Optimization

September 11, 2018

1 Optimization

1.1 Local Nonlinear Optimization with Optim.jl

One of the core libraries for nonlinear optimization is Optim.jl. Optim.jl is a lot like the standard optimizers you'd find in SciPy or MATLAB. You give it a function and it finds the minimum. For example, if you give it a univariate function it uses Brent's method to find the minimum in an interval:

```
In [6]: using Optim
        f(x) = sin(x)+cos(x)
        Optim.optimize(f,0.0,2) # Find a minimum between 0 and 2
```

```
Out[6]: Results of Optimization Algorithm
         * Algorithm: Brent's Method
         * Search Interval: [0.000000, 6.283185]
         * Minimizer: 3.926991e+00
         * Minimum: -1.414214e+00
         * Iterations: 11
         * Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08*|x|+2.2e-16): true
         * Objective Function Calls: 12
```

If you give it a function which requires vector input with scalar output, it will give the vector local minima:

```
In [7]: f(x) = sin(x[1])+cos(x[1]+x[2])
        Optim.optimize(f,zeros(2)) # Find a minimum starting at [0.0,0.0]
```

```
Out[7]: Results of Optimization Algorithm
         * Algorithm: Nelder-Mead
         * Starting Point: [0.0,0.0]
         * Minimizer: [-1.570684758073873,-1.5708688186478836]
         * Minimum: -2.000000e+00
         * Iterations: 49
         * Convergence: true
           *  ((y-y)š)/n < 1.0e-08: true
           * Reached Maximum Number of Iterations: false
         * Objective Calls: 95
```

You can refer to Optim's large library of methods and pass in a method choice to have different properties. Let's choose BFGS:

```
In [8]: Optim.optimize(f,zeros(2),BFGS())
```

```
Out[8]: Results of Optimization Algorithm
         * Algorithm: BFGS
         * Starting Point: [0.0,0.0]
         * Minimizer: [-1.5707963314270867,-1.5707963181008544]
         * Minimum: -2.000000e+00
         * Iterations: 6
         * Convergence: true
           * |x - x'|  0.0e+00: false
             |x - x'| = 4.10e-04
           * |f(x) - f(x')|  0.0e+00 |f(x)|: false
             |f(x) - f(x')| = -9.67e-08 |f(x)|
           * |g(x)|  1.0e-08: true
             |g(x)| = 4.06e-09
           * Stopped by an increasing objective: false
           * Reached Maximum Number of Iterations: false
         * Objective Calls: 15
         * Gradient Calls: 15
```

1.2 Global Nonlinear Optimization with BlackBoxOptim.jl

Global optimization is provided with a native Julia implementation at BlackBoxOptim.jl. You have to give it box constraints and tell it the size of the input vector:

```
In [9]: using BlackBoxOptim

        function rosenbrock2d(x)
          return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
        end
        res = bboptimize(rosenbrock2d; SearchRange = (-5.0, 5.0), NumDimensions = 2)
```

```
Starting optimization with optimizer DiffEvoOpt{FitPopulation{Float64},RadiusLimitedSelector,B
0.00 secs, 0 evals, 0 steps

Optimization stopped after 10001 steps and 0.013683080673217773 seconds
Termination reason: Max number of steps (10000) reached
Steps per second = 730902.6555383248
Function evals per second = 738722.5319649422
Improvements/step = 0.2172
Total function evaluations = 10108


Best candidate found: [1.0, 1.0]

Fitness: 0.000000000
```

```
Out[9]: BlackBoxOptim.OptimizationResults("adaptive_de_rand_1_bin_radiuslimited", "Max number
```

1.3 JuMP, Convex.jl, NLopt.jl

JuMP.jl is a large library for all sorts of optimization problems. It has solvers for linear, quadratic, etc. programming problems. If you're not doing nonlinear optimization JuMP is a great choice. If you're looking to do convex programming, Convex.jl is a library with methods specific for this purpose. If you want to do nonlinear optimization with constraints, NLopt.jl is a library with a large set of choices. It also has a bunch of derivative-free local optimization methods. It's only issue is that its an interface to a C library and can be more difficult to debug than the native Julia codes, but otherwise it's a great alternative to Optim and BlackBoxOptim.

1.4 Problem 1

Use Optim.jl to optimize Hosaki's Function. Use the initial condition `[2.0,2.0]`.

1.5 Problem 2

BlackBoxOptim.jl to find global minima of the Adjiman Function with $-1 < x_1 < 2$ and $-1 < x_2 < 1$.