Learn GalacticOptim and Optim

In [119]:
```
using GalacticOptim, Optim
```

In [120]:
```
rosenbrock(x,p) =  (  - x[1])^2 + p[2] * (x[2] - x[1]^2)^2
```

Out[120]:
```
rosenbrock (generic function with 3 methods)
```

In [121]:
```
x0 = zeros(2)
```

Out[121]:
```
2-element Vector{Float64}:
 0.0
 0.0
```

In [122]:
```
p  = [1.0,100.0]
```

Out[122]:
```
2-element Vector{Float64}:
   1.0
 100.0
```

In [123]:
```
prob = OptimizationProblem(rosenbrock,x0,p)
```

Out[123]:
```
OptimizationProblem. In-place: true
u0: [0.0, 0.0]
```

In [124]:
```
sol = solve(prob,NelderMead())
```

Out[124]:
```
u: 2-element Vector{Float64}:
 0.9999634355313174
 0.9999315506115275
```

In [125]:
```
using BlackBoxOptim
```

In [126]:
```
prob = OptimizationProblem(rosenbrock, x0, p, lb = [-1.0,-1.0], ub = [1.0,1.0])
```

Out[126]:
```
OptimizationProblem. In-place: true
u0: [0.0, 0.0]
```

In [127]:
```
sol = solve(prob,BBO())
```

```
Starting optimization with optimizer DiffEvoOpt{FitPopulation{Float64}, RadiusLimitedSelector, BlackBoxOptim.AdaptiveDiffE
voRandBin{3}, RandomBound{ContinuousRectSearchSpace}}
0.00 secs, 0 evals, 0 steps

Optimization stopped after 10001 steps and 0.04 seconds
Termination reason: Max number of steps (10000) reached
Steps per second = 256435.67
Function evals per second = 259025.41
Improvements/step = 0.21690
Total function evaluations = 10102


Best candidate found: [1.0, 1.0]

Fitness: 0.000000000
```

Out[127]:
```
u: 2-element Vector{Float64}:
 0.9999999999999996
 0.9999999999999992
```

In [129]:
```
f2 = OptimizationFunction(rosenbrock, GalacticOptim.AutoForwardDiff())
```

Out[129]:
```
OptimizationFunction{true, GalacticOptim.AutoForwardDiff{nothing}, typeof(rosenbrock), Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing}(rosenbrock, GalacticOptim.AutoForwardDiff{nothing}(), nothing, nothing, nothing, nothing, nothing, nothi
ng)
```

In [130]:
```
prob = OptimizationProblem(f2, x0, p)
```
Out[130]:
```
OptimizationProblem. In-place: true
u0: [0.0, 0.0]
```
In [131]:
```
sol = solve(prob,BFGS())
```
Out[131]:
```
u: 2-element Vector{Float64}:
 0.9999999999373614
 0.999999999868622
```
In [132]:
```
prob = OptimizationProblem(f2, x0, p, lb = [-1.0,-1.0], ub = [1.0,1.0])
```
Out[132]:
```
OptimizationProblem. In-place: true
u0: [0.0, 0.0]
```
In [133]:
```
sol = solve(prob, Fminbox(GradientDescent()))
```
Out[133]:
```
u: 2-element Vector{Float64}:
 0.9999999899064821
 0.9999999797630695
```
Rosenbrock function examples

In [134]:
```
using GalacticOptim, Optim, Test, Random
```
In [135]:
```
rosenbrock(x, p) =  (p[1] - x[1])^2 + p[2] * (x[2] - x[1]^2)^2
x0 = zeros(2)
_p  = [1.0, 100.0]
```
Out[135]:
```
2-element Vector{Float64}:
   1.0
 100.0
```
In [136]:
```
f3 = OptimizationFunction(rosenbrock, GalacticOptim.AutoForwardDiff())
```
Out[136]:
```
OptimizationFunction{true, GalacticOptim.AutoForwardDiff{nothing}, typeof(rosenbrock), Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}(rosenbrock, GalacticOptim.AutoForwardDiff{nothing}(), nothing, nothing, nothing, nothing, nothing, nothi
ng)
```
In [137]:
```
l1 = rosenbrock(x0, _p)
prob = OptimizationProblem(f3, x0, _p)
sol = solve(prob, SimulatedAnnealing())
```
Out[137]:
```
u: 2-element Vector{Float64}:
 0.9339883554832432
 0.8673154506302181
```
In [138]:
```
@test 10*sol.minimum < l1
```
Out[138]:
```
Test Passed
```
In [143]:
```
Random.seed!(1234)
prob = OptimizationProblem(f3, x0, _p, lb=[-1.0, -1.0], ub=[0.8, 0.8])
sol = solve(prob, SAMIN())
```
```
================================================================================
SAMIN results
NO CONVERGENCE: MAXEVALS exceeded

     Obj. value:           0.05281

       parameter      search width
         0.77067           0.35556
         0.59542           0.15000
================================================================================
```
Out[143]:
```
u: 2-element Vector{Float64}:
 0.7706700409293477
 0.5954190179306745
```

In [144]:
```
@test 10*sol.minimum < l1
```
Out[144]: Test Passed

In [145]:
```
using CMAEvolutionStrategy
sol = solve(prob, CMAEvolutionStrategyOpt())
```
```
(3_w,6)-aCMA-ES (mu_w=2.0,w_1=64%) in dimension 2 (seed=9180451354441353971, 2021-03-27T19:05:55.007)

     1        6    1.60109981e+00   9.58e-02   1.253e+00     0.109
     2       12    1.50796511e+00   7.95e-02   1.245e+00     0.109
     3       18    1.40892846e+00   6.89e-02   1.340e+00     0.109
    89      534    4.00000000e-02   2.66e-04   8.403e+00     0.109
(3_w,6)-aCMA-ES (mu_w=2.0,w_1=64%) in dimension 2 (seed=9180451354441353971, 2021-03-27T19:05:55.007)
  termination reason: ftol = 1.0e-11 (2021-03-27T19:05:55.007)
  lowest observed function value: 0.040000000000344295 at [0.7999999999947122, 0.6399999901161894]
  population mean: [0.7999999999993345, 0.6400000261357587]
```
Out[145]:
```
u: 2-element Vector{Float64}:
 0.8900013797245124
 0.6399999901161894
```

In [146]:
```
@test 10*sol.minimum < l1
```
Out[146]: Test Passed

In [147]:
```
rosenbrock(x, p=nothing) =  (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
```
Out[147]: rosenbrock (generic function with 3 methods)

In [148]:
```
l1 = rosenbrock(x0)
prob = OptimizationProblem(rosenbrock, x0)
sol = solve(prob, NelderMead())
```
Out[148]:
```
u: 2-element Vector{Float64}:
 0.9999634355313174
 0.9999315506115275
```

In [149]:
```
@test 10*sol.minimum < l1
```
Out[149]: Test Passed

In [151]:
```
cons= (x,p) -> [x[1]^2 + x[2]^2]
optprob = OptimizationFunction(rosenbrock, GalacticOptim.AutoForwardDiff();cons= cons)
prob = OptimizationProblem(optprob, x0)
sol = solve(prob, ADAM(0.1), maxiters = 1000)
```
Out[151]:
```
u: 2-element Vector{Float64}:
 0.9985769782748619
 0.9971971594653897
```

In [152]:
```
@test 10*sol.minimum < l1
```
Out[152]: Test Passed

In [153]:
```
sol = solve(prob, BFGS())
```
Out[153]:
```
u: 2-element Vector{Float64}:
 0.9999999999373614
 0.999999999868622
```

In [154]:
```
@test 10*sol.minimum < l1
```
Out[154]: Test Passed

In [155]:
```
sol = solve(prob, Newton())
```
Out[155]:
```
u: 2-element Vector{Float64}:
 0.9999999999999994
 0.9999999999999989
```

In [156]:
```
@test 10*sol.minimum < l1
```
Out[156]: Test Passed

In [157]:
```
sol = solve(prob, Optim.KrylovTrustRegion())
```
Out[157]:
```
u: 2-element Vector{Float64}:
 0.999999999999108
 0.9999999999981819
```
In [158]:
```
@test 10*sol.minimum < 11
```
Out[158]:
```
Test Passed
```
In [159]:
```
prob = OptimizationProblem(optprob, x0, lcons = [-Inf], ucons = [Inf])
sol = solve(prob, IPNewton())
```
Out[159]:
```
u: 2-element Vector{Float64}:
 0.9999999992619217
 0.9999999985003628
```
In [160]:
```
@test 10*sol.minimum < 11
```
Out[160]:
```
Test Passed
```
In [161]:
```
prob = OptimizationProblem(optprob, x0, lcons = [-5.0], ucons = [10.0])
sol = solve(prob, IPNewton())
```
Out[161]:
```
u: 2-element Vector{Float64}:
 0.9999999992669327
 0.9999999985109471
```
In [162]:
```
@test 10*sol.minimum < 11
```
Out[162]:
```
Test Passed
```
In [163]:
```
prob = OptimizationProblem(optprob, x0, lcons = [-Inf], ucons = [Inf], lb = [-500.0,-500.0], ub=[50.0,50.0])
sol = solve(prob, IPNewton())
```
Out[163]:
```
u: 2-element Vector{Float64}:
 0.9999999992541948
 0.9999999984843432
```
In [164]:
```
@test sol.minimum < 11
```
Out[164]:
```
Test Passed
```
In [166]:
```
function con2_c(x,p)
    [x[1]^2 + x[2]^2, x[2]*sin(x[1])-x[1]]
end
optprob = OptimizationFunction(rosenbrock, GalacticOptim.AutoForwardDiff();cons= con2_c)
prob = OptimizationProblem(optprob, x0, lcons = [-Inf,-Inf], ucons = [Inf,Inf])
sol = solve(prob, IPNewton())
```
Out[166]:
```
u: 2-element Vector{Float64}:
 0.9999999992619217
 0.9999999985003628
```
In [168]:
```
cons_circ = (x,p) -> [x[1]^2 + x[2]^2]
optprob = OptimizationFunction(rosenbrock, GalacticOptim.AutoForwardDiff();cons= cons_circ)
prob = OptimizationProblem(optprob, x0, lcons = [-Inf], ucons = [0.25^2])
sol = solve(prob, IPNewton())
```
Out[168]:
```
u: 2-element Vector{Float64}:
 0.24327905408863862
 0.05757865786675858
```
In [169]:
```
@test sqrt(cons(sol.minimizer,nothing)[1]) ≈ 0.25 rtol = 1e-6
```
Out[169]:
```
Test Passed
```
In [170]:
```
optprob = OptimizationFunction(rosenbrock, GalacticOptim.AutoZygote())
prob = OptimizationProblem(optprob, x0)
sol = solve(prob, ADAM(), maxiters = 1000, progress = false)
```
Out[170]:
```
u: 2-element Vector{Float64}:
 0.7396709479564985
 0.5470724299209024
```
In [171]:
```
@test 10*sol.minimum < 11
```
Out[171]:
```
Test Passed
```

In [172]:
```
prob = OptimizationProblem(optprob, x0, lb=[-1.0, -1.0], ub=[0.8, 0.8])
sol = solve(prob, Fminbox())
```

Out[172]:
```
u: 2-element Vector{Float64}:
 0.799999998888889
 0.6399999982096882
```

In [173]:
```
@test 10*sol.minimum < l1
```

Out[173]: Test Passed

In [174]:
```
prob = OptimizationProblem(optprob, x0, lb=[-1.0, -1.0], ub=[0.8, 0.8])
@test_broken @test_nowarn sol = solve(prob, SAMIN())
```

```
================================================================================
SAMIN results
NO CONVERGENCE: MAXEVALS exceeded

     Obj. value:           0.04776

       parameter      search width
         0.78575           1.80000
         0.62170           1.80000
================================================================================
```

Out[174]:
```
Test Broken
  Expression: #= In[174]:2 =# @test_nowarn sol = solve(prob, SAMIN())
```

In [175]:
```
@test 10*sol.minimum < l1
```

Out[175]: Test Passed

In [176]:
```
using NLopt
prob = OptimizationProblem(optprob, x0)
sol = solve(prob, Opt(:LN_BOBYQA, 2))
```

Out[176]:
```
u: 2-element Vector{Float64}:
 0.9999999999999999
 0.9999999999999998
```

In [177]:
```
@test 10*sol.minimum < l1
```

Out[177]: Test Passed

In [178]:
```
sol = solve(prob, Opt(:LD_LBFGS, 2))
```

Out[178]:
```
u: 2-element Vector{Float64}:
 0.9999999999894374
 0.9999999999844783
```

In [179]:
```
@test 10*sol.minimum < l1
```

Out[179]: Test Passed

In [180]:
```
prob = OptimizationProblem(optprob, x0, lb=[-1.0, -1.0], ub=[0.8, 0.8])
sol = solve(prob, Opt(:LD_LBFGS, 2))
```

Out[180]:
```
u: 2-element Vector{Float64}:
 0.8
 0.6400000000000001
```

In [181]:
```
@test 10*sol.minimum < l1
```

Out[181]: Test Passed

In [182]:
```
sol = solve(prob, Opt(:G_MLSL_LDS, 2), nstart=2, local_method = Opt(:LD_LBFGS, 2), maxiters=10000)
```

Out[182]:
```
u: 2-element Vector{Float64}:
 0.8
 0.6400000000000001
```

In [183]:
```
@test 10*sol.minimum < l1
```

Out[183]: Test Passed

In [185]:
```
Pkg.add("Evolutionary")
```

```
   Resolving package versions...
   Installed Evolutionary ── v0.9.0
    Updating `C:\Users\lishu\.julia\environments\v1.6\Project.toml`
  [86b6b26d] + Evolutionary v0.9.0
    Updating `C:\Users\lishu\.julia\environments\v1.6\Manifest.toml`
  [86b6b26d] + Evolutionary v0.9.0
Precompiling project...
  ✓ Evolutionary
1 dependency successfully precompiled in 8 seconds (172 already precompiled, 4 skipped during auto due to previous errors)
```

In [186]:
```
using Evolutionary
sol = solve(prob, CMAES(μ =40 , λ = 100),abstol=1e-15)
```

Out[186]:
```
u: 2-element Vector{Float64}:
 0.986821120271575
 0.9737506156710922
```

In [187]:
```
@test 10*sol.minimum < 11
```

Out[187]:
```
Test Passed
```

In [188]:
```
using BlackBoxOptim
prob = GalacticOptim.OptimizationProblem(optprob, x0, lb=[-1.0, -1.0], ub=[0.8, 0.8])
sol = solve(prob, BBO())
```

```
Starting optimization with optimizer DiffEvoOpt{FitPopulation{Float64}, RadiusLimitedSelector, BlackBoxOptim.AdaptiveDiffE
voRandBin{3}, RandomBound{ContinuousRectSearchSpace}}
0.00 secs, 0 evals, 0 steps

Optimization stopped after 10001 steps and 0.03 seconds
Termination reason: Max number of steps (10000) reached
Steps per second = 384653.51
Function evals per second = 376576.59
Improvements/step = 0.45310
Total function evaluations = 9791


Best candidate found: [0.8, 0.64]

Fitness: 0.040000000
```

Out[188]:
```
u: 2-element Vector{Float64}:
 0.8
 0.6399999999935596
```

In [189]:
```
@test 10*sol.minimum < 11
```

Out[189]:
```
Test Passed
```

Defining OptimizationProblems

In [ ]:
```
OptimizationProblem(f, x, p = DiffEqBase.NullParameters(),;
                    lb = nothing,
                    ub = nothing,
                    lcons = nothing,
                    ucons = nothing,
                    kwargs...)
```

Formally, the OptimizationProblem finds the minimum of f(x,p) with an initial condition x. The parameters p are optional. lb and ub are arrays matching the size of x, which stand for the lower and upper bounds of x, respectively.

f is an OptimizationFunction, as defined here. If f is a standard Julia function, it is automatically converted into an OptimizationFunction with NoAD(), i.e., no automatic generation of the derivative functions.

Any extra keyword arguments are captured to be sent to the optimizers.

OptimizationFunction

In [ ]:
```
OptimizationFunction{iip}(f,adtype=NoAD();
                          grad=nothing,
                          hess=nothing,
                          hv=nothing,
                          cons=nothing,
                          cons_j=nothing,
                          cons_h=nothing)
```

The keyword arguments are as follows:

grad: Gradient

hess: Hessian

hv: Hessian vector products hv(du,u,p,t,v) = H*v

cons: Constraint function

cons_j

cons_h

Common Solver Options

In [ ]:
```
solve(prob,alg;kwargs...)
```

The arguments to solve are common across all of the optimizers. These common arguments are:

maxiters (the maximum number of iterations)

abstol (absolute tolerance)

reltol (relative tolerance)

Local Gradient-Based Optimization

ADAM() is a good default with decent convergence rate. BFGS() can converge faster but is more prone to hitting bad local optima. LBFGS() requires less memory than BFGS and thus can have better scaling.

Flux.Optimise.Descent: Classic gradient descent optimizer with learning rate

Flux.Optimise.Momentum: Classic gradient descent optimizer with learning rate and momentum

Flux.Optimise.Nesterov: Gradient descent optimizer with learning rate and Nesterov momentum

Flux.Optimise.RMSProp: RMSProp optimizer

Flux.Optimise.ADAM: ADAM optimizer

Flux.Optimise.RADAM: Rectified ADAM optimizer

Flux.Optimise.AdaMax: AdaMax optimizer

Flux.Optimise.ADAGRad: ADAGrad optimizer

Flux.Optimise.ADADelta: ADADelta optimizer

Flux.Optimise.AMSGrad: AMSGrad optimizer

Flux.Optimise.NADAM: Nesterov variant of the ADAM optimizer

Flux.Optimise.ADAMW: ADAMW optimizer

Local Derivative-Free Optimization

Derivative-free optimizers are optimizers that can be used even in cases where no derivatives or automatic differentiation is specified. While they tend to be less efficient than derivative-based optimizers, they can be easily applied to cases where defining derivatives is difficult.

Optim.NelderMead: Nelder-Mead optimizer

Optim.SimulatedAnnealing: Simulated Annealing

In [ ]: