optimization

March 24, 2017

1 Computational Economics: Unconstrained Optimization

Florian Oswald Sciences Po, 2017

1.1 Some Taxonomy and Initial Examples

- In most of the examples to follow, we talk about *minimization* of a function f. Everything we do also applies to maximization, since $\min_x f(x) = \max_x -f(x)$.
- Here is a generic optimization problem:

$$\min_{x \in \mathbb{R}^n} f(x) \text{ s.t. } \begin{array}{ll} c_i(x) = 0, & i \in E \\ c_i(x) \geq 0, & i \in I \end{array}$$

- This is a general way of writing an optimization problem. E are all indices as equality constraints, I are all inequality constraints.
- An example of such a problem might be

$$\min(x_1 - 2)^2 + (x_2 - 1)^2 \text{ s.t. } \begin{array}{l} x_1^2 - x_2 \leq 0 \\ x_1 + x_2 \leq 2 \end{array}$$

- Here is a picture of that problem taken from the textbook [@nocedal-wright] (for copyright reasons, I cannot show this in the online version of the slides.):

1.2 Kinds of problems considered

- Don't talk about stochastic optimization methods:
  - Simluated Annealing
  - MCMC
  - other Stochastic Search Methods
  - A gentle introduction is [@casella-R]

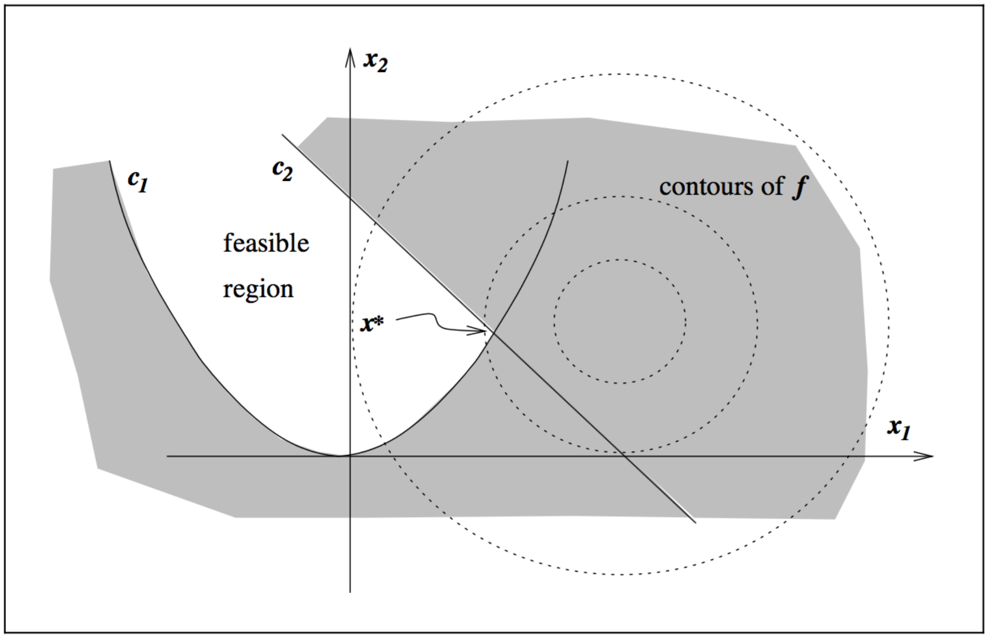


Figure 1.1 in [@nocedal-wright]

1.3 Transportation Problem

A chemical company has two factories F_1, F_2 and a dozen retail outlets $R_1, \ldots, R_{12}$. Each factory i can produce at most a_i tons of output each week. Each retail outlet j has a weekly demand of b_j tons per week. The cost of shipping from F_i to R_j is given by c_{ij}. How much of the product to ship from each factory to each outlet, minimize cost, and satisfy all constraints? let's call x_{ij} the number of tons shipped from i to j.

- A mathematical formulation of this problem is

$$
\begin{aligned}
\min \sum_{ij} c_{ij} x_{ij} \\
\text{subject to} \sum_{j=1}^{12} x_{ij} \leq a_i, \quad i = 1, 2 \\
\sum_{i=1}^{2} x_{ij} \geq b_j, \quad j = 1, \ldots, 12 \\
x_{ij} \geq 0, \quad i = 1, 2, j = 1, \ldots, 12
\end{aligned}
$$

- This is called a *linear programming* problem, because both objective function and all constrains are linear.
- With any of those being nonlinear, we would call this a non-linear problem.

1.4 Constrained vs Unconstrained

- There are many applications of both in economics.

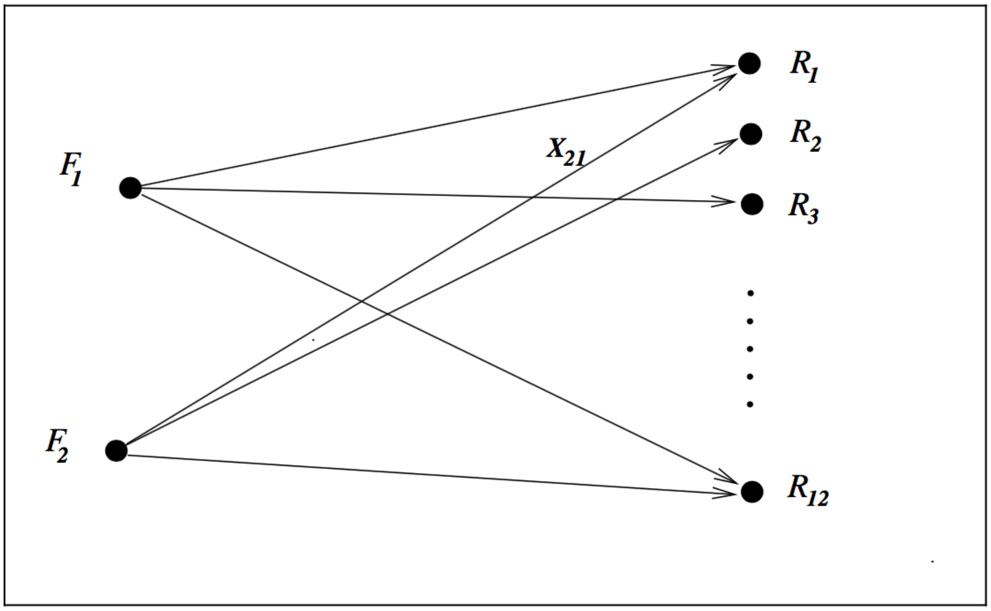


Figure 1.2 in [@nocedal-wright]

- Unconstrained: maximimum likelihood
- Constrained: MPEC
- It is sometimes possible to transform a constrained problem into an unconstrained one.

1.5 Convexity

- Convex problems are easier to solve.
- What is convex?

  A set $S \in \mathbb{R}^n$ is convex if the straight line segment connecting any two points in S lies entirely inside S. A function f is a convex function, if its domain S is a convex set, and for any two points $x, y \in S$, we have that

  $$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y)$$

  for all $\alpha \in [0,1]$

- Simple instances of convex sets are the unit ball $\{y \in \mathbb{R}^n, \|y\|_2 \leq 1\}$, and any set defined by linear equalities and inequalities.
- *convex Programming* describes a special case of the introductory minimizatin problem where
  - the objective function is convex,
  - the equality constrains are linear, and
  - the inequality constraints are concave.

1.6 Optimization Algorithms

- All of the algorithms we are going to see employ some kind of *iterative* proceedure.
- They try to improve the value of the objective function over successive steps.

- The way the algorithm goes about generating the next step is what distinguishes algorithms from one another.
  - Some algos only use the objective function
  - Some use both objective and gradients
  - Some add the Hessian
  - and many variants more

1.7 Desirable Features of any Algorithm

- Robustness: We want good performance on a wide variety of problems in their class, and starting from *all* reasonable starting points.
- Efficiency: They should be fast and not use an excessive amount of memory.
- Accuracy: They should identify the solution with high precision.

1.8 A Word of Caution

- You should **not** normally attempt to write a numerical optimizer for yourself.
- Entire generations of Applied Mathematicians and other numerical pro's have worked on those topics before you, so you should use their work.
  - Any optimizer you could come up with is probably going to perform below par, and be highly likely to contain mistakes.
  - Don't reinvent the wheel.
- That said, it's very important that we understand some basics about the main algorithms, because your task is **to choose from the wide array of available ones**.

2 Unconstrained Optimization: What is a solution?

- A typical unconstrained optimization problem will look something like this:

$$\min_x f(x), \quad x \in \mathbb{R}^n$$

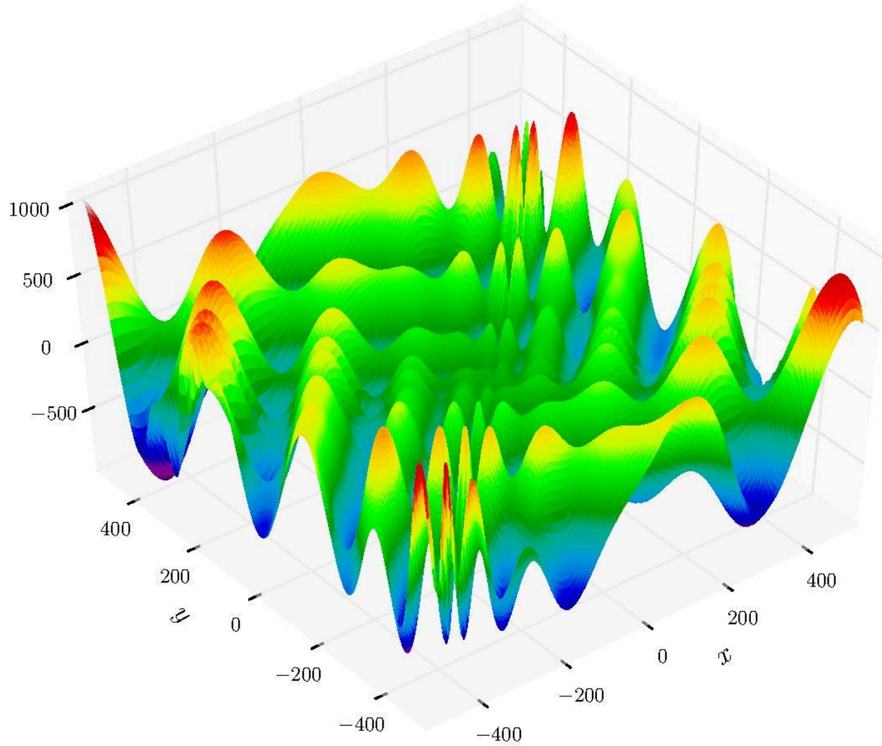
  and where $f : \mathbb{R}^n \mapsto \mathbb{R}$ is a smooth function.
- In general, we would always like to find a *global* minimizer, i.e. a point

$$x^* \text{ where } f(x^*) \leq f(x) \quad \forall x$$

- Since our algorithm is not going to visit many points in $\mathbb{R}^n$ (or so we hope), we can never be totally sure that we find a global optimizer.
- Most optimizers can only find a *local* minimizer. That is a point

$$x^* \text{ where } f(x^*) \leq f(x) \quad \forall x \in \mathcal{N}$$

  where $\mathcal{N}$ is a neighborhood around x^*.



Global min at $f(512, 404.2319)$. By Gaortizg GFDL or CC BY-SA 3.0, via Wikimedia Commons

2.1 Global minization can be very hard sometimes.

2.2 (Unconstrained) Optimization in `Julia`

- Umbrella Organisation: `http://www.juliaopt.org`
  - We will make ample use of this when we talk about constrained optimsation.
  - The Julia Interface to the very well established C-Library NLopt is called `NLopt.jl`. One could use `NLopt` without constraints in an unconstrained problem.
- `Roots.jl`: Simple algorithms that find the zeros of a univariate function.
- Baseline Collection of unconstrained optimization algorithms: `Optim.jl`

2.3 Introducing `Optim.jl`

- Multipurpose unconstrained optimization package
  - provides 8 different algorithms with/without derivatives
  - univariate optimization without derivatives

2.4 The Golden Ratio or Bracketing Search for 1D problems

- A derivative-free method
- a Bracketing method
  - find the local minimum of f on $[a, b]$
  - select 2 interior points c, d such that $a < c < d < b$

    * $f(c) \leq f(d) \implies$ min must lie in $[a,d]$. replace b with d, start again with $[a,d]$
    * $f(c) > f(d) \implies$ min must lie in $[c,b]$. replace a with c, start again with $[c,b]$
  - how to choose b,d though?
  - we want the length of the interval to be independent of whether we replace upper or lower bound
  - we want to reuse the non-replaced point from the previous iteration.
  - these imply the golden rule:
  - new point $x_i = a + \alpha_i(b-a)$, where $\alpha_1 = \frac{3-\sqrt{5}}{2}, \alpha_2 = \frac{\sqrt{5}-1}{2}$
  - α_2 is known as the *golden ratio*, well known for it's role in renaissance art.

2.4.1 Bracketing Search in Julia

- The package `Optim.jl` provides an implementation of "Brent's Method" as well as the golden section search:

```
In [9]: using Plots
        using Optim
        f(x) = exp(x) - x^4
        minf(x) = -f(x)
        brent = optimize(minf,0,2,Brent())
        golden = optimize(minf,0,2,GoldenSection())
        println("brent = $brent")
        println("golden = $golden")
        plot(f,0,2)
```

```
brent = Results of Optimization Algorithm
 * Algorithm: Brent's Method
 * Search Interval: [0.000000, 2.000000]
 * Minimizer: 8.310315e-01
 * Minimum: -1.818739e+00
 * Iterations: 12
 * Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08*|x|+2.2e-16): true
 * Objective Function Calls: 13


WARNING: Method definition f(Any) in module Main at In[1]:2 overwritten at In[9]:3.
WARNING: Method definition minf(Any) in module Main at In[2]:3 overwritten at In[9]:4.


golden = Results of Optimization Algorithm
 * Algorithm: Golden Section Search
 * Search Interval: [0.000000, 2.000000]
 * Minimizer: 8.310315e-01
 * Minimum: -1.818739e+00
 * Iterations: 37
 * Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08*|x|+2.2e-16): true
 * Objective Function Calls: 38
```

```julia
In [2]: # how well does this do with many local minima?
        fun(x) = exp(x) - x^4 +sin(40*x)
        minf(x) = -fun(x)
        grid = collect(0:0.0001:2);
        v,idx  = findmax(Float64[fun(x) for x in grid])
        println("grid maximizer is $(grid[idx])")
        golden = optimize(minf,0,2,GoldenSection())
        brent = optimize(minf,0,2,Brent())
        using Base.Test
        println("brent minimizer = $(brent.minimizer)")
        println("golden minimizer = $(golden.minimizer)")
        plot(fun,0,2)
```

```
grid maximizer is 0.8247


WARNING: Method definition minf(Any) in module Main at In[1]:3 overwritten at In[2]:3.



        UndefVarError: optimize not defined
```

2.5 Beyond One Dimension

2.5.1 Introducing Rosenbrock's Banana function

The Banana function is defined by

$$f(x,y) = (a-x)^2 + b(y-x^2)^2$$

2.5.2 What is the minimum of that function?

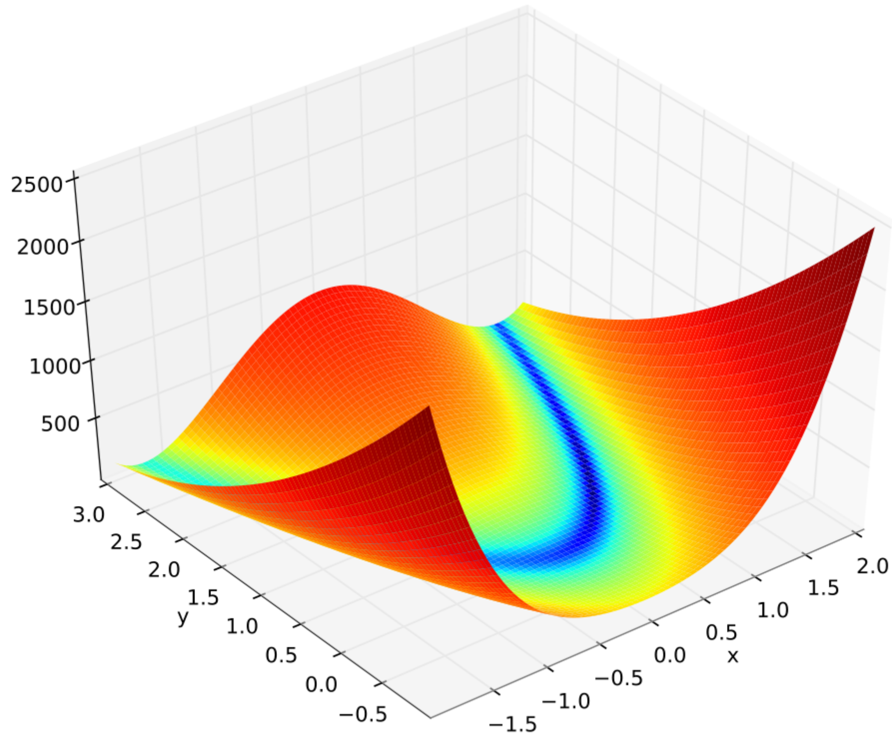
- For $a = 1, b = 100$, what is the global minimum of that function?
- What are the inputs one needs to supply to an algorithm in a more general example?

2.6 Rosenbrock Banana and Optim.jl

- We will use `Optim` for the rest of this lecture.
- We need to supply the objective function and - depending on the solution algorithm - the gradient and hessian as well.

```julia
In [3]: using Optim
        rosenbrock = Optim.UnconstrainedProblems.examples["Rosenbrock"]

        # contains:
        # function rosenbrock(x::Vector)
```



Banana for $a = 0$. By Gaortizg GFDL or CC BY-SA 3.0, via Wikimedia Commons

```
#     return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
# end

# function rosenbrock_gradient!(x::Vector, storage::Vector)
#     storage[1] = -2.0 * (1.0 - x[1]) - 400.0 * (x[2] - x[1]^2) * x[1]
#     storage[2] = 200.0 * (x[2] - x[1]^2)
# end

# function rosenbrock_hessian!(x::Vector, storage::Matrix)
#     storage[1, 1] = 2.0 - 400.0 * x[2] + 1200.0 * x[1]^2
#     storage[1, 2] = -400.0 * x[1]
#     storage[2, 1] = -400.0 * x[1]
#     storage[2, 2] = 200.0
# end

# there are many other examples on Optim.UnconstrainedProblems
```

```
Out[3]: Optim.UnconstrainedProblems.OptimizationProblem("Rosenbrock",Optim.UnconstrainedProblems
```

2.7 Comparison Methods

- We will now look at a first class of algorithms, which are very simple, but sometimes a good starting point.
- They just *compare* function values.

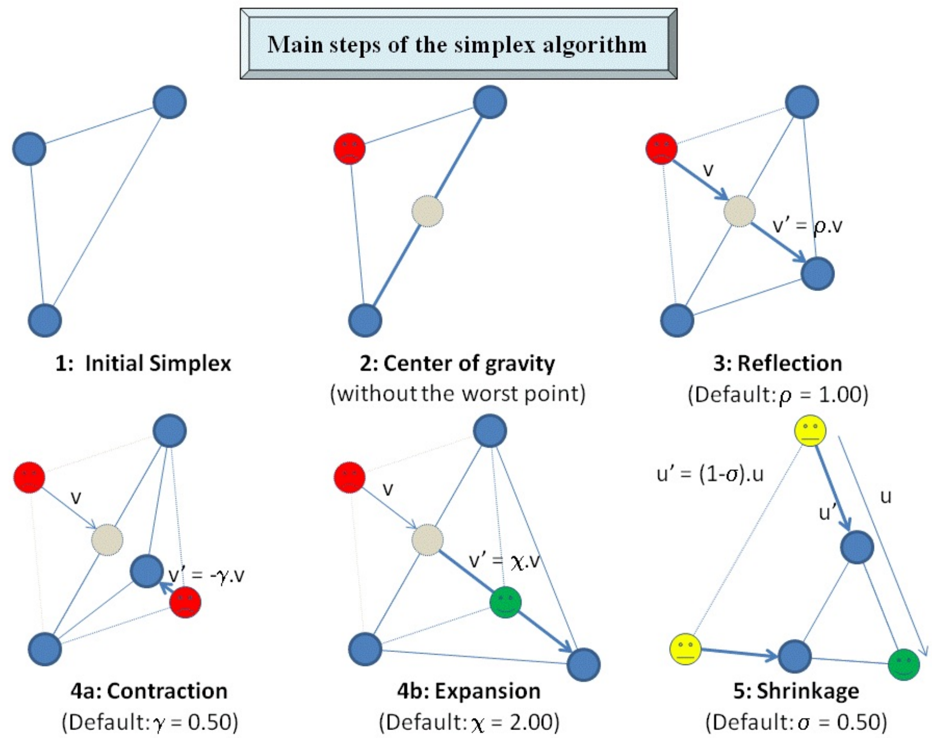
- *Grid Search* : Compute the objective function at $G = \{x_1, \ldots, x_N\}$ and pick the highest value of f.
  - This is very slow.
  - It requires large N.
  - But it's robust (will find global optimizer for large enough N)

```
In [4]: # grid search on rosenbrock
        grid = collect(-1.0:0.1:3);
        grid2D = [[i;j] for i in grid,j in grid];
        val2D = map(rosenbrock.f,grid2D);
        r = findmin(val2D);
        println("grid search results in minimizer = $(grid2D[r[2]])")
```

```
grid search results in minimizer = [1.0,1.0]
```

2.8 Bracketing for Multidimensional Problems: Nelder-Mead

- The Goal here is to find the simplex containing the local minimizer x^*
- In the case where f is n-D, this simplex has $n+1$ vertices
- In the case where f is 2-D, this simplex has $2+1$ vertices, i.e. it's a triangle.
- The method proceeds by evaluating the function at all $n+1$ vertices, and by replacing the worst function value with a new guess.
- this can be achieved by a sequence of moves:
  - reflect
  - expand
  - contract
  - shrink movements.



- this is a very popular method. The matlab functions `fmincon` and `fminsearch` implements it.
- When it works, it works quite fast.
- No derivatives required.

```
In [5]: optimize(rosenbrock, [0.0, 0.0], NelderMead())
```

```
Out[5]: Results of Optimization Algorithm
         * Algorithm: Nelder-Mead
         * Starting Point: [0.0,0.0]
         * Minimizer: [0.9999710322210338,0.9999438685860869]
         * Minimum: 1.164323e-09
         * Iterations: 74
         * Convergence: true
           *  ((y-)š)/n < 1.0e-08: true
           * Reached Maximum Number of Iterations: false
         * Objective Function Calls: 108
```

- But.

2.9 Bracketing for Multidimensional Problems: Comment on Nelder-Mead

Lagarias et al. (SIOPT, 1999): At present there is no function in any dimension greater than one, for which the original Nelder-Mead algorithm has been proved to converge to a minimizer.

Given all the known inefficiencies and failures of the Nelder-Mead algorithm [...], one might wonder why it is used at all, let alone why it is so extraordinarily popular.