SciencesPo Computational Economics
Spring 2017

Florian Oswald

March 15, 2018

1 Optimization 2: Algorithms and Constraints

Florian Oswald Sciences Po, 2018

1.1 Bracketing

- A derivative-free method for *univariate* f
- works only on **unimodal** f
- (Draw choosing initial points and where to move next)

1.2 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.

In [1]:
```
using Plots
using Optim
plotlyjs()
f(x) = exp(x) - x^4
minf(x) = -f(x)
brent = optimize(minf,0,2,Brent())
golden = optimize(minf,0,2,GoldenSection())
plot(f,0,2)
```

```
        gui()
        println("brent = $brent")
        println("golden = $golden")
```

```
WARNING: Method definition midpoints(Base.Range{T} where T) in module Base at deprecated.jl:56 o
WARNING: Method definition midpoints(AbstractArray{T, 1} where T) in module Base at deprecated.j


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
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
```

1.2.1 Bisection Methods

- Root finding

1.3 Rosenbrock Banana and Optim.jl

- We can supply the objective function and - depending on the solution algorithm - the gradient and hessian as well.

```
In [2]: using Optim
        rosenbrock = Optim.UnconstrainedProblems.examples["Rosenbrock"]

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

```
UndefVarError: UnconstrainedProblems not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.4 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 [3]:
```
# 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]])")
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.5 Local Descent Methods

- Applicable to multivariate problems

- We are searching for a *local model* that provides some guidance in a certain region of f over **where to go to next**.
- Gradient and Hessian are informative about this.

1.5.1 Local Descent Outline

All descent methods follow more or less this structure. At iteration k,

1. Check if candidate $\mathbf{x}^{(k)}$ satisfies stopping criterion:
    - if yes: stop
    - if no: continue
2. Get the local *descent direction* $\mathbf{d}^{(k)}$, using gradient, hessian, or both.
3. Set the *step size*, i.e. the length of the next step, α^k
4. Get the next candidate via
$$\mathbf{x}^{(k+1)} \longleftarrow \alpha^k \mathbf{d}^{(k)}$$

1.5.2 The Line Search Strategy

- An algorithm from the line search class chooses a direction $\mathbf{d}^{(k)} \in \mathbb{R}^n$ and searches along that direction starting from the current iterate $x_k \in \mathbb{R}^n$ for a new iterate $x_{k+1} \in \mathbb{R}^n$ with a lower function value.
- After deciding on a direction $\mathbf{d}^{(k)}$, one needs to decide the *step length* α to travel by solving
$$\min_{\alpha>0} f(x_k + \alpha \mathbf{d}^{(k)})$$
- In practice, solving this exactly is too costly, so algos usually generate a sequence of trial values α and pick the one with the lowest f.

In [4]:
```
# https://github.com/JuliaNLSolvers/LineSearches.jl
using LineSearches
prob = Optim.UnconstrainedProblems.examples["Rosenbrock"]

algo_hz = Newton(linesearch = HagerZhang())
res_hz = Optim.optimize(prob.f, prob.g!, prob.h!, prob.initial_x, method=algo_hz)
```

```
UndefVarError: UnconstrainedProblems not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.5.3 The Trust Region Strategy

- First choose max step size, then the direction
- Finds the next step $\mathbf{x}^{(k+1)}$ by minimizing a model of $\hat{f}$ over a *trust region*, centered on $\mathbf{x}^{(k)}$
  - 2nd order Tayloer approx of f is common.
- Radius δ of trust region is changed based on how well $\hat{f}$ fits f in trust region.
- Get $\mathbf{x}'$ via

$$\begin{aligned} \min_{\mathbf{x}'} \quad & \hat{f}(\mathbf{x}') \\ \text{subject to} \quad & \|\mathbf{x} - \mathbf{x}' \leq \delta\| \end{aligned}$$

In [5]:
```
# Optim.jl has a TrustRegion for Newton (see below for Newton's Method)
NewtonTrustRegion(; initial_delta = 1.0, # The starting trust region radius
                    delta_hat = 100.0, # The largest allowable trust region radius
                    eta = 0.1, #When rho is at least eta, accept the step.
                    rho_lower = 0.25, # When rho is less than rho_lower, shrink the trus
                    rho_upper = 0.75) # When rho is greater than rho_upper, grow the tru
res = Optim.optimize(prob.f, prob.g!, prob.h!, prob.initial_x, method=NewtonTrustRegion(
```

```
UndefVarError: prob not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.5.4 Stopping criteria

1. maximum number of iterations reached
2. absolute improvement $|f(x) - f(x')| \leq \epsilon$
3. relative improvement $|f(x) - f(x')|/|f(x)| \leq \epsilon$
4. Gradient close to zero $|g(x)| \approx 0$

1.5.5 Gradient Descent

- Here we define

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{d}^{(k)})$$

- And our descent becomes

$$\mathbf{d}^{(k)} = -\nabla \frac{\mathbf{g}^{(k)}}{\|\mathbf{g}^{(k)}\|}$$

- Minimizing wrt step size results in a jagged path (each direction is orthogonal to previous direction!)

$$\alpha^{(k)} = \arg\min \alpha f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})$$

- *Conjugate* Gradient avoids this issue.

In [6]:
```julia
# Optim.jl again
GradientDescent(; alphaguess = LineSearches.InitialPrevious(),
                  linesearch = LineSearches.HagerZhang(),
                  P = nothing,
                  precondprep = (P, x) -> nothing)
```

Out[6]:
```
Optim.GradientDescent{LineSearches.InitialPrevious{Float64},LineSearches.HagerZhang{Floa
  alpha: Float64 1.0
  alphamin: Float64 0.0
  alphamax: Float64 Inf
, LineSearches.HagerZhang{Float64}
  delta: Float64 0.1
  sigma: Float64 0.9
  alphamax: Float64 Inf
  rho: Float64 5.0
  epsilon: Float64 1.0e-6
  gamma: Float64 0.66
  linesearchmax: Int64 50
  psi3: Float64 0.1
  display: Int64 0
, nothing, #3, Optim.Flat())
```

In [7]:
```julia
# there is a dedicated LineSearch package: https://github.com/JuliaNLSolvers/LineSearche
GD = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent())
GD1 = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent(),Optim.Options(it
GD2 = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent(),Optim.Options(it

println("gradient descent = $GD")
println("\n")
println("gradient descent 2 = $GD1")
println("\n")
println("gradient descent 3 = $GD2")
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.6 Second Order Methods

1.6.1 Newton's Method

- We start with a 2nd order Taylor approx over x at step k:

$$q(x) = f(x^{(k)}) + (x - x^{(k)})f'(x^{(k)}) + \frac{(x - x^{(k)})^2}{2} f''(x^{(k)})$$

- We set find it's root and rearrange to find the next step $k+1$:

$$\frac{\partial q(x)}{\partial x} = f'(x^{(k)}) + (x - x^{(k)}) f''(x^{(k)}) = 0$$
$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})}$$

- The same argument works for multidimensional functions by using Hessian and Gradient
- We would get a descent $\mathbf{d}^k$ by setting:

$$\mathbf{d}^k = -\frac{\mathbf{g}^k}{\mathbf{H}^k}$$

- There are several options to avoid (often costly) computation of the Hessian $\mathbf{H}$:

1. Quasi-Newton updates $\mathbf{H}$ starting from identity matrix
2. Broyden-Fletcher-Goldfarb-Shanno (BFGS) does better with approx linesearch
3. L-BFGS is the limited memory version for large problems

In [8]:
```
optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!, [0.0, 0.0], Newton(),Optim.Options(
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

In [9]:
```
@show optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!,  [-1.0, 3.0], BFGS());
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

In [10]:
```
# low memory BFGS
 @show optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!,  [0.0, 0.0], LBFGS());
```

```
UndefVarError: rosenbrock not defined
```

```
Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.7 Direct Methods

- No derivative information is used - *derivative free*
- If it's very hard / impossible to provide gradient information, this is our only chance.
- Direct methods use other criteria than the gradient to inform the next step (and ulimtately convergence).

1.7.1 Cyclic Coordinate Descent -- Taxicab search

- We do a line search over each dimension, one after the other
- *taxicab* because the path looks like a NYC taxi changing direction at each block.
- given $\mathbf{x}^{(1)}$, we proceed

$$\mathbf{x}^{(2)} = \arg\min_{x_1} f(x_1, x_2^{(1)}, \ldots, x_n^{(1)})$$
$$\mathbf{x}^{(3)} = \arg\min_{x_2} f(x_1^{(2)}, x_2, x_3^{(2)} \ldots, x_n^{(2)})$$

- unfortunately this can easily get stuck because it can only move in 2 directions.

In [11]:
```
# start to setup a basis function, i.e. unit vectors to index each direction:
basis(i, n) = [k == i ? 1.0 : 0.0 for k in 1 : n]
function cyclic_coordinate_descent(f, x, )
    , n = Inf, length(x)
    while abs() >
        x = copy(x)
            for i in 1 : n
                d = basis(i, n)
                x = line_search(f, x, d)
            end
          = norm(x - x)
    end
    return x
end
```

Out[11]:
```
cyclic_coordinate_descent (generic function with 1 method)
```

1.7.2 General Pattern Search

- We search according to an arbitrary *pattern* $\mathcal{P}$ of candidate points, anchored at current guess $\mathbf{x}$.
- With step size α and set $\mathcal{D}$ of directions

$$\mathcal{P} = \mathbf{x} + \alpha\mathbf{d} \text{ for } \mathbf{d} \in \mathcal{D}$$

- Convergence is guaranteed under conditions:

- $\mathcal{D}$ must be a positive spanning set: at least one $\mathbf{d} \in \mathcal{D}$ has a non-zero gradient.

In [12]:
```
function generalized_pattern_search(f, x, , D, , =0.5)
    y, n = f(x), length(x)
    evals = 0
    while  >
        improved = false
        for (i,d) in enumerate(D)
            x = x + *d
            y = f(x)
            evals += 1
            if y < y
                x, y, improved = x, y, true
                D = unshift!(deleteat!(D, i), d)
                break
            end
        end
        if !improved
             *=
        end
    end
    println("$evals evaluations")
    return x
end
```

Out[12]:
```
generalized_pattern_search (generic function with 2 methods)
```

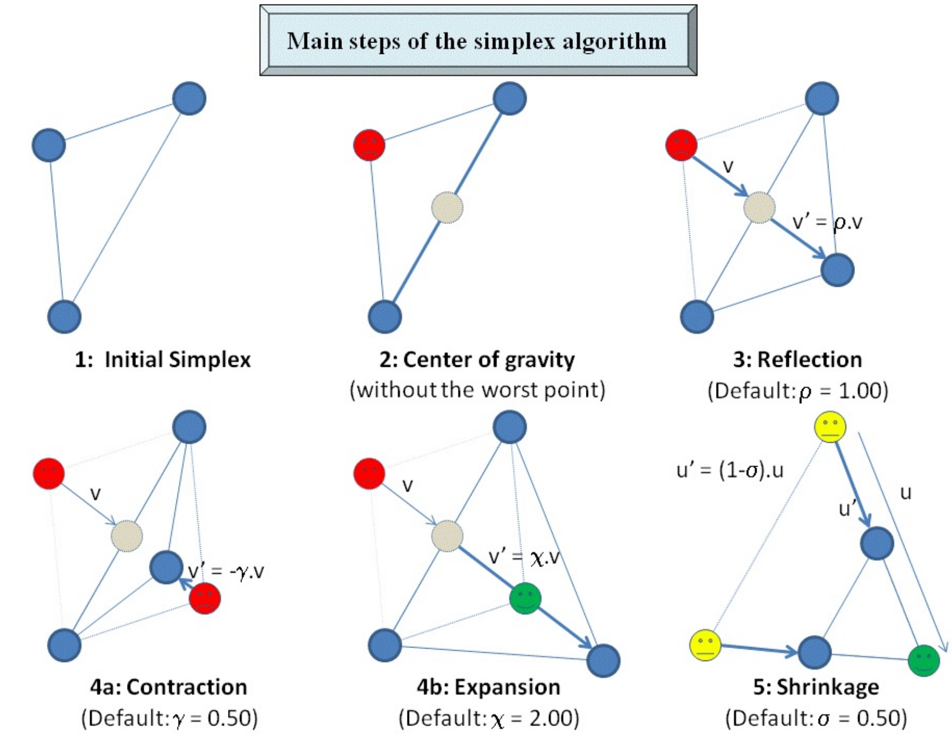
In [13]:
```
D = [[1,0],[0,1],[-1,-0.5]]
y=generalized_pattern_search(rosenbrock.f,zeros(2),0.8,D,1e-6 )
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

1.8 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 [14]: optimize(rosenbrock.f, [0.0, 0.0], NelderMead())
```

```
UndefVarError: rosenbrock not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

- But.

1.9 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.

1.10 things to read up on

- Divided Rectangles (DIRECT)
- simulated annealing and other stochastic gradient methods

2 Constraints

2.1 lagrange multipliers

2.2 duality

2.3 penalty methods

2.4 augmented lagrange

2.5 interior point

2.5.1 Examples

$$\min_{x \in \mathbb{R}^2} \sqrt{x_2} \text{ subject to } \begin{array}{r} x_2 \geq 0 \\ x_2 \geq (a_1 x_1 + b_1)^3 \\ x_2 \geq (a_2 x_1 + b_2)^3 \end{array}$$

2.6 Constrained Optimisation with `NLopt.jl`

- We need to specify one function for each objective and constraint.
- Both of those functions need to compute the function value (i.e. objective or constraint) *and* it's respective gradient.
- `NLopt` expects contraints **always** to be formulated in the format

$$g(x) \leq 0$$

  where g is your constraint function
- The constraint function is formulated for each constraint at x. it returns a number (the value of the constraint at x), and it fills out the gradient vector, which is the partial derivative of the current constraint wrt x.
- There is also the option to have vector valued constraints, see the documentation.
- We set this up as follows:

In [15]:
```
using NLopt

count = 0 # keep track of # function evaluations

function myfunc(x::Vector, grad::Vector)
    if length(grad) > 0
        grad[1] = 0
        grad[2] = 0.5/sqrt(x[2])
```

```
    end

    global count
    count::Int += 1
    println("f_$count($x)")

    sqrt(x[2])
end

function myconstraint(x::Vector, grad::Vector, a, b)
    if length(grad) > 0
        grad[1] = 3a * (a*x[1] + b)^2
        grad[2] = -1
    end
    (a*x[1] + b)^3 - x[2]
end

opt = Opt(:LD_MMA, 2)
lower_bounds!(opt, [-Inf, 0.])
xtol_rel!(opt,1e-4)

min_objective!(opt, myfunc)
inequality_constraint!(opt, (x,g) -> myconstraint(x,g,2,0), 1e-8)
inequality_constraint!(opt, (x,g) -> myconstraint(x,g,-1,1), 1e-8)

(minfn,minx,ret) = NLopt.optimize(opt, [1.234, 5.678])
println("got $minf at $minx after $count iterations (returned $ret)")
```

```
    ArgumentError: Module NLopt not found in current path.
Run `Pkg.add("NLopt")` to install the NLopt package.



    Stacktrace:

     [1] _require(::Symbol) at ./loading.jl:428

     [2] require(::Symbol) at ./loading.jl:398

     [3] include_string(::String, ::String) at ./loading.jl:515
```

2.7 NLopt: Rosenbrock

- Let's tackle the rosenbrock example again.
- To make it more interesting, let's add an inequality constraint.

$$\min_{x\in\mathbb{R}^2}(1-x_1)^2+100(x_2-x_1^2)^2 \text{ subject to } 0.8-x_1^2-x_2^2\geq 0$$

- in NLopt format, the constraint is $x_1 + x_2 - 0.8 \leq 0$

In [16]:
```
function rosenbrockf(x::Vector,grad::Vector)
    if length(grad) > 0
            grad[1] = -2.0 * (1.0 - x[1]) - 400.0 * (x[2] - x[1]^2) * x[1]
            grad[2] = 200.0 * (x[2] - x[1]^2)
    end
    return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
end
function r_constraint(x::Vector, grad::Vector)
    if length(grad) > 0
        grad[1] = 2*x[1]
        grad[2] = 2*x[2]
        end
        return x[1]^2 + x[2]^2 - 0.8
end
opt = Opt(:LD_MMA, 2)
lower_bounds!(opt, [-5, -5.0])
min_objective!(opt,(x,g) -> rosenbrockf(x,g))
inequality_constraint!(opt, (x,g) -> r_constraint(x,g))
ftol_rel!(opt,1e-9)
NLopt.optimize(opt, [-1.0,0.0])
```

```
UndefVarError: Opt not defined



Stacktrace:

 [1] include_string(::String, ::String) at ./loading.jl:515
```

2.8 JuMP.jl

- Introduce `JuMP.jl`
- JuMP is a mathematical programming interface for Julia. It is like AMPL, but for free and with a decent programming language.
- The main highlights are:
  - It uses automatic differentiation to compute derivatives from your expression.
  - It supplies this information, as well as the sparsity structure of the Hessian to your preferred solver.
  - It decouples your problem completely from the type of solver you are using. This is great, since you don't have to worry about different solvers having different interfaces.
  - In order to achieve this, `JuMP` uses `MathProgBase.jl`, which converts your problem formulation into a standard representation of an optimization problem.
- Let's look at the readme
- The technical citation is Lubin et al [**?**]

2.9 JuMP: Quick start guide

- this is form the quick start guide
- please check the docs, they are excellent.

2.9.1 Step 1: create a model

- A model collects variables, objective function and constraints.
- it defines a solver to be used.

```
using Clp
m = Model(solver=ClpSolver())  # provide a solver

#ăDefine variables
@variable(m, x )              # No bounds
@variable(m, x >= lb )        # Lower bound only (note: 'lb <= x' is not valid)
@variable(m, x <= ub )        # Upper bound only
@variable(m, lb <= x <= ub )  # Lower and upper bounds


# we can create arrays of a variable
N = 2
@variable(m, x[1:M,1:N] >= 0 )

# or put them in a block
@variables m begin
    x
    y >= 0
    Z[1:10], Bin
    X[1:3,1:3], SDP
    q[i=1:2], (lowerbound = i, start = 2i, upperbound = 3i)
    t[j=1:3], (Int, start = j)
end

# Equivalent to:
@variable(m, x)
@variable(m, y >= 0)
@variable(m, Z[1:10], Bin)
@variable(m, X[1:3,1:3], SDP)
@variable(m, q[i=1:2], lowerbound = i, start = 2i, upperbound = 3i)
@variable(m, t[j=1:3], Int, start = j)

# bounds can depend on indices
@variable(m, x[i=1:10] >= i )
```

2.10 Objective and Constraints

- We can easily add objective and constraint functions:

```
@constraint(m, x[i] - s[i] <= 0)  # Other options: == and >=
@constraint(m, sum(x[i] for i=1:numLocation) == 1)
@objective(m, Max, 5x + 22y + (x+y)/2) # or Min
```

- This is fully integrated with Julia. you can use the `generator` syntax for sums:

```
@objective(sum(x[i] + y[i]/pi for i = I1, j = I2 if i+j < some_val))
```

In [17]:
```
##ăSimple example
using JuMP
using Clp

m = Model(solver = ClpSolver())
@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )

@objective(m, Max, 5x + 3*y )
@constraint(m, 1x + 5y <= 3.0 )

print(m)

status = solve(m)

println("Objective value: ", getobjectivevalue(m))
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

```
INFO: Recompiling stale cache file /Users/74097/.julia/lib/v0.6/JuMP.ji for module JuMP.INFO: Pr

Max 5 x + 3 y
Subject to
 x + 5 y  3
 0  x  2
 0  y  30
Objective value: 10.6
x = 2.0
y = 0.2
```

In [18]:
```
# JuMP: Rosenbrock Example
# Instead of hand-coding first and second derivatives, you only have to give `JuMP` exp
# Here is an example.

using Ipopt

let

    m = Model(solver=IpoptSolver())
```

```
        @variable(m, x)
        @variable(m, y)

        @NLobjective(m, Min, (1-x)^2 + 100(y-x^2)^2)

        solve(m)

        println("x = ", getvalue(x), " y = ", getvalue(y))

    end
```

```
INFO: Recompiling stale cache file /Users/74097/.julia/lib/v0.6/Ipopt.ji for module Ipopt.


******************************************************************************
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 http://projects.coin-or.org/Ipopt
******************************************************************************

This is Ipopt version 3.12.9, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...:        0
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:        3

Total number of variables............................:        2
                     variables with only lower bounds:        0
                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
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.0000000e+00 0.00e+00 2.00e+00  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  9.5312500e-01 0.00e+00 1.25e+01  -1.0 1.00e+00    -  1.00e+00 2.50e-01f  3
   2  4.8320569e-01 0.00e+00 1.01e+00  -1.0 9.03e-02    -  1.00e+00 1.00e+00f  1
   3  4.5708829e-01 0.00e+00 9.53e+00  -1.0 4.29e-01    -  1.00e+00 5.00e-01f  2
   4  1.8894205e-01 0.00e+00 4.15e-01  -1.0 9.51e-02    -  1.00e+00 1.00e+00f  1
   5  1.3918726e-01 0.00e+00 6.51e+00  -1.7 3.49e-01    -  1.00e+00 5.00e-01f  2
   6  5.4940990e-02 0.00e+00 4.51e-01  -1.7 9.29e-02    -  1.00e+00 1.00e+00f  1
   7  2.9144630e-02 0.00e+00 2.27e+00  -1.7 2.49e-01    -  1.00e+00 5.00e-01f  2
   8  9.8586451e-03 0.00e+00 1.15e+00  -1.7 1.10e-01    -  1.00e+00 1.00e+00f  1
   9  2.3237475e-03 0.00e+00 1.00e+00  -1.7 1.00e-01    -  1.00e+00 1.00e+00f  1
```

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  2.3797236e-04 0.00e+00 2.19e-01  -1.7 5.09e-02    -  1.00e+00 1.00e+00f  1
  11  4.9267371e-06 0.00e+00 5.95e-02  -1.7 2.53e-02    -  1.00e+00 1.00e+00f  1
  12  2.8189505e-09 0.00e+00 8.31e-04  -2.5 3.20e-03    -  1.00e+00 1.00e+00f  1
  13  1.0095040e-15 0.00e+00 8.68e-07  -5.7 9.78e-05    -  1.00e+00 1.00e+00f  1
  14  1.3288608e-28 0.00e+00 2.02e-13  -8.6 4.65e-08    -  1.00e+00 1.00e+00f  1

Number of Iterations...: 14

                                   (scaled)                 (unscaled)
Objective...:   1.3288608467480825e-28    1.3288608467480825e-28
Dual infeasibility...:   2.0183854587685121e-13    2.0183854587685121e-13
Constraint violation...:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity...:   0.0000000000000000e+00    0.0000000000000000e+00
Overall NLP error...:   2.0183854587685121e-13    2.0183854587685121e-13


Number of objective function evaluations             = 36
Number of objective gradient evaluations             = 15
Number of equality constraint evaluations            = 0
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 0
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 14
Total CPU secs in IPOPT (w/o function evaluations)   =      0.117
Total CPU secs in NLP function evaluations           =      0.026

EXIT: Optimal Solution Found.
x = 0.9999999999999899 y = 0.9999999999999792
```

In [19]:
```
# not bad, right?
# adding the constraint from before:

let

    m = Model(solver=IpoptSolver())

    @variable(m, x)
    @variable(m, y)

    @NLobjective(m, Min, (1-x)^2 + 100(y-x^2)^2)
    @NLconstraint(m,x^2 + y^2 <= 0.8)

    solve(m)

    println("x = ", getvalue(x), " y = ", getvalue(y))
```

```
        end
```

```
This is Ipopt version 3.12.9, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...:        0
Number of nonzeros in inequality constraint Jacobian.:        2
Number of nonzeros in Lagrangian Hessian...:        5

Total number of variables...:        2
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints...:        0
Total number of inequality constraints...:        1
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        1

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.0000000e+00 0.00e+00 2.00e+00  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  9.5312500e-01 0.00e+00 1.25e+01  -1.0 5.00e-01    -  1.00e+00 5.00e-01f  2
   2  4.9204994e-01 0.00e+00 9.72e-01  -1.0 8.71e-02    -  1.00e+00 1.00e+00f  1
   3  2.0451702e+00 0.00e+00 3.69e+01  -1.7 3.80e-01    -  1.00e+00 1.00e+00H  1
   4  1.0409466e-01 0.00e+00 3.10e-01  -1.7 1.46e-01    -  1.00e+00 1.00e+00f  1
   5  8.5804626e-02 0.00e+00 2.71e-01  -1.7 9.98e-02    -  1.00e+00 1.00e+00h  1
   6  9.4244879e-02 0.00e+00 6.24e-02  -1.7 3.74e-02    -  1.00e+00 1.00e+00h  1
   7  8.0582034e-02 0.00e+00 1.51e-01  -2.5 6.41e-02    -  1.00e+00 1.00e+00h  1
   8  7.8681242e-02 0.00e+00 2.12e-03  -2.5 1.12e-02    -  1.00e+00 1.00e+00h  1
   9  7.6095770e-02 0.00e+00 6.16e-03  -3.8 1.37e-02    -  1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  7.6033892e-02 0.00e+00 2.23e-06  -3.8 3.99e-04    -  1.00e+00 1.00e+00h  1
  11  7.5885642e-02 0.00e+00 2.07e-05  -5.7 7.99e-04    -  1.00e+00 1.00e+00h  1
  12  7.5885428e-02 0.00e+00 2.74e-11  -5.7 1.38e-06    -  1.00e+00 1.00e+00h  1
  13  7.5883585e-02 0.00e+00 3.19e-09  -8.6 9.93e-06    -  1.00e+00 1.00e+00f  1

Number of Iterations...: 13

                                   (scaled)                 (unscaled)
Objective...:   7.5883585442440671e-02    7.5883585442440671e-02
Dual infeasibility...:   3.1949178858070582e-09    3.1949178858070582e-09
Constraint violation...:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity...:   2.5454985882932001e-09    2.5454985882932001e-09
Overall NLP error...:   3.1949178858070582e-09    3.1949178858070582e-09


Number of objective function evaluations             = 20
Number of objective gradient evaluations             = 14
```

```
Number of equality constraint evaluations            = 0
Number of inequality constraint evaluations          = 20
Number of equality constraint Jacobian evaluations   = 0
Number of inequality constraint Jacobian evaluations = 14
Number of Lagrangian Hessian evaluations             = 13
Total CPU secs in IPOPT (w/o function evaluations)   =      0.005
Total CPU secs in NLP function evaluations           =      0.002

EXIT: Optimal Solution Found.
x = 0.7247018392092258 y = 0.5242206029480763
```

2.11 JuMP: Maximium Likelihood

- Let's redo the maximum likelihood example in JuMP.
- Let μ, σ^2 be the unknown mean and variance of a random sample generated from the normal distribution.
- Find the maximum likelihood estimator for those parameters!
- density:

$$f(x_i|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)$$

- Likelihood Function

$$\begin{aligned} L(\mu,\sigma^2) = \Pi_{i=1}^{N} f(x_i|\mu,\sigma^2) &= \frac{1}{(\sigma\sqrt{2\pi})^n} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i-\mu)^2\right) \\ &= \left(\sigma^2 2\pi\right)^{-\frac{n}{2}} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i-\mu)^2\right) \end{aligned}$$

- Constraints: $\mu \in \mathbb{R}, \sigma > 0$
- log-likelihood:

$$\log L = l = -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i-\mu)^2$$

- Let's do this in JuMP.

In [20]:
```
#  Copyright 2015, Iain Dunning, Joey Huchette, Miles Lubin, and contributors
#  example modified
using Distributions

distrib = Normal(4.5,3.5)
n = 10000
```

```julia
data = rand(distrib,n);

m = Model(solver=IpoptSolver())

@variable(m, mu, start = 0.0)
@variable(m, sigma >= 0.0, start = 1.0)

@NLobjective(m, Max, -(n/2)*log(2*sigma^2)-sum((data[i] - mu) ^ 2 for i = 1:n)/(2*sigma

solve(m)
println(" = ", getvalue(mu),", mean(data) = ", mean(data))
println("^2 = ", getvalue(sigma)^2, ", var(data) = ", var(data))
```

```
This is Ipopt version 3.12.9, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...:        0
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian...:        3

Total number of variables...:        2
                     variables with only lower bounds:        1
                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
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.6887292e+05 0.00e+00 1.01e+02  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.2430166e+05 0.00e+00 1.03e+02  -1.0 9.54e+00    -  1.00e+00 5.00e-01f  2
   2  7.6133884e+04 0.00e+00 4.27e+01  -1.0 2.30e-01    -  8.90e-01 1.00e+00f  1
   3  5.0257956e+04 0.00e+00 1.77e+01  -1.0 2.99e-01    -  1.00e+00 1.00e+00f  1
   4  3.6881104e+04 0.00e+00 7.15e+00  -1.0 3.78e-01    -  1.00e+00 1.00e+00f  1
   5  3.0407076e+04 0.00e+00 2.81e+00  -1.0 4.59e-01    -  1.00e+00 1.00e+00f  1
   6  2.7667909e+04 0.00e+00 1.02e+00  -1.0 5.11e-01    -  1.00e+00 1.00e+00f  1
   7  2.6785732e+04 0.00e+00 3.17e-01  -1.0 4.84e-01    -  1.00e+00 1.00e+00f  1
   8  2.6637016e+04 0.00e+00 5.63e-02  -1.7 3.00e-01    -  1.00e+00 1.00e+00f  1
   9  2.6629671e+04 0.00e+00 3.26e-03  -2.5 8.74e-02    -  1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  2.6629653e+04 0.00e+00 8.70e-06  -3.8 4.73e-03    -  1.00e+00 1.00e+00f  1
  11  2.6629653e+04 0.00e+00 1.49e-09  -5.7 6.18e-05    -  1.00e+00 1.00e+00f  1
  12  2.6629653e+04 0.00e+00 4.01e-13  -8.6 9.87e-07    -  1.00e+00 1.00e+00f  1

Number of Iterations...: 12
```

```
                                   (scaled)                 (unscaled)
Objective...:   8.6077852994465989e+00    2.6629653293957461e+04
Dual infeasibility...:   4.0102196914099508e-13    1.2406299216328470e-09
Constraint violation...:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity...:   2.5064286232902822e-09    7.7540648287966047e-06
Overall NLP error...:   2.5064286232902822e-09    7.7540648287966047e-06


Number of objective function evaluations             = 18
Number of objective gradient evaluations             = 13
Number of equality constraint evaluations            = 0
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 0
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 12
Total CPU secs in IPOPT (w/o function evaluations)   =      0.005
Total CPU secs in NLP function evaluations           =      0.017

EXIT: Optimal Solution Found.
 = 4.460816556166759, mean(data) = 4.46081655616676
^2 = 12.037822802323896, var(data) = 12.039026695664448
```

In [ ]: