Line Search Algorithm for Optimization

- Mathe 3 (CES)
- WS24
- Lambert Theisen (`theisen@acom.rwth-aachen.de`)


```julia
using PlutoUI, Calculus, Gadfly, LinearAlgebra
```

Define Objective

$$f(x) = x^2$$

f = #7 (generic function with 1 method)

```julia
f = (x -> x[1]^2)
```

Line Search

1. Given $x^{(0)}$
2. For $k = 0, 1, 2, \ldots$ do
    1. Update: $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
3. End

line_search (generic function with 1 method)

```julia
function line_search(f, x0, α, d, kmax)
    x = x0
    hist = []
    push!(hist, x)
    for k=1:kmax
        x = x + α(x) * d(x)
        push!(hist, x)
    end
    return x, hist
end
```

Check Line Search

- Observe that different step sizes change the result!

```
(0.0, [1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

```
line_search(f, 1, (x->1.0), (x->-sign(x)), 10)
```

Gradient Descent

- Is line search with $d^{(k)} = -\nabla f(x^{(k)})$

```
hessian (generic function with 9 methods)
```

```
begin
    # some notation
    ∇ = derivative
    ∇² = hessian
end
```

```
gradient_descent (generic function with 1 method)
```

```
function gradient_descent(f, x0, α, kmax)
    return line_search(f, x0, α, (x->-∇(f, x)), kmax)
end
```

Check Gradient Descent

```
(2.03704e-10, [1, 0.8, 0.64, 0.512, 0.4096, 0.32768, 0.262144, 0.209715, 0.167772,    more
```

```
gradient_descent(f, 1, (x->0.1), 100)
```

```
(2.65614e-5, [1, -0.9, 0.81, -0.729, 0.6561, -0.59049, 0.531441, -0.478297, 0.430467,    m
```

```
gradient_descent(f, 1, (x->0.95), 100) # slower, oscillating but converging
```

Newton's Method for Optimization

- Is line search with $d^{(k)} = -[\nabla^2 f(x^{(k)})]^{-1} \nabla f(x^{(k)})$

```
newton (generic function with 1 method)
```

```
function newton(f, x0, α, kmax)
    return line_search(f, x0, α, (x->-inv(∇²(f, x))*∇(f, x)), kmax)
end
```

Check Newton's Method

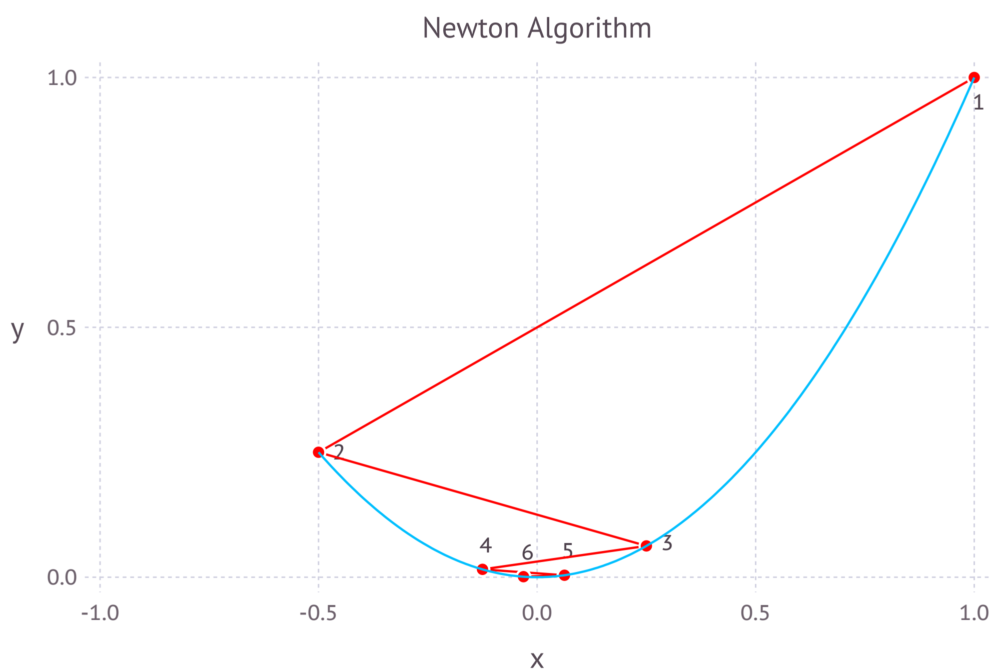
```
(5.01609e-24, [1.0, -0.500001, 0.25, -0.125, 0.0625001, -0.0312501, 0.015625, -0.00781251
```

```
newton(f, 1., (x->1.5), 100) # works well 😎
```

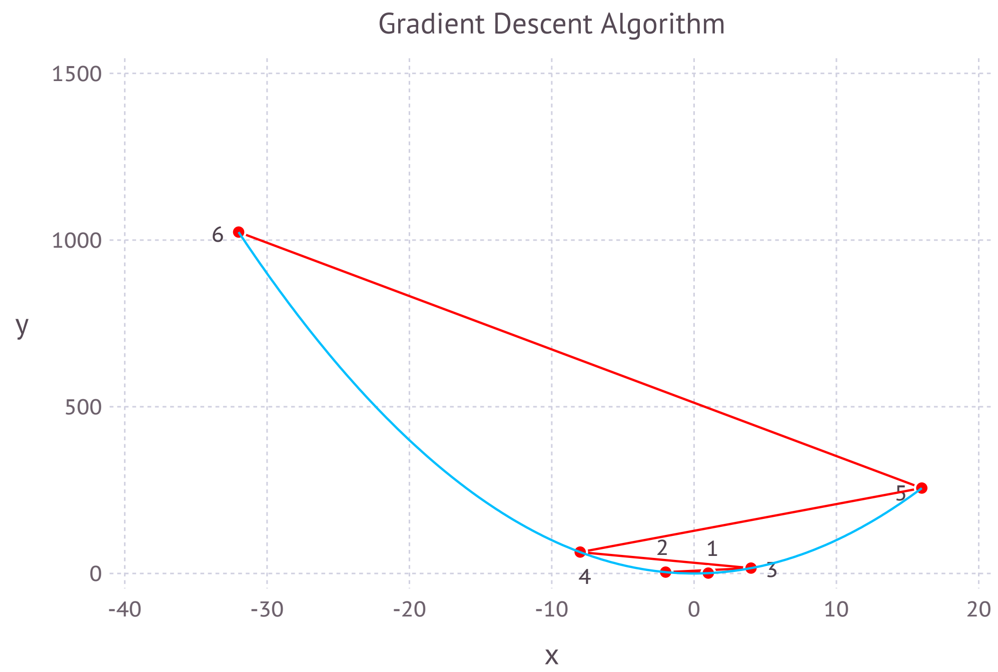
```
(1.26764e30, [1.0, -2.0, 4.00001, -7.99999, 16.0, -31.9999, 63.9998, -128.0, 255.999,    m
```

```
newton(f, 1., (x->3.0), 100) # diverged 😓
```

Visualize Results




```
begin
    res_n = newton(f, 1., (x->1.5), 5)
    Gadfly.plot(
        Guide.title("Newton Algorithm"),
        layer(f, minimum(res_n[2]), maximum(res_n[2])),
        layer(x=res_n[2], y=f.(res_n[2]), label=string.(1:length(res_n[2])),
        Geom.point, Geom.path, Geom.label, Theme(default_color=color("red")))
    )
end
```




```
begin
    res_gd = gradient_descent(f, 1., (x->1.5), 5)
    Gadfly.plot(
        Guide.title("Gradient Descent Algorithm"),
        layer(f, minimum(res_gd[2]), maximum(res_gd[2])),
        layer(x=res_gd[2], y=f.(res_gd[2]), label=string.(1:length(res_gd[2])),
        Geom.point, Geom.path, Geom.label, Theme(default_color=color("red")))
    )
end
```

Two-Dimensional Optimization

Define Objective

$$g(x, y) = x^2 + y^2$$

`g = #25 (generic function with 1 method)`

```
g = (x->x[1]^2+x[2]^2)
```

Check Methods

- both work

res_gd_2d =

([4.18545e-22, 4.18545e-22], [[1.0, 1.0], [0.2, 0.2], [0.04, 0.04], [0.008, 0.008], [0.0016

```
res_gd_2d = gradient_descent(g, [1.,1.], (x->0.4), 100)
```

res_n_2d =

([2.05496e-22, 2.98015e-22], [[1.0, 1.0], [0.6, 0.6], [0.36, 0.36], [0.216, 0.216], [0.1296

```
res_n_2d = newton(g, [1.,1.], (x->0.4), 100)
```

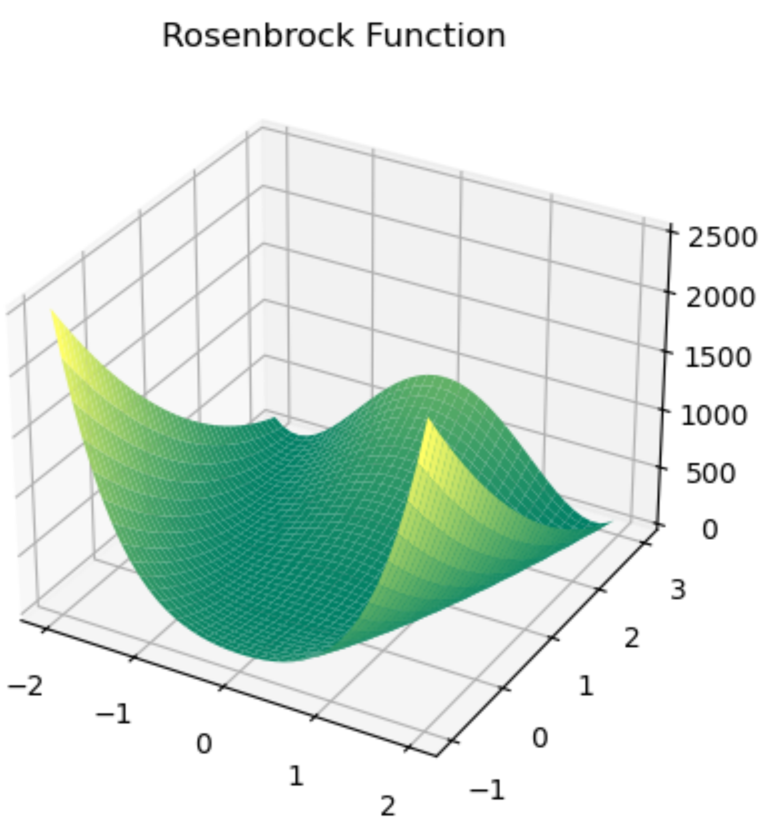
[2.05496e-22, 2.98015e-22]

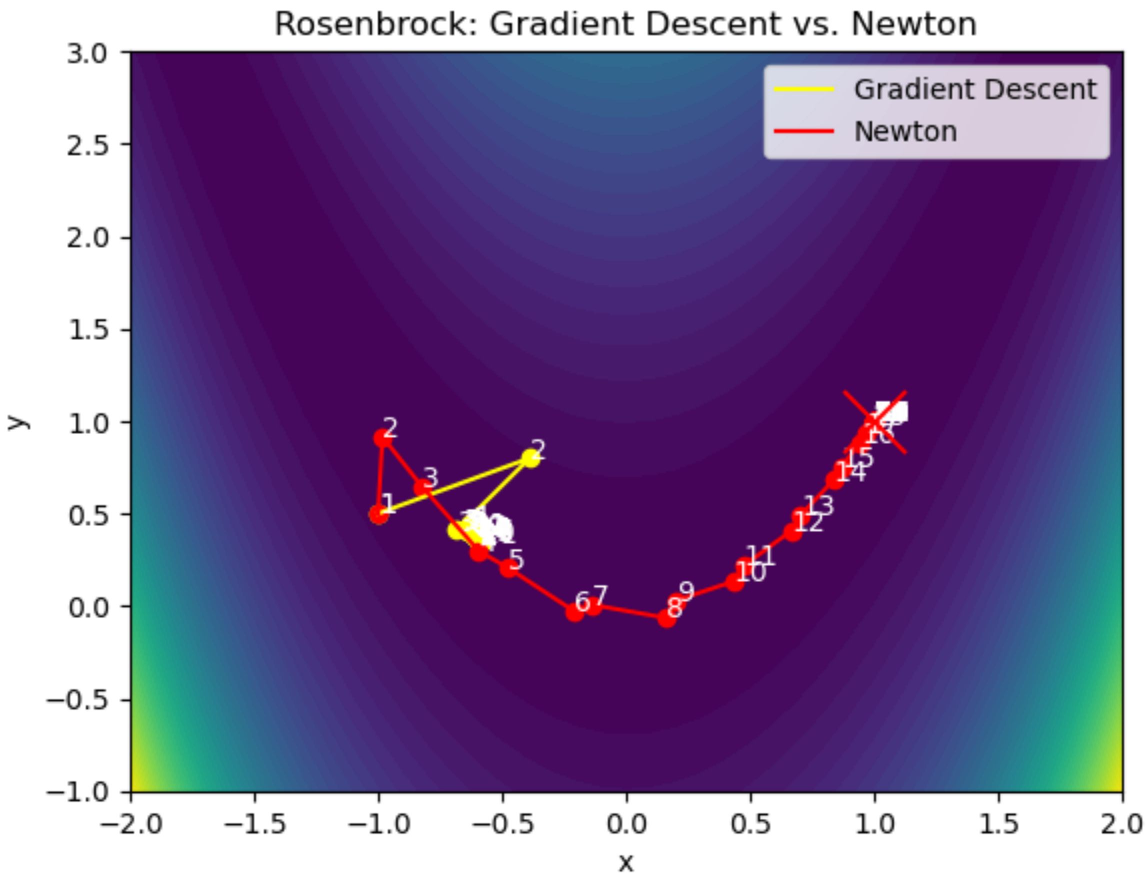
```
res_n_2d[2][end]
```

true

```
norm(res_n_2d[2][end] - [0,0]) < eps(Float64) # is converged to machine-precision?
```

Test Gradient Descent vs Newton for 2D Rosenbrock




Rosenbrock: Gradient Descent vs. Newton
Gradient Descent
Newton
x
y
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

```julia
begin
    # Rosenbrock function with x∗ = [a,a^2], f(x∗)=0
    a = 1
    b = 100
    h = (x -> (a-x[1])^2 + b*(x[2]-x[1]^2)^2)

    x0 = [-1.0,0.5]

    # Gradient Descent
    res_gd_2d_rb = gradient_descent(h, x0, (x->0.003), 20)
    res_gd_2d_rb_x = [res_gd_2d_rb[2][i][1] for i=1:length(res_gd_2d_rb[2])]
    res_gd_2d_rb_y = [res_gd_2d_rb[2][i][2] for i=1:length(res_gd_2d_rb[2])]

    # Newton
    res_n_2d_rb = newton(h, x0, (x->0.9), 50)
    res_n_2d_rb_x = [res_n_2d_rb[2][i][1] for i=1:length(res_n_2d_rb[2])]
    res_n_2d_rb_y = [res_n_2d_rb[2][i][2] for i=1:length(res_n_2d_rb[2])]

    clf()
    Δ = 0.1
    X=collect(-2:Δ:2)
    Y=collect(-1:Δ:3)
    F=[h([X[j],Y[i]]) for i=1:length(X), j=1:length(Y)]
    contourf(X,Y,F, levels=50)
    PyPlot.title("Rosenbrock: Gradient Descent vs. Newton")

    # res_gd_2d_rb
    PyPlot.plot(res_gd_2d_rb_x, res_gd_2d_rb_y, color="yellow", label="Gradient Descent")
    PyPlot.scatter(res_gd_2d_rb_x, res_gd_2d_rb_y, color="yellow")
    for i=1:length(res_gd_2d_rb_x)
        annotate(string(i), [res_gd_2d_rb_x[i], res_gd_2d_rb_y[i]], color="w", zorder=2)
    end

    # res_n_2d_rb
    PyPlot.plot(res_n_2d_rb_x, res_n_2d_rb_y, color="red", label="Newton")
    PyPlot.scatter(res_n_2d_rb_x, res_n_2d_rb_y, color="red")
    for i=1:length(res_n_2d_rb_x)
        annotate(string(i), [res_n_2d_rb_x[i], res_n_2d_rb_y[i]], color="w", zorder=2)
    end

    # legend(["Gradient Descent", "Newton"])
    legend()

    xlabel("x")
    ylabel("y")

    # Mark minimum
    scatter(a, a^2, color="r", s=500, zorder=3, marker="x")

    gcf()
end
```

```
end
```

Broyden's Method

Homework: Adapt GD and Newton to use the generic framework

line_search2 (generic function with 1 method)

```julia
function line_search2(f, x0, α, B0, Bk, kmax, tol)
    x = x0
    B = B0
    k = 0
    Δx = Inf
    hist = []
    push!(hist, x)
    while (k <= kmax) && (norm(Δx) > tol)
        # invB = length(x)==1 ? 1/B
        d = -inv(B) * ∇(f, x)
        Δx = α(x) * d
        x = x + Δx
        B = Bk(x, d, f, B)
        push!(hist, x)
        k =+ 1
    end
    return x, hist
end
```

broyden (generic function with 1 method)

```julia
function broyden(f, x0, α, kmax, tol)
    return line_search2(
        f, x0, α,
        I(length(x0)),
        (x,d,f,B)->(B + (∇(f, x) * d') / (norm(d,2)^2)), kmax, tol
    )
end
```

```
([-1.04412e-22, -1.04412e-22], [[1.0, 1.0], [0.2, 0.2], [-2.64139e-12, -2.64139e-12], [-1.
```

```julia
broyden(g, [1.,1.], (x->0.4), 100, 1E-10)  # works quite fast
```

```
([2.05496e-22, 2.98015e-22], [[1.0, 1.0], [0.6, 0.6], [0.36, 0.36], [0.216, 0.216], [0.1296
```

```julia
newton(g, [1.,1.], (x->0.4), 100)  # slower than Broyden 🤔
```

```
([-5.59446e-22], [[1], [0.8], [-6.48059e-11], [-5.59446e-22]])
```

```julia
broyden(f, [1], (x->0.1), 100, 1E-10)  # works
```