BasicTrustRegion

June 16, 2020

1 Basic Trust Region Algorithm

[1]:
```
using LinearAlgebra
using Optim
```

[2]:
```
struct BasicTrustRegion{T <: Real}
     1:: T
     2:: T
     1:: T
     2:: T
end

function BTRDefaults()
    return BasicTrustRegion(0.01,0.9,0.5,0.5)
end
```

[2]:
```
BTRDefaults (generic function with 1 method)
```

The state of the algorithm is declared with the additional keyword mutable at the content has to be modified from iteration to iteration.

[3]:
```
mutable struct BTRState
    iter::Int
    x::Vector
    xcand::Vector
    g::Vector
    step::Vector
    Δ::Float64
     ::Float64

    function BTRState()
        return new()
    end
end
```

[4]:
```
function acceptCandidate!(state::BTRState, b::BasicTrustRegion)
    # If the iteration is successful, update the iterate
    if (state.  >= b. 1)
```

```
        state.x = copy(state.xcand)
        return true
    else
        return false
    end
end
```

[4]: acceptCandidate! (generic function with 1 method)

[5]:
```
function updateRadius!(state::BTRState, b::BasicTrustRegion)
    if (state.  >= b.2)
        # very successful iterate
        stepnorm = norm(state.step)
        state.Δ = min(1e20,max(4*stepnorm,state.Δ))
    elseif (state.  >= b.1)
        # successful iterate
        state.Δ *= b.2
    else
        # unsuccessful iterate
        state.Δ *= b.1
    end
end
```

[5]: updateRadius! (generic function with 1 method)

1.1 Cauchy Step

Check the Cauchy Step value.

[6]:
```
function CauchyStep(g::Vector,H::Matrix,Δ::Float64)
    q = dot(g,H*g)
    normg = norm(g)

    if (q <= 0)
        # the curvature along g is non positive
          = 1.0
    else
        # the curvature along g is positive
          = min((normg*normg*normg)/(q*Δ),1.0)
    end

    return - *g*Δ/normg
end
```

[6]: CauchyStep (generic function with 1 method)

1.2 Basic Trust Region Algorithm

[9]:
```
function btr(f::Function, g!::Function, H!::Function,
    x0::Vector, tol::Float64 = 1e-6, verbose::Bool = false)

    b = BTRDefaults()
    state = BTRState()
    state.iter = 0
    state.Δ = 1.0
    state.x = x0
    n=length(x0)

    state.g = zeros(n)
    H = zeros(n,n)

    fx = f(x0)
    g!(state.g, x0)
    H!(H, x0)

    nmax = 100000

    tol2 = tol*tol

    function model(s::Vector, g::Vector, H::Matrix)
        return dot(s, g)+0.5*dot(s, H*s)
    end

    if (verbose)
           println(state)
       end

    while (dot(state.g,state.g) > tol2 && state.iter < nmax)
        # Compute the step by approximately minimize the model
        state.step = CauchyStep(state.g, H, state.Δ)
        state.xcand = state.x+state.step

        # Compute the actual reduction over the predicted reduction
        fcand = f(state.xcand)
        state.  = (fcand-fx)/(model(state.step, state.g, H))

        if (acceptCandidate!(state, b))
            g!(state.g, state.x)
            H!(H, state.x)
            fx = fcand
        end

        if (verbose)
```

```
            println(state)
        end

        updateRadius!(state, b)
        state.iter += 1
    end

    return state
end
```

[9]: btr (generic function with 3 methods)

1.3 Example

1.3.1 Rosenbrock function

[10]:
```
function rosenbrock(x::Vector)
    return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
end

function rosenbrock_gradient!(storage::Vector, x::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!(storage::Matrix, x::Vector)
    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
```

[10]: rosenbrock_hessian! (generic function with 1 method)

[11]:
```
state = btr(rosenbrock, rosenbrock_gradient!, rosenbrock_hessian!, [0,0])
```

[11]:
```
BTRState(8969, [0.9999989788332465, 0.9999979544863866], [0.9999989788332465,
0.9999979544863866], [-7.698751132041958e-7, -6.362298465845129e-7],
[-1.7150630021960458e-9, 2.0753261085477156e-9], 0.2462880346108132,
1.0000000006312493)
```

[12]:
```
state.Δ
```

[12]: 0.2462880346108132

Let's check the first iteration, starting from (0,0). In order to compute the step length, we first have to know the gradient and the Hessian at (0,0).

[13]:
```
x = [0; 0]
grad=zeros(2)
hess=zeros(2,2)
rosenbrock_gradient!(grad,x)
rosenbrock_hessian!(hess,x)
```

[13]:
```
200.0
```

We know that

$$\alpha^* = \min\left\{\frac{\Delta_k}{\|\nabla f(x_k)\|_k}, \frac{\|\nabla f(x_k)\|_2^2}{\nabla f(x_k)^T H_k \nabla f(x_k)}\right\}$$

The first term of the minimization is

[14]:
```
1.0/norm(grad)
```

[14]:
```
0.5
```

[15]:
```
grad
```

[15]:
```
2-element Array{Float64,1}:
 -2.0
  0.0
```

The second is

[16]:
```
dot(grad,grad)/dot(grad,hess*grad)
```

[16]:
```
0.5
```

Therefore $\alpha^* = 0.5$ and $x_1 = (0,0) + 0.5 * \nabla f(0.0) = (1,0)$. We can easily check this value by looking at the first iterations of the optimization procedure.

[17]:
```
state = btr(rosenbrock, rosenbrock_gradient!, rosenbrock_hessian!, [0,0], 1.0,␣
↪true)
```

```
BTRState(0, [0, 0], #undef, [-2.0, 0.0], #undef, 1.0, 0.0)
BTRState(0, [0, 0], [1.0, 0.0], [-2.0, 0.0], [1.0, -0.0], 1.0, -99.0)
BTRState(1, [0, 0], [0.5, 0.0], [-2.0, 0.0], [0.5, -0.0], 0.5,
-7.333333333333333)
BTRState(2, [0.25, 0.0], [0.25, 0.0], [4.75, -12.5], [0.25, -0.0], 0.25,
0.10714285714285714)
BTRState(3, [0.23106741878274778, 0.04982258215066378], [0.23106741878274778,
0.04982258215066378], [-1.2079406438155806, -0.7139139744515923],
[-0.018932581217252234, 0.04982258215066378], 0.125, 1.0118911526078314)
BTRState(4, [0.4146031514360326, 0.1582953991083784], [0.4146031514360326,
0.1582953991083784], [1.0847094833077477, -2.7200748144622757],
[0.18353573265328482, 0.10847281695771462], 0.2131940833590836,
1.2474597336770734)
BTRState(5, [0.4110649159627491, 0.1671680653352532], [0.4110649159627491,
```

```
0.1671680653352532], [-0.8808675778439622, -0.3612599600417543],
[-0.0035382354732835064, 0.008872666226874808], 0.8527763334363345,
1.0021125158850077)
```

[17]:
```
BTRState(6, [0.4110649159627491, 0.1671680653352532], [0.4110649159627491,
 0.1671680653352532], [-0.8808675778439622, -0.3612599600417543],
 [-0.0035382354732835064, 0.008872666226874808], 0.8527763334363345,
 1.0021125158850077)
```

1.3.2 Example 2

[18]:
```
using ForwardDiff

f(x) = x[1]^4 + x[1]^2 + x[1]*x[2] + (1+x[2])^2
g = x -> ForwardDiff.gradient(f, x);
H = x -> ForwardDiff.hessian(f, x);
function g!(storage::Vector, x::Vector)
    s = g(x)
    storage[1:length(s)] = s[1:length(s)]
end
function H!(storage::Matrix, x::Vector)
    s = H(x)
    n, m = size(s)
    storage[1:n,1:m] = s[1:length(s)]
end
```

[18]:
```
H! (generic function with 1 method)
```

[19]:
```
state = btr(f, g!, H!, [0,0], 1e-6, true)
```

```
BTRState(0, [0, 0], #undef, [0.0, 2.0], #undef, 1.0, 0.0)
BTRState(0, [0.0, -1.0], [0.0, -1.0], [-1.0, 0.0], [-0.0, -1.0], 1.0, 1.0)
BTRState(1, [0.5, -1.0], [0.5, -1.0], [0.5, 0.5], [0.5, -0.0], 4.0, 0.75)
BTRState(2, [0.38888888888888884, -1.1111111111111112], [0.38888888888888884,
-1.1111111111111112], [-0.09807956104252424, 0.16666666666666652],
[-0.11111111111111113, -0.11111111111111113], 2.0, 1.0466392318244182)
BTRState(3, [0.45047313503384356, -1.2157612636511388], [0.45047313503384356,
-1.2157612636511388], [0.05083593379895146, 0.018950607731565983],
[0.061584246144954696, -0.10465015254002756], 2.0, 0.9678297689788521)
BTRState(4, [0.43986596102821646, -1.2197154035419506], [0.43986596102821646,
-1.2197154035419506], [0.000441214028436665, 0.00043515394431520305],
[-0.010607174005627131, -0.003954139890811828], 2.0, 1.0069616641639731)
BTRState(5, [0.4397603278573833, -1.219819585838922], [0.4397603278573833,
-1.219819585838922], [-0.00011943320330654039, 0.00012115617953922797],
[-0.00010563317083319118, -0.000104182296971509 8], 2.0, 1.0000451096982255)
BTRState(6, [0.43981603834919525, -1.2198761000256082], [0.43981603834919525,
-1.2198761000256082], [6.47756259040122e-5, 6.383829797884655e-5],
```

```
[5.571049181199027e-5, -5.651418668616214e-5], 2.0, 0.9999549343274435)
BTRState(7, [0.43980053228737653, -1.2198913817088193], [0.43980053228737653,
-1.2198913817088193], [-1.751047847320386e-5, 1.7768869737977422e-5],
[-1.550606181871003e-5, -1.528168321102881e-5], 2.0, 1.0000065675814054)
BTRState(8, [0.43980870080183004, -1.2198996707609717], [0.43980870080183004,
-1.2198996707609717], [9.497741120689795e-6, 9.359279886600458e-6],
[8.168514453503695e-6, -8.289052152355949e-6], 2.0, 0.9999933996771568)
BTRState(9, [0.43980642726885405, -1.2199019111496265], [0.43980642726885405,
-1.2199019111496265], [-2.566966205819199e-6, 2.604969600972229e-6],
[-2.273532976014118e-6, -2.2403886548959134e-6], 2.0, 0.9999997844893755)
BTRState(10, [0.43980762475385293, -1.2199031263631392], [0.43980762475385293,
-1.2199031263631392], [1.3923477224508929e-6, 1.372027574442214e-6],
[1.1974849988809678e-6, -1.2152135126023924e-6], 2.0, 1.0000284734255356)
BTRState(11, [0.43980729145998415, -1.2199034547928633], [0.43980729145998415,
-1.2199034547928633], [-3.763005362866778e-7, 3.8187425749347526e-7],
[-3.332938688061702e-7, -3.284297241422172e-7], 2.0, 1.000040618838091)
```

[19]:
```
BTRState(12, [0.43980729145998415, -1.2199034547928633], [0.43980729145998415,
 -1.2199034547928633], [-3.763005362866778e-7, 3.8187425749347526e-7],
 [-3.332938688061702e-7, -3.284297241422172e-7], 2.0, 1.000040618838091)
```

[ ]: