$\min_x -exp\left(-(x_1x_2 - 1.5)^2 - (x_2 - 1.5)^2\right)$$

subject to $x_1 - x_2^2 = 0$

- The equality constraint limits solutions along the curve where $x_1 = x_2^2$

Constraint types



Constraint types

The problem can also be formulated with an inequality constraint

$$\min_x -exp\left(-(x_1x_2 - 1.5)^2 - (x_2 - 1.5)^2\right)$$

subject to $-x_1 + x_2^2 \leq 0$

How would that change feasible set compared to the equality constraint?

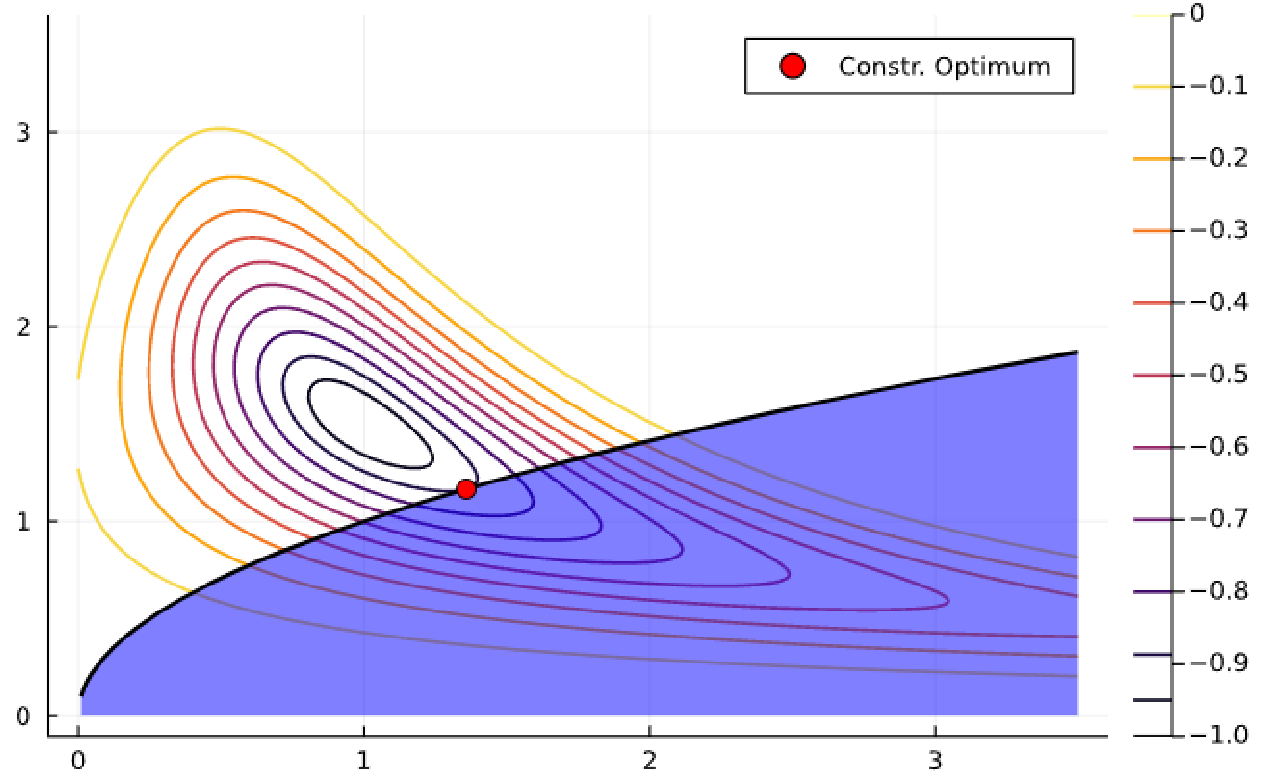
Constraint types

The feasible set is in blue

- It extends below and to the right

The solution in this case is along the boundaries of the feasible set

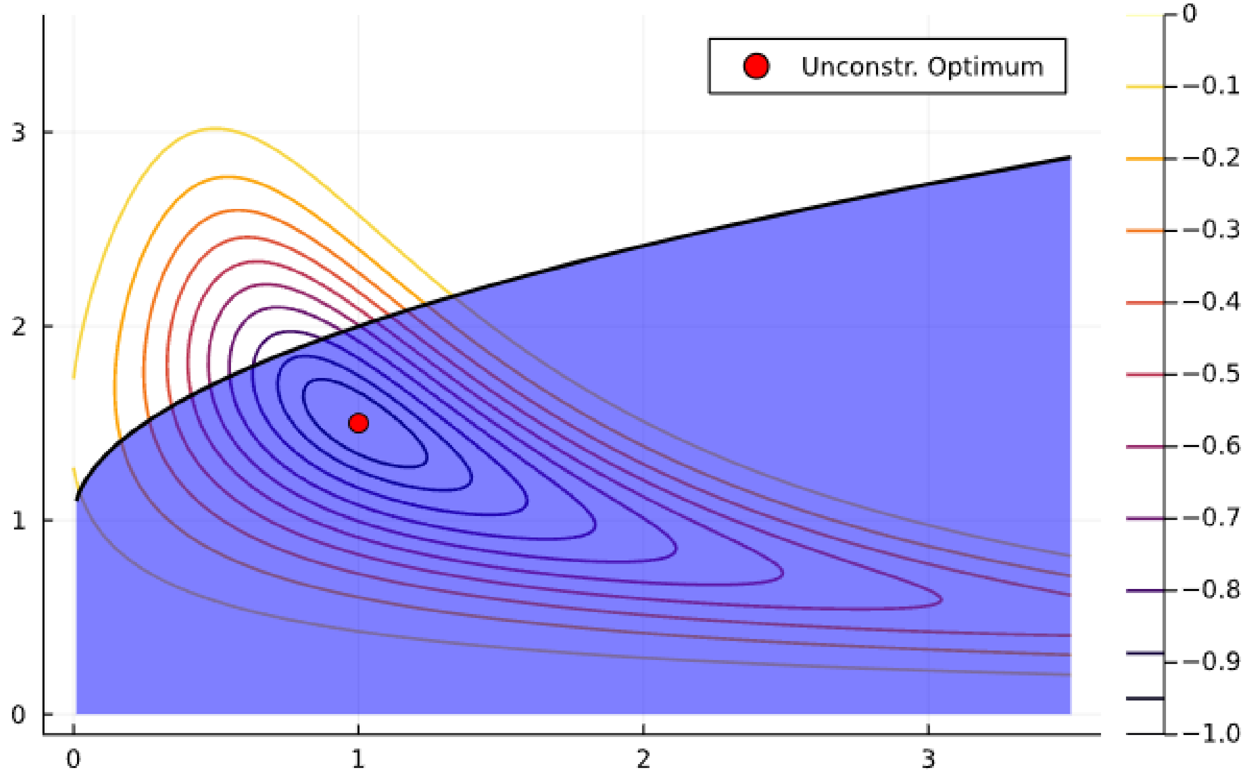
- It coincides with the equality constraint
- In those cases, we say the constraint is **binding** or **active**



Constraint types

If the solution is *interior* to the feasible set, we say the constraint is **slack** or **inactive**

- The solution to the constrained optimization problem is the same as the unconstrained one



Solving constrained optimization problems

You may recall from Math Econ courses that, under certain conditions, we can solve a constrained optimization problem by solving instead the corresponding *mixed complementary problem* using the first order conditions

That trick follows from the **Karush-Kuhn-Tucker (KKT) Theorem**

What does it say?

Karush-Kuhn-Tucker Theorem

> If x^* is a local minimizer and the constraint qualification[1] holds, then there are multipliers $\lambda^* \in \mathbb{R}^m$ and $\mu^* \in \mathbb{R}^l$ such that x^* is a stationary point of $\mathcal{L}$, the *Lagrangian*

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T g(x) + \mu^T h(x)$$

- Variables λ and μ are called *Lagrange multipliers* and in Economics have the intepretation of shadow prices

How does this theorem help us?

[1]Constraint qualification, or regularity conditions, can be formulated depending on the nature of the constraint. We tend to overlook those in Economics, though.

Karush-Kuhn-Tucker Theorem

Put another way, the theorem states that $\mathcal{L}_x(x^*, \lambda^*, \mu^*) = 0$

So, it tell us that (x^*, λ^*, μ^*) solve the system

$$
\begin{gathered}
f_x + \lambda^T g_x + \mu^T h_x = 0 \\
\mu_i h^i(x) = 0,\ i = 1, \dots, l \\
g(x) = 0 \\
h(x) \leq 0 \\
\mu \leq 0
\end{gathered}
$$

- Subscripts ($_x$) denote derivatives w.r.t. x (it's a vector)
- $h^i(x)$ is the i-th element of $h(x)$

The KKT approach

The KKT theorem gives us a first approach to solving unconstrained optimization problems

- If the problem has box constraints ($a \leq x \leq b$), we can solve the corresponding *mixed complementarity problem* $CP(f', a, b)$ as we saw in unit 3
- If constraints are more elaborated and multidimensional, we need to solve a series of nonlinear systems: one for each possible combination of binding inequality constraints
  - This is probably how you learned to solve utility maximization with a budget constraint

The KKT approach

Let $\mathcal{I}$ be the set of $1, 2, \ldots, l$ inequality constraints. For a subset $\mathcal{P} \in \mathcal{I}$ of, we define the $\mathcal{P}$ problem as the nonlinear system of equations

$$
\begin{gathered}
f_x + \lambda^T g_x + \mu^T h_x = 0 \\
h^i(x) = 0,\ i \in \mathcal{P} \\
\mu_i = 0,\ i \in \mathcal{I} - \mathcal{P} \\
g(x) = 0
\end{gathered}
$$

We solve this system for every possible combination of binding constraints $\mathcal{P}$

- There might not be a solution for some combinations. That's OK
- Compare the solutions of all combinations and pick the optimal (where f attains the smallest value, in this case)

The KKT approach

When we have a good intuition about the problem, we may know ahead of time which constraints will bind

- For example, with monotonically increasing utility functions, we know the budget constraint binds

But as the number of constraints grows, we have an even larger number of possible combinations

- More combinations = more nonlinear systems to solve and compare

Other solution approaches

The combinatorial nature of the KKT approach is not that desirable from a computational perspective

- However, if the resulting nonlinear systems are simple to solve, we may still favor KKT

There are computational alternatives to KKT. We'll discuss three types of algorithms

- Penalty methods
- Active set methods
- Interior point methods

Constrained optimization algorithms

Penalty methods

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

$$\min_x f(x) \;\text{ s. t. }\; g(x) = 0$$

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

$$Q(x;\rho) = f(x) + \rho P(g(x))$$

where ρ is the penalty parameter

Penalty methods

With this, we transformed it into an unconstrained optimization problem

$$\min_x Q(x;\rho) = f(x) + \rho P(g(x))$$

How do we pick P and ρ?

A first idea is to penalize a candidate solution as much as possible whenever it leaves the feasible set: infinite penalty!

$$Q(x) = f(x) + \infty\mathbf{1}(g(x) \neq 0)$$

where $\mathbf{1}$ is an indicator function

- This is the **infinity step** method

Penalty methods

However, the infinite step method is a pretty bad idea

- Q becomes discontinuous and non-differentiable: it's very hard for algorithms to iterate near the region where the constraint binds
- Any really large value or ρ leads to the same practical problem

So we might instead use a more forgiving penalty function

Penalty methods

A widely-used choice is the quadratic penalty function

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

- For inequality constraint $h(x) \leq 0$, we can use $[\max(0, h_i(x))]^2$

The second term increases the value of the function

- bigger $\rho \to$ bigger penalty from violating the constraint

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

Penalty methods

Algorithms generally iterate on sequences of $\rho_k \to \infty$ 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 explicit estimates of Lagrange multipliers to help force binding constraints to bind precisely

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 $(-1, -1)$

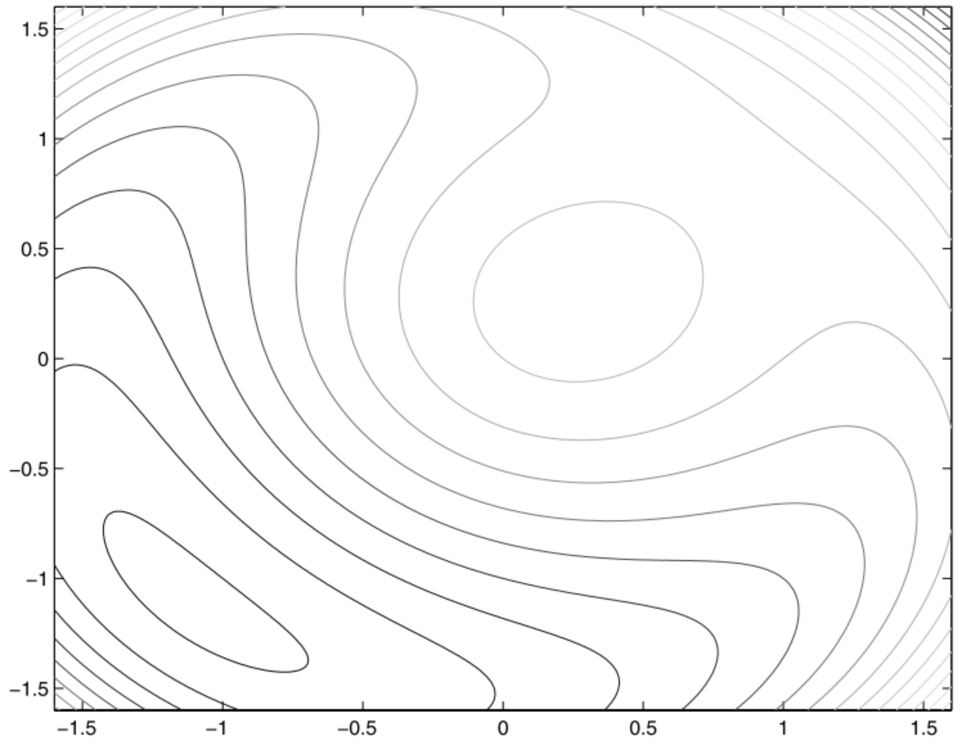
The penalty method function $Q(x_1, x_2; \rho)$ is

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

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

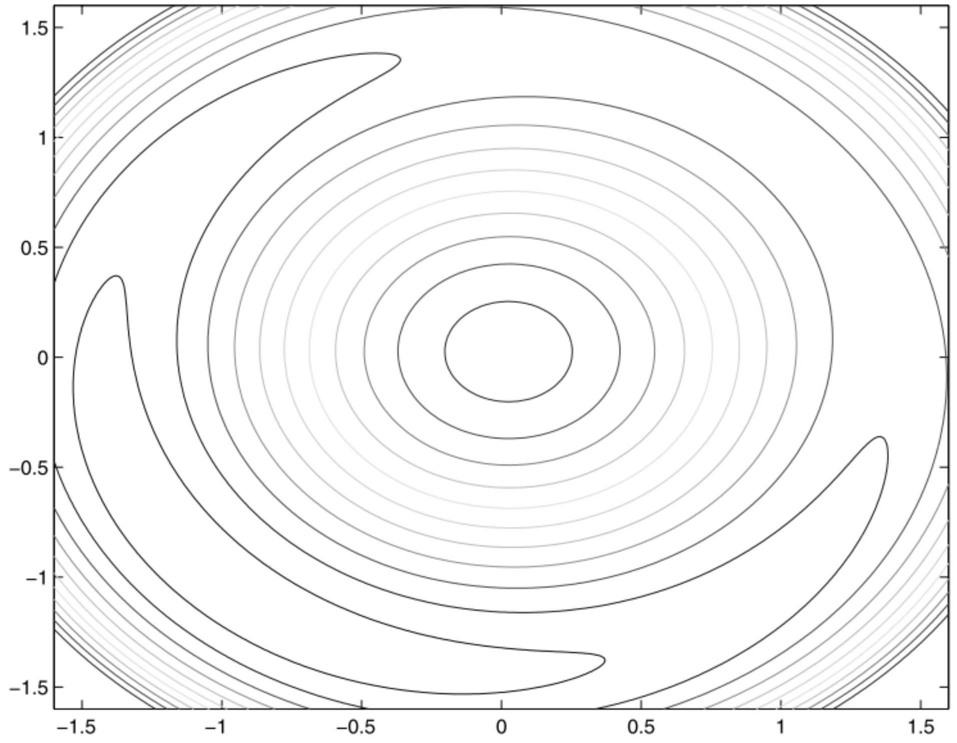
Penalty method example

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



Penalty method example

$\rho = 10$, solution is very close to $(-1, -1)$. Notice how quickly value increases outside $x_1^2 + x_2^2 = 2$ circle



Active set methods

The KKT method can lead to too many combinations of constraints to evaluate

Penalty methods don't have the same problem but still require us to evaluate every constraint, even if they are not binding

--

Improving on the KKT approach, **active set methods** strategically to pick a sequence of combinations of constraints

Active set methods

Instead of trying all possible combinations, like in KKT, active set methods start with an initial guess of the binding constraints set

Then, iterate by periodically checking constraints

- Add or keep the ones that are active (binding)
- Drop the ones that are inactive (slack)

If an appropriate strategy of picking sets is chosen, active set algorithms converge to the optimal solution

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

Interior point methods

Consider the following constrained optimization problem

$$\min_x f(x)$$
$$\text{subject to: } g(x) = 0, h(x) \leq 0$$

Reformulate this problem as

$$\min_{x,s} f(x)$$
$$\text{subject to: } g(x) = 0, h(x) + s = 0, s \geq 0$$

where s is a vector of slack variables for the constraints

Interior point methods

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

$$\min_{x,s} f(x) - \mu \sum_{i=1}^{l} log(s_i)$$
$$\text{subject to: } g(x) = 0, h(x) + s = 0$$

where $\mu > 0$ is a barrier parameter

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

- Another common barrier function is $\sum_{i=1}^{l}(1/s_i)$

Interior point methods solve a sequence of barrier problems until μ_k converges to zero

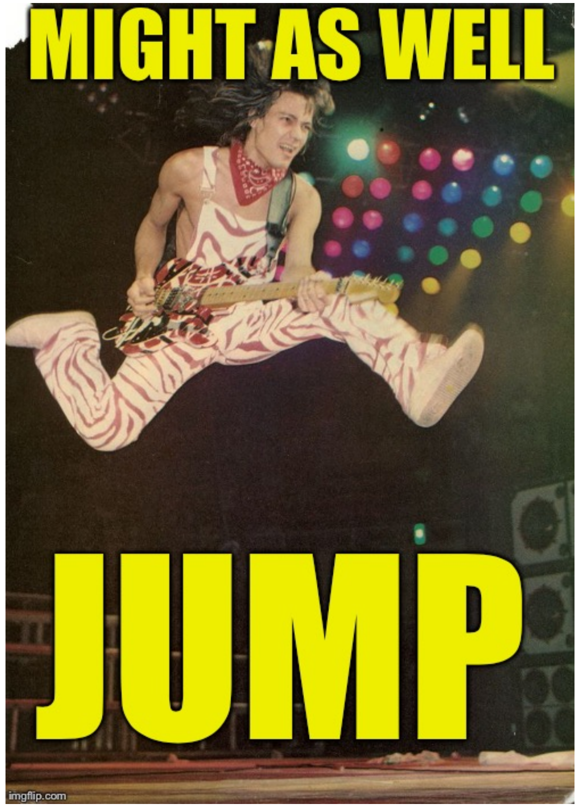
The solution to the barrier problem converges to that of the original problem

Constrained optimization in Julia

Constrained optimization in Julia

We are going to cover a cool package called `JuMP.jl`

- It offers a whole modeling language inside Julia
- You define your model and plug it into one of the many solvers available
- It's like GAMS and AMPL... *but FREE and with a full-fledged programming language around it*



Constrained optimization in Julia

Most solvers can be accessed directly in their own packages

- Like we did to use `Optim.jl`
- These packages are usually just a Julia interface for a solver programmed in another language

But `JuMP` gives us a unified way of specifying our models and switching between solvers

`JuMP` specifically designed for constrained optimization but works with unconstrained too

- With more overhead relative to using `Optim` or `NLopt` directly

Getting stated with JuMP

There are 5 key steps:

1) Initialize your model and solver:

- `mymodel = Model(SomeOptimizer)`

2) Declare variables (adding any box constraints)

- `@variable(mymodel, x >= 0)`

3) Declare the objective function

- If linear: `@objective(mymodel, Min, 12x + 20y)`
- If nonlinear: `@NLobjective(mymodel, Min, 12x^0.7 + 20y^2)`

Getting stated with JuMP

4) Declare constraints

- If linear: `@constraint(mymodel, c1, 6x + 8y >= 100)`
- If nonlinear: `@NLconstraint(mymodel, c1, 6x^2 - 2y >= 100)`

5) Solve it

- `optimize!(mymodel)`
- Note the `!`, so we are modifying `mymodel` and saving results in this object

Follow along!

Let's use `JuMP` to solve the illustrative problem from the first slides

We will use solver `Ipopt`, which stands for *Interior Point Optimizer*. It's a free solver we can access through package `Ipopt.jl`

```julia
using JuMP, Ipopt;
```

Follow along: function definition

Define the function:

$$\min_{x} -exp\left(-(x_1 x_2 - 1.5)^2 - (x_2 - 1.5)^2\right)$$

```
f(x1,x2) = -exp.(-(x1.*x2 - 3/2).^2 - (x2-3/2).^2);
```

Follow along: initialize model

Initialize the model for `Ipopt`

```
model = Model(Ipopt.Optimizer)
```

```
## A JuMP Model
## Feasibility problem with:
## Variables: 0
## Model mode: AUTOMATIC
## CachingOptimizer state: EMPTY_OPTIMIZER
## Solver name: Ipopt
```

You can set optimzer parameters like this

- There are TONS of parameters you can adjust (see the manual)

```
# This is relative tol. Default is 1e-8
set_optimizer_attribute(model, "tol", 1e-6)
```

Follow along: declare variables

We will focus on non-negative values

```
@variable(model, x1 >=0)
```

```
## x1
```

```
@variable(model, x2 >=0)
```

```
## x2
```

- You could type `@variable(model, x1)` to declare a x_1 as a free variable

Follow along: declare objective

We will focus on non-negative values

```
@NLobjective(model, Min, f(x1, x2))
```

`JuMP` will use autodiff (with `ForwardDiff` package) by default. If you want to use your define gradient and Hessian, you need to "register" the function like this

```
register(model, :my_f, n, f, grad, hessian)
```

- `:my_f` is the name you want to use inside `model`, `n` is the number of variables `f` takes, and `grad` `hessian` are user-defined functions

Follow along: solving the model

First, let's solve the (mostly) unconstrained problem

- Not really unconstrained because we defined non-negative `x1` and `x2`

Checking our model

```
print(model)
```

```
## Min f(x1, x2)
## Subject to
##  x1 >= 0.0
##  x2 >= 0.0
```

Follow along: solving the model

```
optimize!(model)
```

```
##
## ******************************************************************************
## This program contains Ipopt, a library for large-scale nonlinear optimization.
##  Ipopt is released as open source code under the Eclipse Public License (EPL).
##          For more information visit https://github.com/coin-or/Ipopt
## ******************************************************************************
##
## This is Ipopt version 3.14.4, running with linear solver MUMPS 5.4.1.
##
## Number of nonzeros in equality constraint Jacobian...:        0
## Number of nonzeros in inequality constraint Jacobian.:        0
## Number of nonzeros in Lagrangian Hessian.............:        0
##
## Total number of variables............................:        2
##                      variables with only lower bounds:        2
##                 variables with lower and upper bounds:        0
##                      variables with only upper bounds:        0
## Total number of equality constraints.................:        0
## Total number of inequality constraints...............:        0
##         inequality constraints with only lower bounds:        0
```

Follow along: solving the model

The return message is rather long and contains many details about the execution. You can turn this message off with

```
set_silent(model);
```

We can check minimizers with

```
unc_x1 = value(x1)
```

```
## 1.0000000326687912
```

```
unc_x2 = value(x2)
```

```
## 1.4999999423446968
```

```
unc_obj = objective_value(model)
```

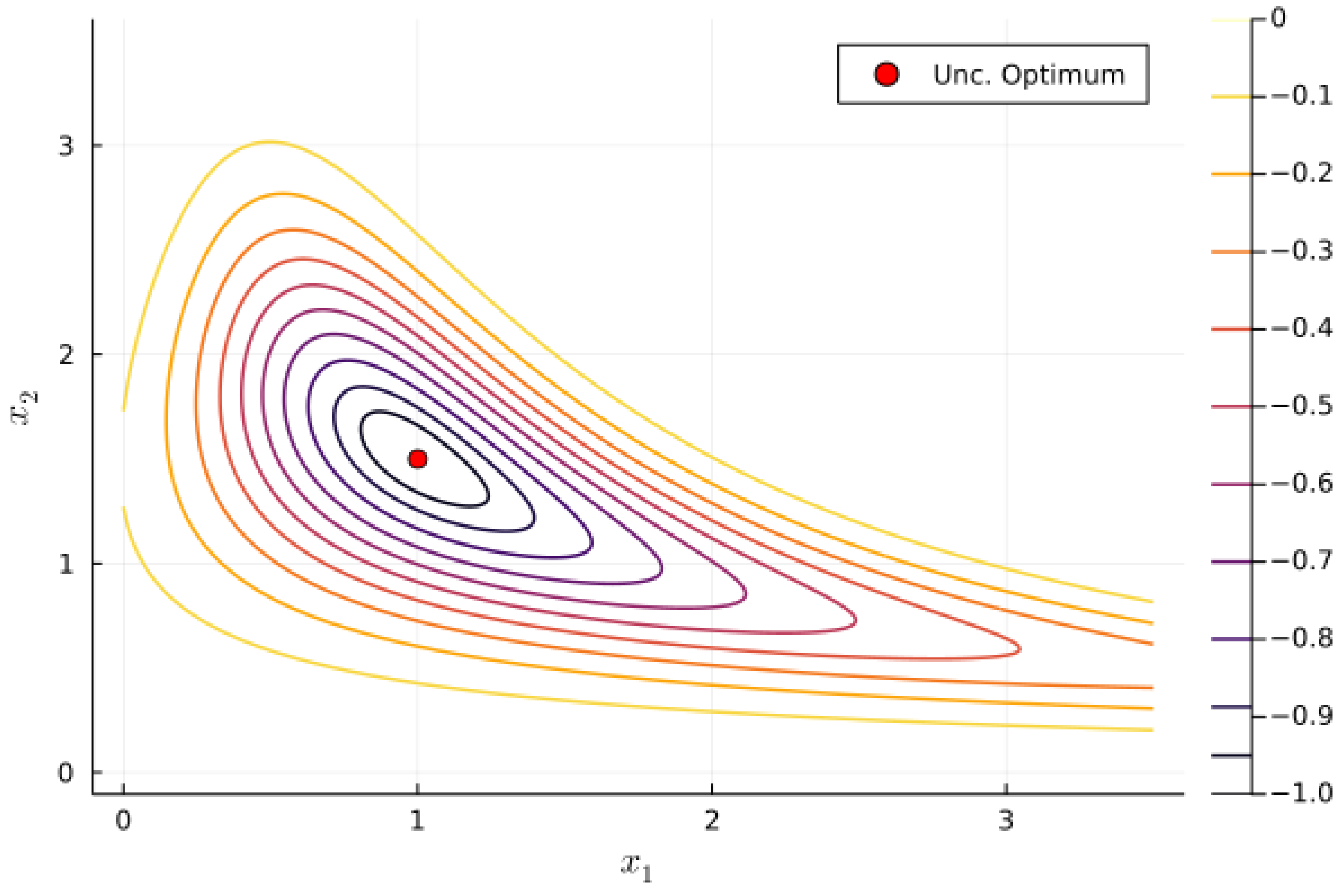
Follow along: solving the model

And the minimum with

```
unc_obj = objective_value(model)
```

```
## -0.9999999999999966
```

Follow along: solving the model



Follow along: declaring constraints

Let's add a nonlinear equality constraint $-x_1 + x^2 = 0$ and re-solve the model

```
@NLconstraint(model, -x1 +x2^2 == 0)
```

```
## (-x1 + x2 ^ 2.0) - 0.0 == 0
```

```
print(model)
```

```
## Min f(x1, x2)
## Subject to
##  x1 >= 0.0
##  x2 >= 0.0
##  (-x1 + x2 ^ 2.0) - 0.0 == 0
```

```
optimize!(model)
```

Follow along: solving the equality constrained model

```
eqcon_x1 = value(x1)
```

```
## 1.3578043097074932
```

```
eqcon_x2 = value(x2)
```

```
## 1.1652486042512478
```

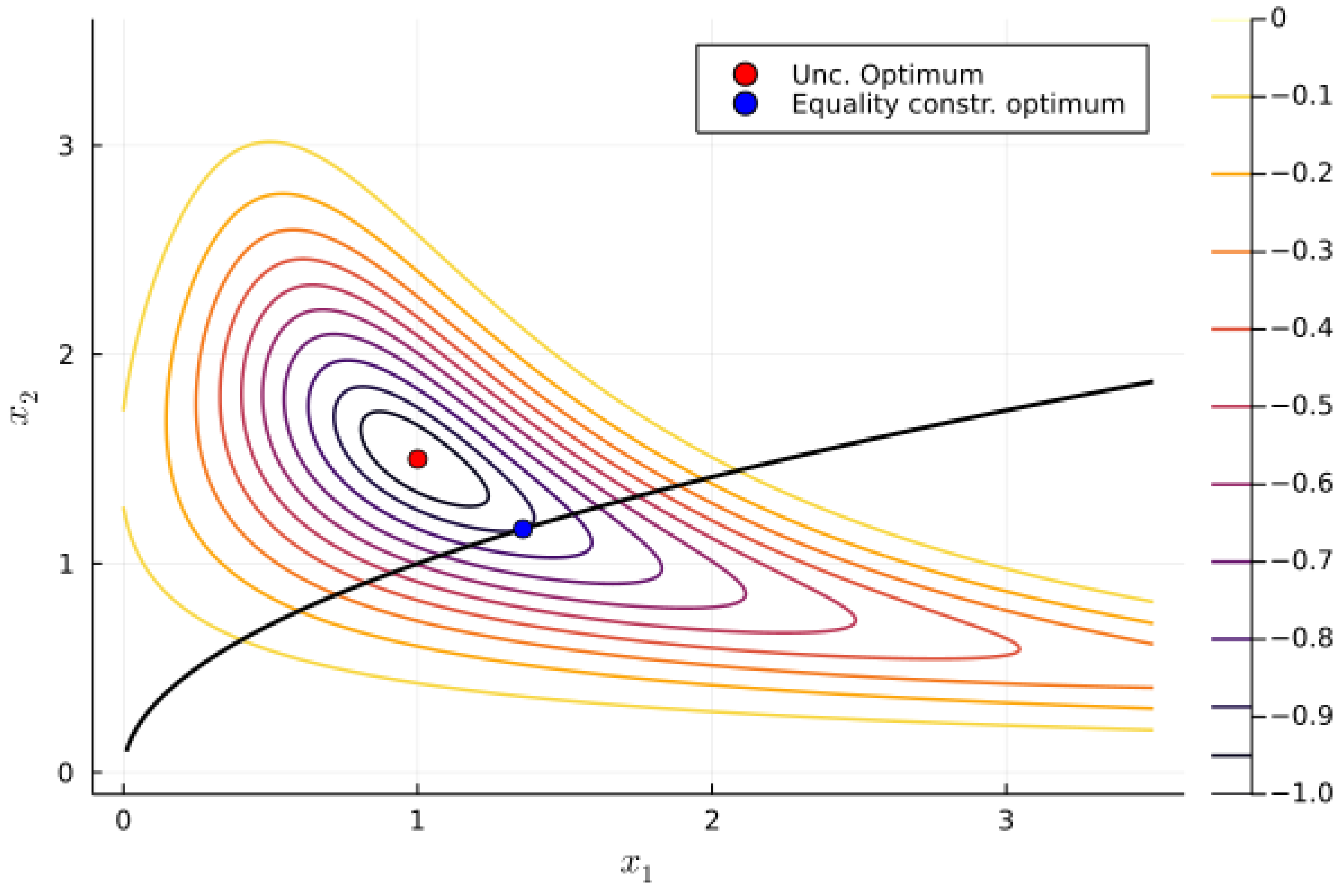
```
value(-x1 + x2^2) # We can evaluate expressions too
```

```
## 1.9877433032888803e-12
```

```
eqcon_obj = objective_value(model)
```

```
## -0.887974742266783
```

Follow along: solving the equality constrained model



Solving the inequality constrained model

I now initialize a new model with inequality constraint $-x_1 + x^2 \leq 0$

```
model2 = Model(Ipopt.Optimizer);
@variable(model2, x1 >=0);
@variable(model2, x2 >=0);
@NLobjective(model2, Min, f(x1, x2));
@NLconstraint(model2, -x1 + x2^2 <= 0);
optimize!(model2);
```

Solving the inequality constrained model

```
ineqcon_x1 = value(x1)
```

```
## 1.357804311747407
```

```
ineqcon_x2 = value(x2)
```

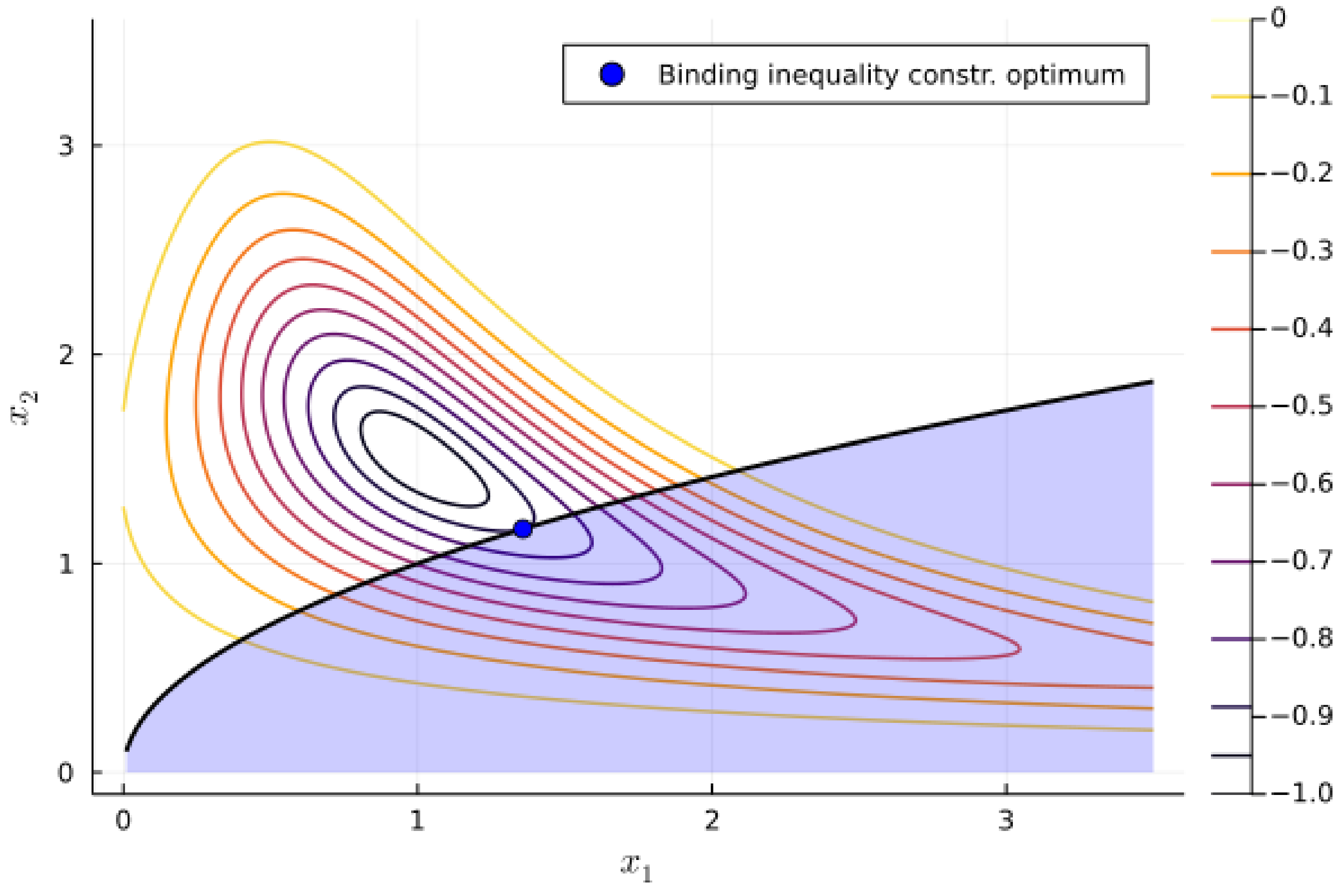
```
## 1.165248609391406
```

```
ineqcon_obj = objective_value(model2)
```

```
## -0.887974743957088
```

Same results as in the equality constraint: the constraint is binding

Solving the inequality constrained model



Relaxing the inequality constraint

What if instead we use inequality constraint $-x_1 + x^2 \leq 1.5$?

```
model3 = Model(Ipopt.Optimizer);
@variable(model3, x1 >=0);
@variable(model3, x2 >=0);
@NLobjective(model3, Min, f(x1, x2));
@NLconstraint(model3, c1, -x1 + x2^2 <= 1.5);
optimize!(model3);
```

Relaxing the inequality constraint

```
ineqcon2_obj = objective_value(model3)
```

```
## -1.0
```

```
ineqcon2_x1 = value(x1)
```

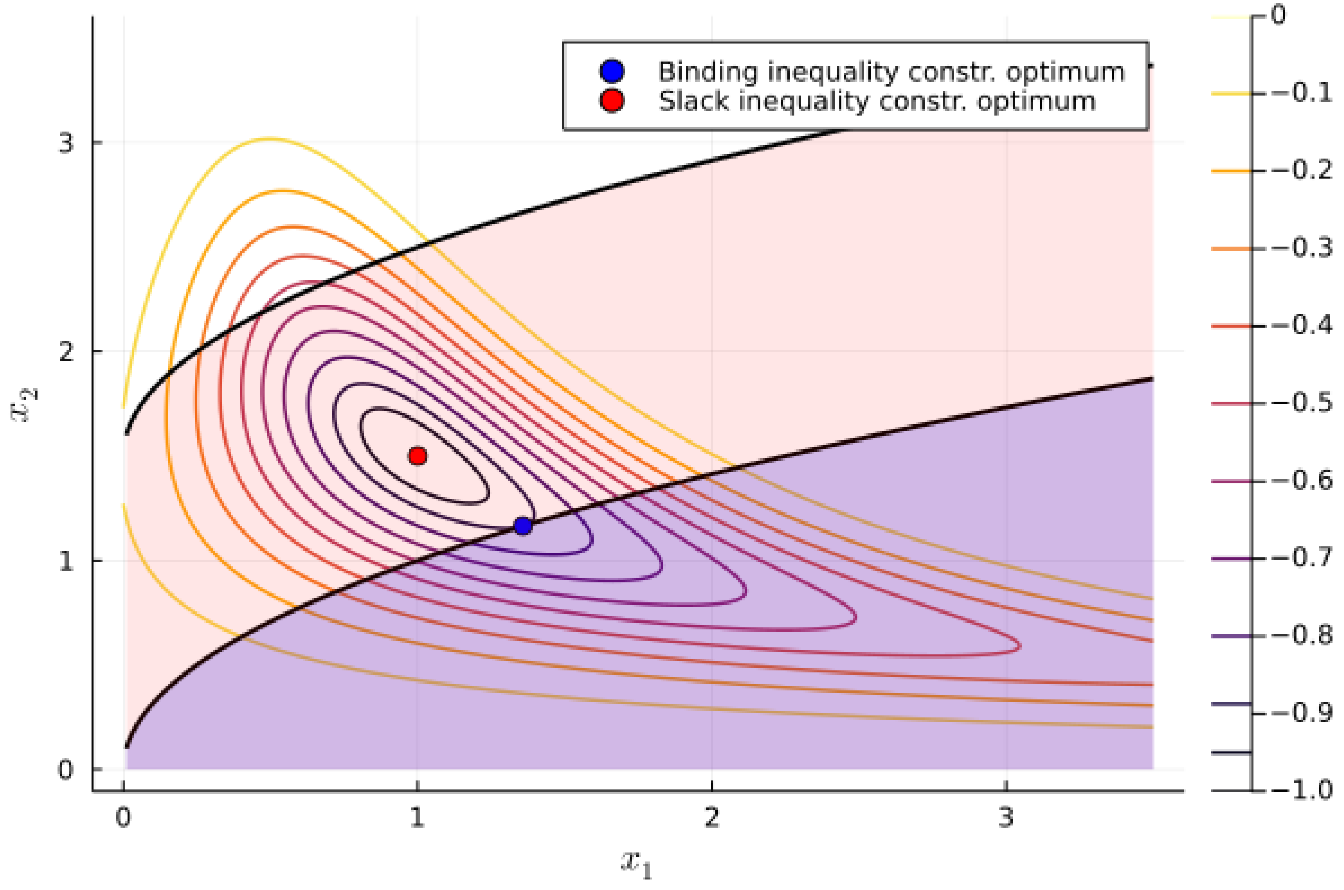
```
## 1.0000000035503052
```

```
ineqcon2_x2 = value(x2)
```

```
## 1.4999999931258383
```

We get the same results as in the unconstrained case

Solving the inequality constrained model



Practical advice for numerical optimization

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 return codes

Return codes (or exit flags) tell you why the solver stopped

- There are all sorts of reasons why a solver ends execution
- Each solver has its own way of reporting errors
- In `JuMP` you can use `@show termination_status(mymodel)`

READ THE SOLVER DOCUMENTATION!

Use trace options to get a sense of what went wrong

- Did guesses grow unexpectedly?
- Did a gradient-based operation fail? (E.g., division by zero)

Check return codes

Examples from Ipopt.jl documentation

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

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

This happens when things change at different rates:

- Investment rates are between 0 and 1
- Consumption can be in trillions of 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

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

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

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

If you are using simulation-based estimators or estimators that depend on successive optimizations, be even more conservative *because errors compound*

Be aware of tolerances

May be a substantial trade-off 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

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