SciencesPo Computational Economics
Spring 2019

Florian Oswald

April 15, 2019

1 Optimization 2: Algorithms and Constraints

Florian Oswald Sciences Po, 2019

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

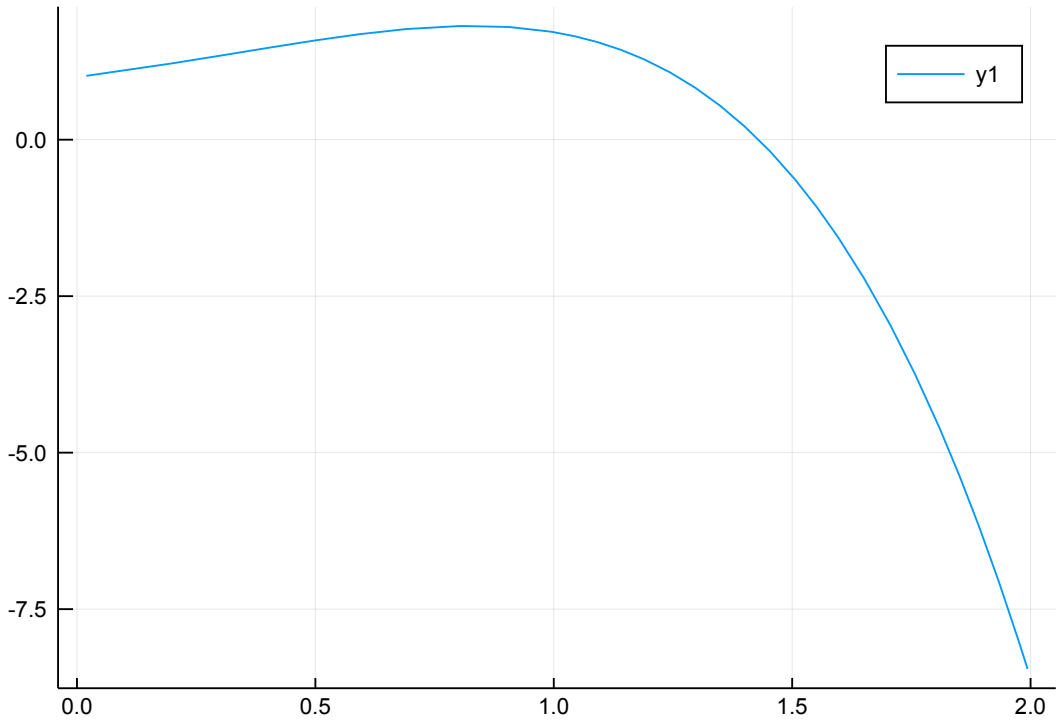
- A derivative-free method
- a Bracketing method
  - find the local minimum of f on $[a, b]$
  - select 2 interior points c, d such that $a < c < d < b$
    - $f(c) \leq f(d) \implies$ min must lie in $[a, d]$. replace b with d, start again with $[a, d]$
    - $f(c) > f(d) \implies$ min must lie in $[c, b]$. 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
        gr()
        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
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
```

Out[1]:



1.2.1 Bisection Methods

- Root finding: `Roots.jl`
- Root finding in multivariate functions: `IntervalRootFinding.jl`

```
In [80]: using Roots
         #ăfind the zeros of this function:
         f(x) = exp(x) - x^4
         ## bracketing
         fzero(f, 8, 9)      # 8.613169456441398
         fzero(f, -10, 0) # -0.8155534188089606

Out[80]: -0.8155534188089606

In [36]: using IntervalRootFinding, IntervalArithmetic
         -10..10

Out[36]: [-10, 10]

In [37]: X = IntervalBox(1..3, 2..4)

Out[37]: [1, 3] 𝔼 [2, 4]

In [38]: a = @interval(0.1, 0.3)
         b = @interval(0.3, 0.6)
         a + b

Out[38]: [0.399999, 0.900001]

In [41]: rts = roots(x->x^2 - 2, -10..10, IntervalRootFinding.Bisection)

Out[41]: 2-element Array{Root{Interval{Float64}},1}:
          Root([1.41377, 1.41439], :unknown)
          Root([-1.41471, -1.41407], :unknown)
```

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 [4]: using Optim
        using OptimTestProblems
        for (name, prob) in MultivariateProblems.UnconstrainedProblems.examples
           println(name)
        end

Rosenbrock
Quadratic Diagonal
Hosaki
Large Polynomial
```

```
Penalty Function I
Beale
Extended Rosenbrock
Polynomial
Powell
Exponential
Paraboloid Diagonal
Paraboloid Random Matrix
Extended Powell
Trigonometric
Fletcher-Powell
Parabola
Himmelblau
```

```
In [5]: rosenbrock = MultivariateProblems.UnconstrainedProblems.examples["Rosenbrock"]
```

```
Out[5]: OptimTestProblems.MultivariateProblems.OptimizationProblem{Nothing,Nothing,Float64,Str
```

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 [44]: # 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]
```

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 [46]:
```
# https://github.com/JuliaNLSolvers/LineSearches.jl
using LineSearches

algo_hz = Optim.Newton(linesearch = HagerZhang())    # Both Optim.jl and IntervalRoot
res_hz = Optim.optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!, rosenbrock.initial
```

Out[46]:
```
Results of Optimization Algorithm
 * Algorithm: Newton's Method
 * Starting Point: [-1.2,1.0]
 * Minimizer: [1.0000000000000033,1.0000000000000067]
 * Minimum: 1.109336e-29
 * Iterations: 23
 * Convergence: true
   * |x - x'|  0.0e+00: false
     |x - x'| = 1.13e-08
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 6.35e+13 |f(x)|
   * |g(x)|  1.0e-08: true
     |g(x)| = 6.66e-15
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 71
 * Gradient Calls: 71
 * Hessian Calls: 23
```

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 [47]: # 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 t
                             rho_upper = 0.75) # When rho is greater than rho_upper, grow the
         res = Optim.optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!, rosenbrock.initial_x
```

```
Out[47]: Results of Optimization Algorithm
          * Algorithm: Newton's Method (Trust Region)
          * Starting Point: [-1.2,1.0]
          * Minimizer: [0.9999999994405535,0.9999999988644926]
          * Minimum: 3.405841e-19
          * Iterations: 25
          * Convergence: true
            * |x - x'|  0.0e+00: false
              |x - x'| = 8.84e-06
            * |f(x) - f(x')|  0.0e+00 |f(x)|: false
              |f(x) - f(x')| = 1.87e+08 |f(x)|
            * |g(x)|  1.0e-08: true
              |g(x)| = 5.53e-09
            * Stopped by an increasing objective: false
            * Reached Maximum Number of Iterations: false
          * Objective Calls: 26
          * Gradient Calls: 26
          * Hessian Calls: 22
```

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

```
Out[48]: GradientDescent{InitialPrevious{Float64},HagerZhang{Float64,Base.RefValue{Bool}},Noth
           alpha: Float64 1.0
           alphamin: Float64 0.0
           alphamax: Float64 Inf
         , HagerZhang{Float64,Base.RefValue{Bool}}
           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
           mayterminate: Base.RefValue{Bool}
         , nothing, getfield(Main, Symbol("##49#50"))(), Flat())
```

```
In [49]: # there is a dedicated LineSearch package: https://github.com/JuliaNLSolvers/LineSear
         GD = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent())
         GD1 = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent(),Optim.Options
         GD2 = optimize(rosenbrock.f, rosenbrock.g!,[0.0, 0.0],GradientDescent(),Optim.Options

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

```
gradient descent = Results of Optimization Algorithm
 * Algorithm: Gradient Descent
 * Starting Point: [0.0,0.0]
 * Minimizer: [0.9356732500354086,0.875073922357589]
 * Minimum: 4.154782e-03
 * Iterations: 1000
```

```
 * Convergence: false
   * |x - x'|  0.0e+00: false
     |x - x'| = 1.82e-04
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 1.97e-03 |f(x)|
   * |g(x)|  1.0e-08: false
     |g(x)| = 8.21e-02
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: true
 * Objective Calls: 2532
 * Gradient Calls: 2532


gradient descent 2 = Results of Optimization Algorithm
 * Algorithm: Gradient Descent
 * Starting Point: [0.0,0.0]
 * Minimizer: [0.9978398797724763,0.9956717950747302]
 * Minimum: 4.682073e-06
 * Iterations: 5000
 * Convergence: false
   * |x - x'|  0.0e+00: false
     |x - x'| = 5.08e-06
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 1.62e-03 |f(x)|
   * |g(x)|  1.0e-08: false
     |g(x)| = 2.53e-03
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: true
 * Objective Calls: 12532
 * Gradient Calls: 12532


gradient descent 3 = Results of Optimization Algorithm
 * Algorithm: Gradient Descent
 * Starting Point: [0.0,0.0]
 * Minimizer: [0.9999999914304203,0.9999999828109042]
 * Minimum: 7.368706e-17
 * Iterations: 20458
 * Convergence: true
   * |x - x'|  0.0e+00: false
     |x - x'| = 2.00e-11
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 1.61e-03 |f(x)|
   * |g(x)|  1.0e-08: true
     |g(x)| = 9.99e-09
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 51177
```

```
 * Gradient Calls: 51177
```

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

```
Iter     Function value   Gradient norm
     0     1.000000e+00     2.000000e+00
     1     8.431140e-01     1.588830e+00
     2     6.776980e-01     3.453340e+00
     3     4.954645e-01     4.862093e+00
     4     3.041921e-01     2.590086e+00
     5     1.991512e-01     3.780900e+00
     6     9.531907e-02     1.299090e+00
     7     5.657827e-02     2.445401e+00
     8     2.257807e-02     1.839332e+00
     9     6.626125e-03     1.314236e+00
    10     8.689753e-04     5.438279e-01
    11     4.951399e-06     7.814556e-02
    12     9.065070e-10     6.017046e-04
    13     9.337686e-18     1.059738e-07
    14     3.081488e-31     1.110223e-15
```

```
Out[6]: Results of Optimization Algorithm
         * Algorithm: Newton's Method
         * Starting Point: [0.0,0.0]
         * Minimizer: [0.9999999999999994,0.9999999999999989]
         * Minimum: 3.081488e-31
         * Iterations: 14
         * Convergence: true
           * |x - x'|  0.0e+00: false
             |x - x'| = 3.06e-09
           * |f(x) - f(x')|  0.0e+00 |f(x)|: false
             |f(x) - f(x')| = 3.03e+13 |f(x)|
           * |g(x)|  1.0e-08: true
             |g(x)| = 1.11e-15
           * Stopped by an increasing objective: false
           * Reached Maximum Number of Iterations: false
         * Objective Calls: 44
         * Gradient Calls: 44
         * Hessian Calls: 14
```

```
In [7]: @show optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!,  [-1.0, 3.0], BFGS());
```

```
optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!, [-1.0, 3.0], BFGS()) = Results of Optimiza
 * Algorithm: BFGS
 * Starting Point: [-1.0,3.0]
 * Minimizer: [0.9999999999999956,0.999999999999987]
 * Minimum: 1.707144e-27
 * Iterations: 39
 * Convergence: true
   * |x - x'|  0.0e+00: false
     |x - x'| = 1.54e-08
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 3.55e+10 |f(x)|
   * |g(x)|  1.0e-08: true
     |g(x)| = 1.63e-12
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 137
 * Gradient Calls: 137
```

```
In [8]: # low memory BFGS
        @show optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!,  [0.0, 0.0], LBFGS());
```

```
optimize(rosenbrock.f, rosenbrock.g!, rosenbrock.h!, [0.0, 0.0], LBFGS()) = Results of Optimiza
 * Algorithm: L-BFGS
 * Starting Point: [0.0,0.0]
 * Minimizer: [0.999999999999928,0.9999999999998559]
 * Minimum: 5.191703e-27
 * Iterations: 24
```

```
 * Convergence: true
   * |x - x'|  0.0e+00: false
     |x - x'| = 4.58e-11
   * |f(x) - f(x')|  0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 8.50e+07 |f(x)|
   * |g(x)|  1.0e-08: true
     |g(x)| = 1.44e-13
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 67
 * Gradient Calls: 67
```

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).

1.6.2 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 [9]:
```
# 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[9]:
```
cyclic_coordinate_descent (generic function with 1 method)
```

1.6.3 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 [10]: 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 = pushfirst!(deleteat!(D, i), d)
                         break
                     end
                 end
                 if !improved
                      *=
                 end
             end
             println("$evals evaluations")
             return x
         end
```

```
Out[10]: generalized_pattern_search (generic function with 2 methods)
```

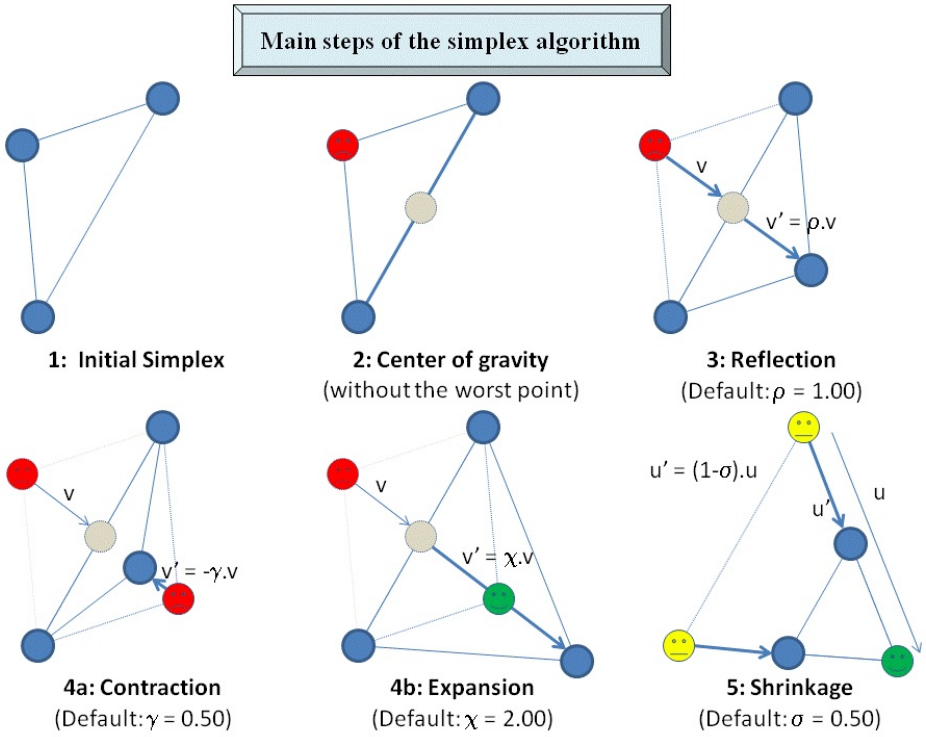
```
In [11]: D = [[1,0],[0,1],[-1,-0.5]]
         D = [[1,0],[0,1]]
         y=generalized_pattern_search(rosenbrock.f,zeros(2),0.8,D,1e-6 )
```

```
11923 evaluations
```

```
Out[11]: 2-element Array{Float64,1}:
          0.9996734619140493
          0.9993469238280956
```

1.7 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 [12]: nm=optimize(rosenbrock.f, [0.0, 0.0], NelderMead());
         nm.minimizer

Out[12]: 2-element Array{Float64,1}:
          0.9999634355313174
          0.9999315506115275
```

- But.

1.8 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.

1.9 things to read up on

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

1.10 Stochastic Optimization Methods

- Gradient based methods like steepest descent may be susceptible to getting stuck at local minima.
- Randomly shocking the value of the descent direction may be a solution to this.
- For example, one could modify our gradient descent from before to become

$$\mathbf{x}^{(k+1)} \longleftarrow \mathbf{x}^{(k)} + \alpha^k \mathbf{g}^{(k)} + \varepsilon^{(k)}$$

- where $\varepsilon^{(k)} \sim N(0, \sigma_k^2)$, decreasing with k.
- This *stochastic gradient descent* is often used when training neural networks.

1.10.1 Simulated Annealing

- We specify a *temperature* that controls the degree of randomness.
- At first the temperature is high, letting the search jump around widely. This is to escape local minima.
- The temperature is gradually decreased, reducing the step sizes. This is to find the local optimimum in the *best* region.
- At every iteration k, we accept new point $\mathbf{x}'$ with

$$\Pr(\text{accept } \mathbf{x}') = \begin{cases} 1 & \text{if } \Delta y \leq 0 \\ \min(e^{\Delta y / t}, 1) & \text{if } \Delta y > 0 \end{cases}$$

- here $\Delta y = f(\mathbf{x}') - f(\mathbf{x})$, and t is the *temperature*.
- $\Pr(\text{accept } \mathbf{x}')$ is called the **Metropolis Criterion**, building block of *Accept/Reject* algorithms.

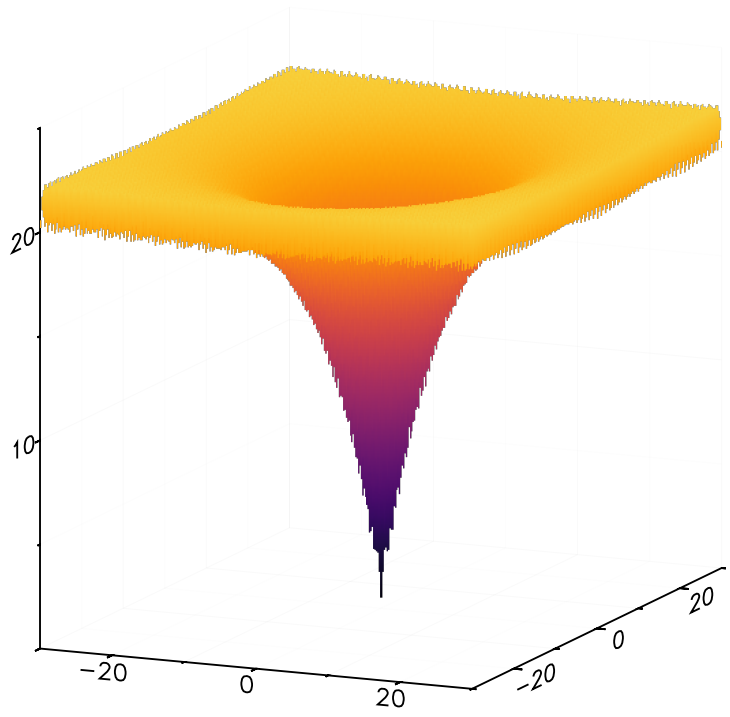
```
In [15]: #ăf: function
         # x: initial point
         # T: transition distribution
         #ăt: temp schedule, k_max: max iterations
         function simulated_annealing(f, x, T, t, k_max)
             y = f(x)
             ytrace = zeros(typeof(y),k_max)
```

```
        x_best, y_best = x, y
        for k in 1 : k_max
            x = x + rand(T)
            y = f(x)
            y = y - y
            if y  0 || rand() < exp(-y/t(k))
                x, y = x, y
            end
            if y < y_best
                x_best, y_best = x, y
            end
            ytrace[k] = y_best
        end
        return x_best,ytrace
    end
```

Out[15]: simulated_annealing (generic function with 1 method)

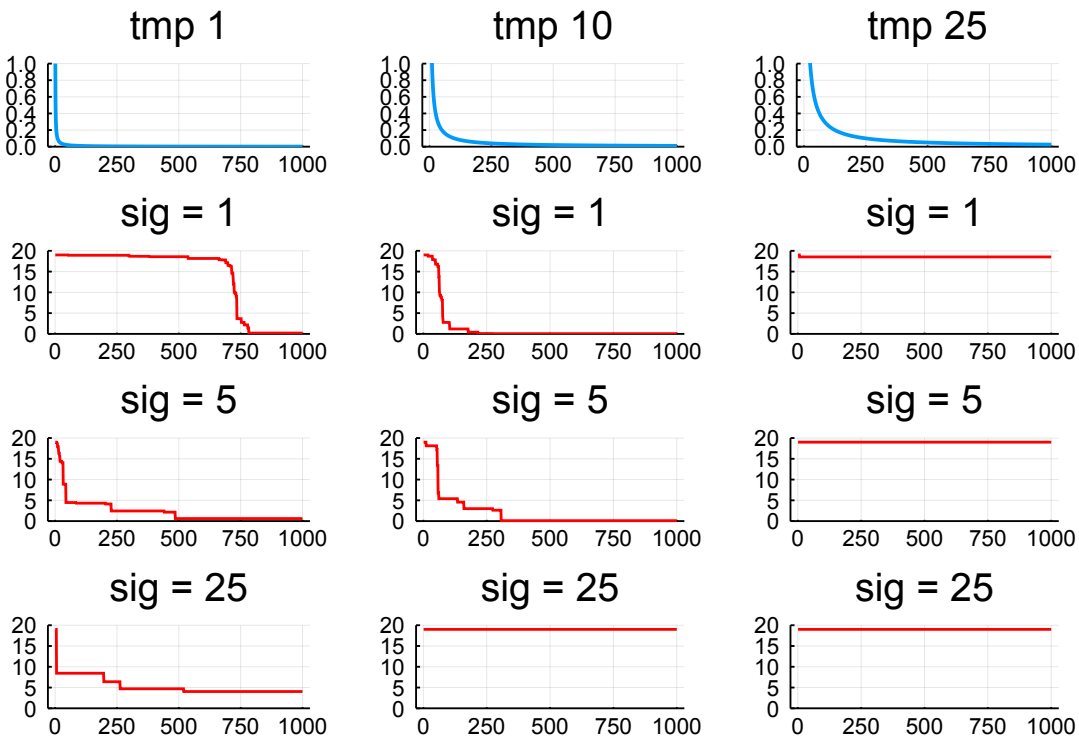
In [1]:
```
function ackley(x, a=20, b=0.2, c=2)
    d = length(x)
    return -a*exp(-b*sqrt(sum(x.^2)/d)) - exp(sum(cos.(c*xi) for xi in x)/d) + a + exp
end
using Plots
gr()
surface(-30:0.1:30,-30:0.1:30,(x,y)->ackley([x, y]),cbar=false)
```

Out[1]:



In [16]:
```
p = Any[]
using Distributions
gr()
niters = 1000
temps = (1,10,25)
push!(p,[plot(x->i/x,1:1000,title = "tmp $i",lw=2,ylims = (0,1),leg = false) for i in
for sig in (1,5,25), t1 in (1,10,25)
    y = simulated_annealing(ackley,[15,15],MvNormal(2,sig),x->t1/x,1000)[2][:]
    push!(p,plot(y,title = "sig = $sig",leg=false,lw=1.5,color="red",ylims = (0,20)))
end
plot(p...,layout = (4,3))
```

Out[16]:



2 Constraints

Recall our core optimization problem:

$$\min_{x \in \mathbb{R}^n} f(x) \text{ s.t. } x \in \mathcal{X}$$

- Up to now, the feasible set was $\mathcal{X} \in \mathbb{R}^n$.
- In **constrained problems** $\mathcal{X}$ is a subset thereof.
- We already encountered *box constraints*, e.g. $x \in [a, b]$.
- Sometimes the contrained solution coincides with the unconstrained one, sometimes it does not.
- There are *equality constraints* and *inequality constraints*.

2.1 Lagrange Multipliers

- Used to optimize a function subject to equality constraints.

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

where both f and h have continuous partial derivatives.

- We look for contour lines of f that are aligned to contours of $h(x) = 0$.

In other words, we want to find the best x s.t. $h(x) = 0$ and we have

$$\nabla f(x) = \lambda \nabla h(x)$$

for some *Lagrange Mutliplier* λ * Notice that we need the scalar λ because the magnitudes of the gradients may be different. * We therefore form the the **Lagrangian**:

$$\mathcal{L}(x, \lambda) = f(x) - \lambda h(x)$$

2.1.1 Example

Suppose we have

$$\min_x -\exp\left(-\left(x_1 x_2 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right)$$
$$\text{subject to } x_1 - x_2^2 = 0$$

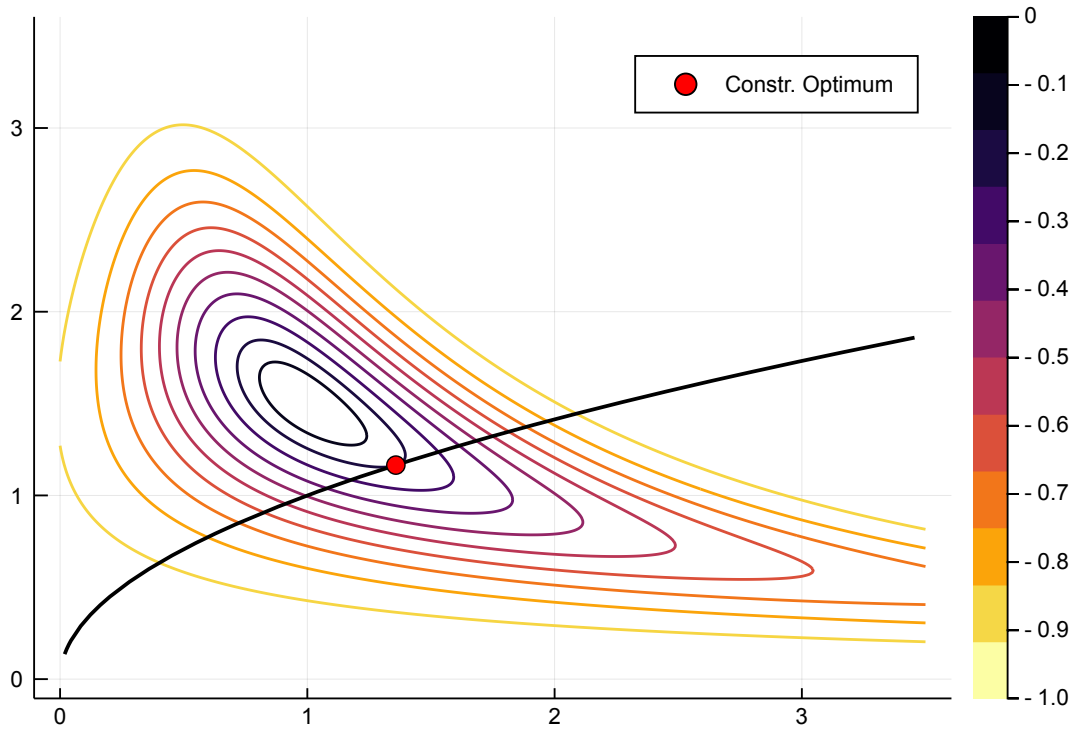
We form the Lagrangiagn:

$$\mathcal{L}(x_1, x_2, \lambda) = -\exp\left(-\left(x_1 x_2 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right) - \lambda(x_1 - x_2^2)$$

Then we compute the gradient wrt to x_1, x_2, λ, set to zero and solve.

```
In [11]: gr()
         f(x1,x2) = -exp.(-(x1.*x2 - 3/2).^2 - (x2-3/2).^2)
         c(x1) = sqrt(x1)
         x=0:0.01:3.5
         contour(x,x,(x,y)->f(x,y),lw=1.5,levels=[collect(0:-0.1:-0.85)...,-0.887,-0.95,-1])
         plot!(c,0.01,3.5,label="",lw=2,color=:black)
         scatter!([1.358],[1.165],markersize=5,markercolor=:red,label="Constr. Optimum")
```

Out[11]:



- If we had multiple constraints (l), we'd just add them up to get

$$\mathcal{L}(\mathbf{x}, \breve{}) = f(\mathbf{x}) - \sum_{i=1}^{l} \lambda_i h_i(\mathbf{x})$$

2.2 Inequality Constraints

Suppose now we had

$$\min_{\mathbf{x}} f(\mathbf{x})$$
$$\text{subject to } g(\mathbf{x}) \leq 0$$

which, if the solution lies right on the constraint *boundary*, means that

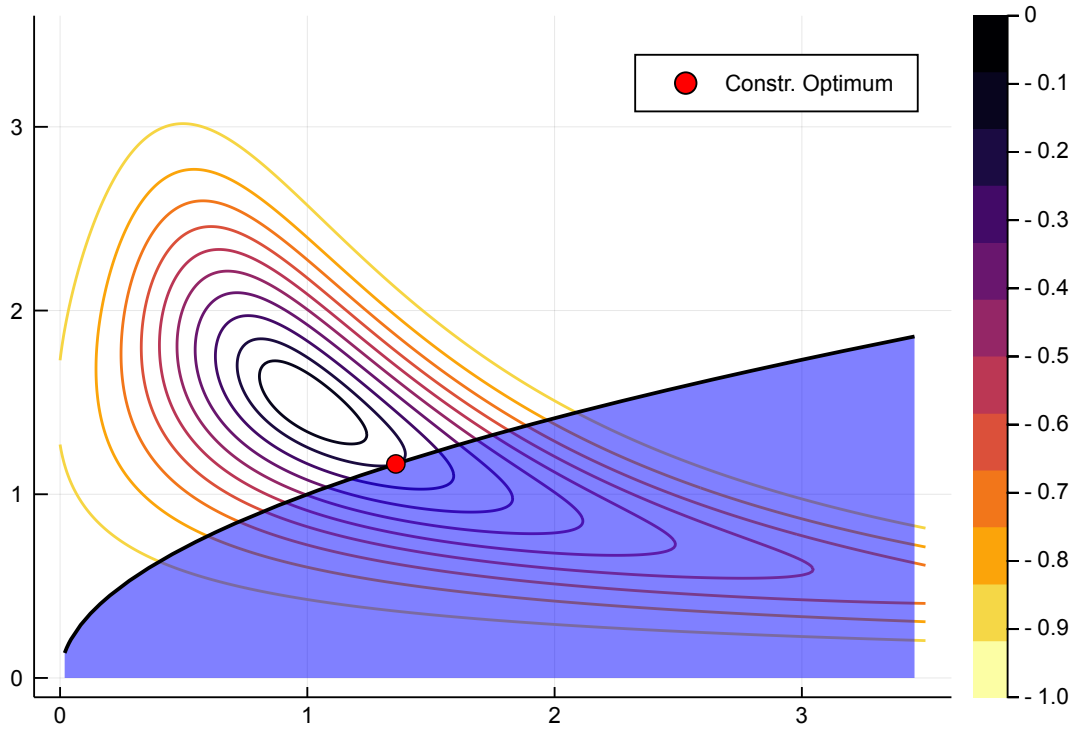
$$\nabla f - \mu \nabla g = 0$$

for some scalar μ - as before.

- In this case, we say the **constraint is active**.
- In the opposite case, i.e. the solution lies **inside** the contrained region, we way the contraint is **inactive**.
- In that case, we are back to an *unconstrained* problem, look for $\nabla f = 0$, and set $\mu = 0$.

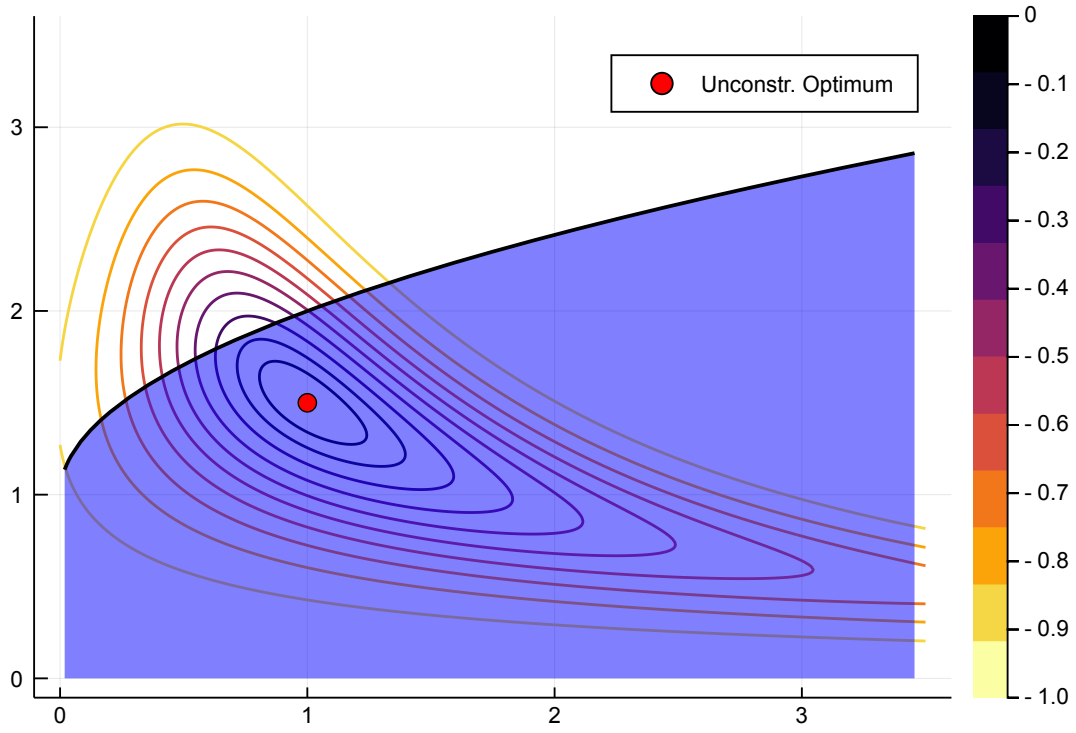
In [12]:
```
#ăthe blue area shows the FEASIBLE SET
contour(x,x,(x,y)->f(x,y),lw=1.5,levels=[collect(0:-0.1:-0.85)...,-0.887,-0.95,-1])
plot!(c,0.01,3.5,label="",lw=2,color=:black,fill=(0,0.5,:blue))
scatter!([1.358],[1.165],markersize=5,markercolor=:red,label="Constr. Optimum")
```

Out[12]:



In [13]:
```
#ăthe blue area shows the FEASIBLE SET
#ăNOW THE CONSTRAINT IS INACTIVE OR SLACK!
c2(x1) = 1+sqrt(x1)
contour(x,x,(x,y)->f(x,y),lw=1.5,levels=[collect(0:-0.1:-0.85)...,-0.887,-0.95,-1])
plot!(c2,0.01,3.5,label="",lw=2,color=:black,fill=(0,0.5,:blue))
scatter!([1],[1.5],markersize=5,markercolor=:red,label="Unconstr. Optimum")
```

Out[13]:



2.3 Infinity Step

- We could do an **infinite step** to avoid *infeasible points*:

$$f_{\infty\text{-step}} = \begin{cases} f(\mathbf{x}) & \text{if } g(\mathbf{x}) \leq 0 \\ \infty & \text{else.} \end{cases}$$
$$= f(\mathbf{x}) + \infty(g(\mathbf{x} > 0)$$

- Unfortunately, this is discontinous and non-differentiable, i.e. hard to handle for algorithms.
- Instead, we use a *linear penalty* $\mu g(\mathbf{x})$ on the objective if the constraint is violated.
- The penalty provides a lower bound to ∞:

$$\mathcal{L}(\mathbf{x}, \mu) = f(\mathbf{x}) + \mu g(\mathbf{x})$$

- We can get back the infinite step by maximizing the penalty:

$$f_{\infty\text{-step}} = \max_{\mu \geq 0} \mathcal{L}(\mathbf{x}, \mu)$$

- Every infeasible $\mathbf{x}$ returns ∞, all others return $f(\mathbf{x})$

2.4 Kuhn-Karush-Tucker (KKT)

- Our problem thus becomes

$$\min_{\mathbf{x}} \max_{\mu \geq 0} \mathcal{L}(\mathbf{x}, \mu)$$

- This is called the **primal problem**. Optimizing this requires:

1. $g(\mathbf{x}^*) \leq 0$. Point is feasible.
2. $\mu \geq 0$. Penalty goes into the right direction. *Dual feasibility*.
3. $\mu g(\mathbf{x}^*) = 0$. Feasible point on the boundary has $g(\mathbf{x}) = 0$, otherwise $g(\mathbf{x}) < 0$ and $\mu = 0$.
4. $\nabla f(\mathbf{x}^*) - \mu \nabla g(\mathbf{x}^*) = 0$. With an active constraint, we want parallel contours of objective and constraint. When inactive, our optimum just has $\nabla f(\mathbf{x}^*) = 0$, which means $\mu = 0$.

The preceding four conditions are called the Kuhn-Karush-Tucker (KKT) conditions. In the above order, and in general terms, they are:

1. Feasibility
2. Dual Feasibility
3. Complementary Slackness
4. Stationarity.

The KKT conditions are the FONCs for problems with smooth constraints.

2.5 Duality

We can combine equality and inequality constraints:

$$\mathcal{L}(\mathbf{x}, \breve{}, \bar{}) = f(\mathbf{x}) + \sum_i \lambda_i h_i(\mathbf{x}) + \sum_j \mu_j g_j(\mathbf{x})$$

where, notice, we reverted the sign of λ since this is unrestricted.

- The Primal problem is identical to the original problem and just as difficult to solve:

$$\min_{\mathbf{x}} \max_{\bar{}\geq 0, \breve{}} \mathcal{L}(\mathbf{x}, \bar{}, \breve{})$$

- The Dual problem reverses min and max:

$$\max_{\bar{}\geq 0, \breve{}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \bar{}, \breve{})$$

2.5.1 Dual Values

- The *max-min-inequality* states that for any function $f(a, b)$

$$\max_{\mathbf{a}} \min_{\mathbf{b}} f(\mathbf{a}, \mathbf{b}) \leq \min_{\mathbf{b}} \max_{\mathbf{a}} f(\mathbf{a}, \mathbf{b})$$

- Hence, the solution to the dual is a lower bound to the solution of the primal problem.
- The solution to the *dual function*, $\min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \bar{}, \breve{})$ is the min of a collection of linear functions, and thus always concave.
- It is easy to optimize this.
- In general, solving the dual is easy whenever minimizing $\mathcal{L}$ wrt x is easy.

Penalty Methods

- We can convert the constrained problem back to unconstrained by adding penalty terms for constraint violoations.
- A simple method could just count the number of violations:

$$p_{\text{count}}(\mathbf{x}) = \sum_i (h_i(\mathbf{x}) \neq 0) + \sum_j (g_j(\mathbf{x} > 0)$$

- and add this to the objective in an *unconstrained* problem with penalty $\rho > 0$

$$\min_{\mathbf{x}} f(\mathbf{x}) + \rho p_{\text{count}}(\mathbf{x})$$

- One can choose the penalty function: for example, a quadratic penaly will produce a smooth objective function
- Notice that ρ needs to become very large sometimes here.

Augmented Lagrange Method

- This is very similar, but specific to equality constraints.

Interior Point Method

- Also called *barrier method*.
- These methods make sure that the search point remains always feasible.
- As one approaches the constraint boundary, the barrier function goes to infinity. Properties:

1. $p_{\text{barrier}}(\mathbf{x})$ is continuous
2. $p_{\text{barrier}}(\mathbf{x})$ is non negative
3. $p_{\text{barrier}}(\mathbf{x})$ goes to infinitey as one approaches the constraint boundary

2.5.2 Barriers

- Inverse Barrier

$$p_{\text{barrier}}(\mathbf{x}) = -\sum_i \frac{1}{g_i(\mathbf{x})}$$

- Log Barrier

$$p_{\text{barrier}}(\mathbf{x}) = -\sum_i \begin{cases} \log(-g_i(\mathbf{x})) & \text{if } g_i(\mathbf{x}) \geq -1 \\ 0 & \text{else.} \end{cases}$$

- The approach is as before, one transforms the problem to an unconstrained one and increases ρ until convergence:

$$\min_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{\rho} p_{\text{barrier}}(\mathbf{x})$$

2.5.3 Examples

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

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 [9]: 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)

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

```
f_1([1.234, 5.678])
f_2([0.878739, 5.55137])
f_3([0.826216, 5.0439])
f_4([0.473944, 4.07677])
f_5([0.353898, 3.03085])
f_6([0.333873, 1.97179])
f_7([0.333334, 1.04509])
f_8([0.333334, 0.469503])
f_9([0.333333, 0.305792])
f_10([0.333333, 0.296322])
f_11([0.333333, 0.296296])
got 0.5443310477213124 at [0.333333, 0.296296] after 11 iterations (returned XTOL_REACHED)


WARNING: using NLopt.optimize! in module Main conflicts with an existing identifier.
```

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 [9]: 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])

Out[9]: (0.07588358473630112, [0.724702, 0.524221], :FTOL_REACHED)
```

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 specific solver to be used.
- JuMP makes it very easy to swap out solver backends - This is very valuable!

```
In [18]: using JuMP
         using GLPK
         model = Model(with_optimizer(GLPK.Optimizer))
         @variable(model, 0 <= x <= 2)
         @variable(model, 0 <= y <= 30)
         # next, we set an objective function
         @objective(model, Max, 5x + 3 * y)

         # maybe add a constraint called "con":
         @constraint(model, con, 1x + 5y <= 3);
```

- At this stage JuMP has a mathematical representation of our model internalized
- The MathProgBase machinery knows now exactly how to translate that to different solver interfaces
- For us the only thing left: hit the button!

```
In [15]: JuMP.optimize!(model)

         # look at status
         termination_status(model)

Out[15]: OPTIMAL::TerminationStatusCode = 1

In [16]: # we query objective value and solutions
         @show objective_value(model)
         @show value(x)
         @show value(y)

         # as well as the value of the dual variable on the constraint
         @show dual(con);

objective_value(model) = 10.6
value(x) = 2.0
value(y) = 0.2
dual(con) = -0.6
```

- The last call gets the *dual value associated with a constraint*
- Economists most of the time call that the *value of the lagrange multiplier*.

  For linear programs, a feasible dual on a >= constraint is nonnegative and a feasible dual on a <= constraint is nonpositive

- This is different to some textbooks and has nothing to do with wether max or minimizing.

```
In [71]: # helpfully, we have this, which is always positive:
         shadow_price(con)
```

```
Out[71]: 0.6
```

2.10 JuMP handles...

- linear programming
- mixed-integer programming
- second-order conic programming
- semidefinite programming, and
- nonlinear programming

```
In [17]: # JuMP: nonlinear Rosenbrock Example
         # Instead of hand-coding first and second derivatives, you only have to give `JuMP` es
         # Here is an example.

         using Ipopt

         let

             m = Model(with_optimizer(Ipopt.Optimizer))

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

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

             JuMP.optimize!(m)
             @show value(x)
             @show value(y)
             @show termination_status(m)

         end
```

```
This is Ipopt version 3.12.10, 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)   =      0.006
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.
value(x) = 0.9999999999999899
value(y) = 0.9999999999999792
termination_status(m) = LOCALLY_SOLVED::TerminationStatusCode = 4


Out[17]: LOCALLY_SOLVED::TerminationStatusCode = 4
```

In [18]:
```
# not bad, right?
# adding the constraint from before:

let

    m = Model(with_optimizer(Ipopt.Optimizer))

    @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)

    JuMP.optimize!(m)
    @show value(x)
    @show value(y)
    @show termination_status(m)

end
```

```
This is Ipopt version 3.12.10, 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)   =      0.007
Total CPU secs in NLP function evaluations           =      0.106

EXIT: Optimal Solution Found.
value(x) = 0.7247018392092258
value(y) = 0.5242206029480763
termination_status(m) = LOCALLY_SOLVED::TerminationStatusCode = 4


Out[18]: LOCALLY_SOLVED::TerminationStatusCode = 4
```

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

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

- 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 [5]:
```
#  Copyright 2015, Iain Dunning, Joey Huchette, Miles Lubin, and contributors
#  example modified
using Distributions

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

    data = rand(distrib,n);

    m = Model(with_optimizer(Ipopt.Optimizer))

    @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*s

    JuMP.optimize!(m)
    @show termination_status(m)
    println(" = ", value(mu),", mean(data) = ", mean(data))
    println("^2 = ", value(sigma)^2, ", var(data) = ", var(data))
end
```

```
This is Ipopt version 3.12.10, 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  1.7105531e+05 0.00e+00 1.01e+02  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.6780081e+05 0.00e+00 9.69e+01  -1.0 1.03e-02   4.0 1.00e+00 1.00e+00f  1
   2  1.5836108e+05 0.00e+00 8.80e+01  -1.0 3.19e-02   3.5 1.00e+00 1.00e+00f  1
   3  1.3311821e+05 0.00e+00 6.55e+01  -1.0 1.04e-01   3.0 1.00e+00 1.00e+00f  1
   4  8.4962333e+04 0.00e+00 2.87e+01  -1.0 3.41e-01   2.6 1.00e+00 1.00e+00f  1
   5  5.6465222e+04 0.00e+00 1.16e+01  -1.0 4.68e-01   2.1 1.00e+00 1.00e+00f  1
   6  4.1974321e+04 0.00e+00 4.74e+00  -1.0 5.60e-01   1.6 1.00e+00 1.00e+00f  1
   7  3.4160459e+04 0.00e+00 1.64e+00  -1.0 7.26e-01   1.1 1.00e+00 1.00e+00f  1
   8  3.0717018e+04 0.00e+00 6.93e-01  -1.0 7.87e-01   0.7 1.00e+00 1.00e+00f  1
   9  2.9230803e+04 0.00e+00 3.83e-01  -1.7 8.01e-01   0.2 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  2.8385271e+04 0.00e+00 2.69e-01  -1.7 1.17e+00  -0.3 1.00e+00 1.00e+00f  1
  11  2.8124881e+04 0.00e+00 2.62e-01  -1.7 2.03e-01   0.1 1.00e+00 1.00e+00f  1
  12  2.7372459e+04 0.00e+00 2.07e-01  -1.7 6.44e-01  -0.3 1.00e+00 1.00e+00f  1
  13  2.7167859e+04 0.00e+00 1.78e-01  -1.7 1.92e-01   0.1 1.00e+00 1.00e+00f  1
  14  2.6871675e+04 0.00e+00 2.11e-01  -2.5 9.21e-01  -0.4 1.00e+00 1.00e+00f  1
  15  2.6749431e+04 0.00e+00 7.56e-02  -2.5 9.37e-01   0.0 1.00e+00 5.00e-01f  2
  16  2.6736752e+04 0.00e+00 6.73e-02  -2.5 3.04e-02   0.5 1.00e+00 1.00e+00f  1
  17  2.6706137e+04 0.00e+00 4.90e-02  -2.5 1.15e-01  -0.0 1.00e+00 1.00e+00f  1
  18  2.6702789e+04 0.00e+00 4.36e-02  -2.5 2.14e-02   0.4 1.00e+00 1.00e+00f  1
  19  2.6694444e+04 0.00e+00 2.52e-02  -2.5 7.31e-02  -0.1 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  20  2.6693514e+04 0.00e+00 2.19e-02  -3.8 1.26e-02   0.4 1.00e+00 1.00e+00f  1
  21  2.6691281e+04 0.00e+00 1.02e-02  -3.8 4.46e-02  -0.1 1.00e+00 1.00e+00f  1
  22  2.6691103e+04 0.00e+00 8.65e-03  -3.8 5.83e-03   0.3 1.00e+00 1.00e+00f  1
  23  2.6690726e+04 0.00e+00 3.14e-03  -3.8 2.12e-02  -0.2 1.00e+00 1.00e+00f  1
  24  2.6690695e+04 0.00e+00 2.59e-03  -3.8 2.07e-03   0.3 1.00e+00 1.00e+00f  1
  25  2.6690680e+04 0.00e+00 2.07e-03  -3.8 2.19e-02  -0.2 1.00e+00 2.50e-01f  3
  26  2.6690666e+04 0.00e+00 1.66e-03  -3.8 1.56e-03   0.2 1.00e+00 1.00e+00f  1
  27  2.6690664e+04 0.00e+00 1.64e-03  -3.8 1.17e-01  -0.3 1.00e+00 1.56e-02f  7
  28  2.6690658e+04 0.00e+00 1.26e-03  -3.8 1.43e-03   0.1 1.00e+00 1.00e+00f  1
  29  2.6690656e+04 0.00e+00 1.17e-03  -5.7 3.62e-04   0.6 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  30  2.6690653e+04 0.00e+00 8.61e-04  -5.7 1.18e-03   0.1 1.00e+00 1.00e+00f  1
  31  2.6690652e+04 0.00e+00 7.88e-04  -5.7 2.80e-04   0.5 1.00e+00 1.00e+00f  1
  32  2.6690650e+04 0.00e+00 5.46e-04  -5.7 9.26e-04   0.0 1.00e+00 1.00e+00f  1
  33  2.6690650e+04 0.00e+00 4.93e-04  -5.7 2.02e-04   0.5 1.00e+00 1.00e+00f  1
  34  2.6690649e+04 0.00e+00 3.16e-04  -5.7 6.78e-04  -0.0 1.00e+00 1.00e+00f  1
  35  2.6690649e+04 0.00e+00 2.81e-04  -5.7 1.33e-04   0.4 1.00e+00 1.00e+00f  1
  36  2.6690648e+04 0.00e+00 1.62e-04  -5.7 4.55e-04  -0.1 1.00e+00 1.00e+00f  1
```

```
  37  2.6690648e+04 0.00e+00 1.42e-04  -5.7 7.77e-05   0.4 1.00e+00 1.00e+00f  1
  38  2.6690648e+04 0.00e+00 7.03e-05  -5.7 2.72e-04  -0.1 1.00e+00 1.00e+00f  1
  39  2.6690648e+04 0.00e+00 6.02e-05  -5.7 3.86e-05   0.3 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  40  2.6690648e+04 0.00e+00 2.38e-05  -5.7 1.39e-04  -0.2 1.00e+00 1.00e+00f  1
  41  2.6690648e+04 0.00e+00 1.99e-05  -5.7 1.50e-05   0.3 1.00e+00 1.00e+00f  1
  42  2.6690648e+04 0.00e+00 5.28e-06  -5.7 5.60e-05  -0.2 1.00e+00 1.00e+00f  1
  43  2.6690648e+04 0.00e+00 4.28e-06  -8.6 3.81e-06   0.2 1.00e+00 1.00e+00f  1
  44  2.6690648e+04 0.00e+00 3.25e-06  -8.6 1.49e-05  -0.3 1.00e+00 5.00e-01f  2
  45  2.6690648e+04 0.00e+00 1.82e-06  -8.6 3.46e-06   0.2 1.00e+00 1.00e+00f  1
  46  2.6690648e+04 0.00e+00 1.69e-06  -8.6 5.01e-07   0.6 1.00e+00 1.00e+00f  1
  47  2.6690648e+04 0.00e+00 1.27e-06  -8.6 1.63e-06   0.1 1.00e+00 1.00e+00f  1
  48  2.6690648e+04 0.00e+00 1.16e-06  -8.6 3.95e-07   0.5 1.00e+00 1.00e+00f  1
  49  2.6690648e+04 0.00e+00 8.23e-07  -8.6 1.30e-06   0.1 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  50  2.6690648e+04 0.00e+00 7.47e-07  -8.6 2.92e-07   0.5 1.00e+00 1.00e+00f  1
  51  2.6690648e+04 0.00e+00 4.92e-07  -8.6 9.75e-07   0.0 1.00e+00 1.00e+00f  1
  52  2.6690648e+04 0.00e+00 4.40e-07  -8.6 1.98e-07   0.4 1.00e+00 1.00e+00f  1
  53  2.6690648e+04 0.00e+00 2.63e-07  -8.6 6.75e-07  -0.0 1.00e+00 1.00e+00f  1
  54  2.6690648e+04 0.00e+00 2.32e-07  -8.6 1.21e-07   0.4 1.00e+00 1.00e+00f  1
  55  2.6690648e+04 0.00e+00 1.22e-07  -8.6 4.21e-07  -0.1 1.00e+00 1.00e+00f  1
  56  2.6690648e+04 0.00e+00 1.05e-07  -8.6 6.39e-08   0.3 1.00e+00 1.00e+00f  1
  57  2.6690648e+04 0.00e+00 4.53e-08  -8.6 2.28e-07  -0.1 1.00e+00 1.00e+00f  1
  58  2.6690648e+04 0.00e+00 3.82e-08  -8.6 2.72e-08   0.3 1.00e+00 1.00e+00f  1
  59  2.6690648e+04 0.00e+00 1.19e-08  -8.6 1.00e-07  -0.2 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  60  2.6690648e+04 0.00e+00 9.76e-09  -9.0 8.21e-09   0.2 1.00e+00 1.00e+00f  1

Number of Iterations...: 60

                                   (scaled)                 (unscaled)
Objective...:   8.5074714157088174e+00    2.6690648196634054e+04
Dual infeasibility...:   9.7569600551044561e-09    3.0610692128632544e-05
Constraint violation...:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity...:   9.0909090949984745e-10    2.8521078071934833e-06
Overall NLP error...:   9.7569600551044561e-09    3.0610692128632544e-05


Number of objective function evaluations             = 87
Number of objective gradient evaluations             = 61
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             = 60
Total CPU secs in IPOPT (w/o function evaluations)   =      0.046
Total CPU secs in NLP function evaluations           =      0.169
```

```
EXIT: Optimal Solution Found.
termination_status(m) = LOCALLY_SOLVED::TerminationStatusCode = 4
 =

WARNING: using Distributions.mode in module Main conflicts with an existing identifier.


4.493062898788303, mean(data) = 4.493062936089179
^2 = 12.185571328763903, var(data) = 12.186789996356879
```

3 Linear Constrained Problems (LPs)

- Very similar to before, just that both objective and constraints are *linear*.

$$\begin{aligned} \min_{\mathbf{x}} & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & \mathbf{w}_{LE}^{(i)T}\mathbf{x} \leq b_i \text{ for } i \in 1,2,3,\ldots \\ & \mathbf{w}_{GE}^{(j)T}\mathbf{x} \geq b_j \text{ for } j \in 1,2,3,\ldots \\ & \mathbf{w}_{EQ}^{(k)T}\mathbf{x} = b_k \text{ for } k \in 1,2,3,\ldots \end{aligned}$$

- Our initial JuMP example was of that sort.

3.0.1 Standard Form

- Usually LPs are given in *standard form*
- All constraints are less-than inequalities
- All choice variables are non-negative.

$$\begin{aligned} \min_{\mathbf{x}} & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & \mathbf{A}\mathbf{x} \leq b \\ & \mathbf{x} \geq 0 \end{aligned}$$

- Greater-than inequality constraints are inverted
- equality constraints are split into two
- $\mathbf{x} = \mathbf{x}^+ - \mathbf{x}^-$ and we constrain both components to be positive.

3.0.2 Equality Form

$$\begin{aligned} \min_{\mathbf{x}} & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & \mathbf{A}\mathbf{x} = b \\ & \mathbf{x} \geq 0 \end{aligned}$$

- Can transform standard into equality form

$$\mathbf{Ax} \leq b \rightarrow \mathbf{Ax} + \mathbf{s} = b, \mathbf{s} \geq 0$$

- equality constraints are split into two
- $\mathbf{x} = \mathbf{x}^+ - \mathbf{x}^-$ and we constrain both components to be positive.

3.0.3 Solving LPs

- Simplex Algorithm operates on Equality Form
- Moving from one vertex to the next of the feasible set, this is guaranteed to find the optimal solution if the problem is bounded.

3.1 A Cannery Problem

- A can factory (a cannery) has plants in Seattle and San Diego
- They need to decide how to serve markets New-York, Chicago, Topeka
- Firm wants to

  1. minimize shipping costs
  2. shipments cannot exceed capacity
  3. shipments must satisfy demand

- Formalize that!
- Plant capacity cap_i, demands d_j and transport costs from plant i to market j, $dist_{i,j}c$ are all given.
- Let $\mathbf{x}$ be a matrix with element $x_{i,j}$ for number of cans shipped from i to j.

3.2 From Maths ...

$$\min_{\mathbf{x}} \sum_{i=1}^{2} \sum_{j=1}^{3} dist_{i,j}c \times x_{i,j}$$

$$\text{subject to } \sum_{j=1}^{3} x(i,j) \leq cap_i, \forall i$$

$$\sum_{i=1}^{2} x(i,j) \geq d_j, \forall j$$

```
In [7]: # ... to JuMP
        # https://github.com/JuliaOpt/JuMP.jl/blob/release-0.19/examples/cannery.jl
        #  Copyright 2017, Iain Dunning, Joey Huchette, Miles Lubin, and contributors
        #  This Source Code Form is subject to the terms of the Mozilla Public
        #  License, v. 2.0. If a copy of the MPL was not distributed with this
        #  file, You can obtain one at http://mozilla.org/MPL/2.0/.
        ##############################################################################
        # JuMP
        # An algebraic modeling language for Julia
        # See http://github.com/JuliaOpt/JuMP.jl
        ##############################################################################
```

```
using JuMP, GLPK, Test
const MOI = JuMP.MathOptInterface

"""
    example_cannery(; verbose = true)
JuMP implementation of the cannery problem from Dantzig, Linear Programming and
Extensions, Princeton University Press, Princeton, NJ, 1963.
Author: Louis Luangkesorn
Date: January 30, 2015
"""
function example_cannery(; verbose = true)
    plants = ["Seattle", "San-Diego"]
    markets = ["New-York", "Chicago", "Topeka"]

    # Capacity and demand in cases.
    capacity = [350, 600]
    demand = [300, 300, 300]

    # Distance in thousand miles.
    distance = [2.5 1.7 1.8; 2.5 1.8 1.4]

    # Cost per case per thousand miles.
    freight = 90

    num_plants = length(plants)
    num_markets = length(markets)

    cannery = Model(with_optimizer(GLPK.Optimizer))

    @variable(cannery, ship[1:num_plants, 1:num_markets] >= 0)

    # Ship no more than plant capacity
    @constraint(cannery, capacity_con[i in 1:num_plants],
        sum(ship[i,j] for j in 1:num_markets) <= capacity[i]
    )

    # Ship at least market demand
    @constraint(cannery, demand_con[j in 1:num_markets],
        sum(ship[i,j] for i in 1:num_plants) >= demand[j]
    )

    # Minimize transporatation cost
    @objective(cannery, Min, sum(distance[i, j] * freight * ship[i, j]
        for i in 1:num_plants, j in 1:num_markets)
    )

    JuMP.optimize!(cannery)
```

```
            if verbose
                println("RESULTS:")
                for i in 1:num_plants
                    for j in 1:num_markets
                        println("  $(plants[i]) $(markets[j]) = $(JuMP.value(ship[i, j]))")
                    end
                end
            end

            @assert JuMP.termination_status(cannery) == MOI.OPTIMAL
            @assert JuMP.primal_status(cannery) == MOI.FEASIBLE_POINT
            @assert JuMP.objective_value(cannery) == 151200.0
        end
        example_cannery()
```

```
RESULTS:
  Seattle New-York = 50.0
  Seattle Chicago = 300.0
  Seattle Topeka = 0.0
  San-Diego New-York = 250.0
  San-Diego Chicago = 0.0
  San-Diego Topeka = 300.0
```

4 Discrete Optimization / Integer Programming

- Here the choice variable is contrained to come from a discrete set $\mathcal{X}$.
- If this set is $\mathcal{X} = \mathbb{N}$, we have an **integer program**
- If only *some* x have to be discrete, this is a **mixed integer program**

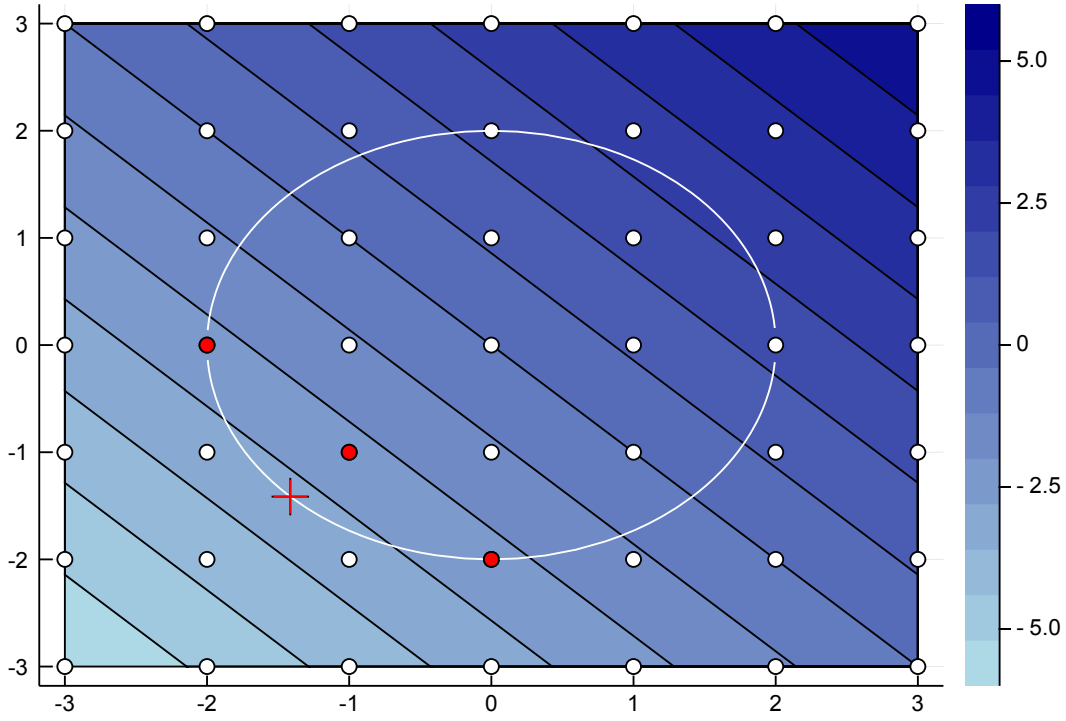
4.1 Example

$$\begin{aligned} &\min_{\mathbf{x}} x_1 + x_2 \\ &\text{subject to } ||\mathbf{x}|| \leq 2 \\ &\mathbf{x} \in \mathbb{N} \end{aligned}$$

- continuous optimum is $(-\sqrt{2}, -\sqrt{2})$ and objective is $y = -2\sqrt{2} = -2.828$
- Integer constrained problem is only delivering $y = -2$, and $\mathbf{x}^* \in (-2, 0), (-1, -1), (0, -2)$

```
In [8]: x = -3:0.01:3
        dx = repeat(range(-3,stop = 3, length = 7),1,7)
        contourf(x,x,(x,y)->x+y,color=:blues)
        scatter!(dx,dx',legend=false,markercolor=:white)
        plot!(x->sqrt(4-x^2),-2,2,c=:white)
        plot!(x->-sqrt(4-x^2),-2,2,c=:white)
        scatter!([-2,-1,0],[0,-1,-2],c=:red)
        scatter!([-sqrt(2)],[-sqrt(2)],c=:red,markershape=:cross,markersize=9)
```

Out[8]:



4.2 Rounding

- One solution is to just *round the continuous solution to the nearest integer*
- We compute the **relaxed** problem, i.e. the one where x is continuous.
- Then we round up or down.
- Can go terribly wrong.

4.3 Cutting Planes

- This is an exact method
- We solve the relaxed problem first.
- Then we add linear constraints that result in the solution becoming integral.

4.4 Branch and Bound

- This enumerates all possible soultions.
- Branch and bound does this, without having to compute all of them.

4.5 Example: The Knapsack Problem

- We are packing our knapsack for a trip but only have space for the most valuable items.
- We have $x_i = 0$ if item i is not in the sack, 1 else.

$$\min_x -\sum_{i=1}^{n} v_i x_i$$
$$\text{s.t. } \sum_{i=1}^{n} w_i x_i \leq w_{max}$$
$$w_i \in \mathbb{N}_+, v_i \in \mathbb{R}$$

- If ther are n items, we have 2^n possible design vectors.
- But there is a useful recursive relationship.
- If we solved $n-1$ knapsack problems so far and deliberate about item n
  - If it's not worth including item n, then the solution **is** the knapsack problem for $n-1$ items and capacity $w_{\max}$
  - If it IS worth including it: solution will have value of knapsack with $n-1$ items and reduced capacity, plus the value of the new item
- This **is** dynamic progamming.

4.5.1 Knacksack Recursion

- In particular, the recursion looks like this:

$$\text{knapsack}\left(i, w_{\max}\right) = \begin{cases} 0 & \text{if} i = 0 \\ \text{knapsack}\left(i-1, w_{\max}\right) & \text{if} w_i > w_{\max} \\ \max \begin{cases} \text{knapsack}\left(i-1, w_{\max}\right) & \text{(discard new item)} \\ \text{knapsack}\left(i-1, w_{\max}-w_i\right)+v_i & \text{(include new item)} \end{cases} & \text{else.} \end{cases}$$

```
In [6]: #  Copyright 2017, Iain Dunning, Joey Huchette, Miles Lubin, and contributors
        #  This Source Code Form is subject to the terms of the Mozilla Public
        #  License, v. 2.0. If a copy of the MPL was not distributed with this
        #  file, You can obtain one at http://mozilla.org/MPL/2.0/.
        ##############################################################################
        # JuMP
        # An algebraic modeling langauge for Julia
        # See http://github.com/JuliaOpt/JuMP.jl
        ##############################################################################
        # knapsack.jl
        #
        # Solves a simple knapsack problem:
        # max sum(p_j x_j)
        #  st sum(w_j x_j) <= C
        #     x binary
        ##############################################################################

        using JuMP, Cbc, LinearAlgebra
```

```
    let

        # Maximization problem
        m = Model(with_optimizer(Cbc.Optimizer))

        @variable(m, x[1:5], Bin)

        profit = [ 5, 3, 2, 7, 4 ]
        weight = [ 2, 8, 4, 2, 5 ]
        capacity = 10

        # Objective: maximize profit
        @objective(m, Max, dot(profit, x))

        # Constraint: can carry all
        @constraint(m, dot(weight, x) <= capacity)

        # Solve problem using MIP solver
        optimize!(m)

        println("Objective is: ", JuMP.objective_value(m))
        println("Solution is:")
        for i = 1:5
            print("x[$i] = ", JuMP.value(x[i]))
            println(", p[$i]/w[$i] = ", profit[i]/weight[i])
        end
    end
```

```
Objective is: 16.0
Solution is:
x[1] = 1.0, p[1]/w[1] = 2.5
x[2] = 0.0, p[2]/w[2] = 0.375
x[3] = 0.0, p[3]/w[3] = 0.5
x[4] = 1.0, p[4]/w[4] = 3.5
x[5] = 1.0, p[5]/w[5] = 0.8
Welcome to the CBC MILP Solver
Version: 2.9.9
Build Date: Dec 31 2018

command line - Cbc_C_Interface -solve -quit (default strategy 1)
Continuous objective value is 16.5 - 0.00 seconds
Cgl0004I processed model has 1 rows, 5 columns (5 integer (5 of which binary)) and 5 elements
Cutoff increment increased from 1e-05 to 0.9999
Cbc0038I Initial state - 1 integers unsatisfied sum - 0.25
Cbc0038I Solution found of -16
Cbc0038I Before mini branch and bound, 4 integers at bound fixed and 0 continuous
Cbc0038I Mini branch and bound did not improve solution (0.00 seconds)
Cbc0038I After 0.00 seconds - Feasibility pump exiting with objective of -16 - took 0.00 secon
```

```
Cbc0012I Integer solution of -16 found by feasibility pump after 0 iterations and 0 nodes (0.0
Cbc0001I Search completed - best objective -16, took 1 iterations and 0 nodes (0.00 seconds)
Cbc0035I Maximum depth 0, 4 variables fixed on reduced cost
Cuts at root node changed objective from -16.5 to -16
Probing was tried 0 times and created 0 cuts of which 0 were active after adding rounds of cuts
Gomory was tried 0 times and created 0 cuts of which 0 were active after adding rounds of cuts
Knapsack was tried 0 times and created 0 cuts of which 0 were active after adding rounds of cu
Clique was tried 0 times and created 0 cuts of which 0 were active after adding rounds of cuts
MixedIntegerRounding2 was tried 0 times and created 0 cuts of which 0 were active after adding
FlowCover was tried 0 times and created 0 cuts of which 0 were active after adding rounds of cu
TwoMirCuts was tried 0 times and created 0 cuts of which 0 were active after adding rounds of

Result - Optimal solution found

Objective value:                16.00000000
Enumerated nodes:               0
Total iterations:               1
Time (CPU seconds):             0.00
Time (Wallclock seconds):       0.02

Total time (CPU seconds):       0.00   (Wallclock seconds):       0.03
```

In [ ]: