present

Line Search Stepsize Control and Trust-Region Methods

- Mathe 3 (CES)
- WS20
- Lambert Theisen (`theisen@acom.rwth-aachen.de`)

Stepsize Control Algorithm

backtracking_linesearch (generic function with 1 method)

```
function backtracking_linesearch(f, x, d, αmax, cond, β)
    @assert 0 < β < 1
    α = αmax
    while !cond(f, d, x, α)
        α *= β
    end
    return α
end
```

Armijo Stepsize Conditon

- We need to specify a conditon for the backtracking algorithm
- Use Armijo condition, which is the first Wolfe condition

$$\begin{aligned} &\textbf{i)} \quad f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k), \\ &\textbf{ii)} \quad -\mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq -c_2 \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k), \end{aligned}$$

armijo (generic function with 1 method)

```
armijo(f, d, x, α) = f(x + α*d) <= f(x) + 1E-4 * α * derivative(f, x)' * d
```

backtracking_linesearch_armijo1 (generic function with 1 method)

```
function backtracking_linesearch_armijo1(f, x, d, αmax, β)
    return backtracking_linesearch(f, x, d, αmax, armijo, β)
end
```

Use Backtracking Algorithm in Gradient Descent

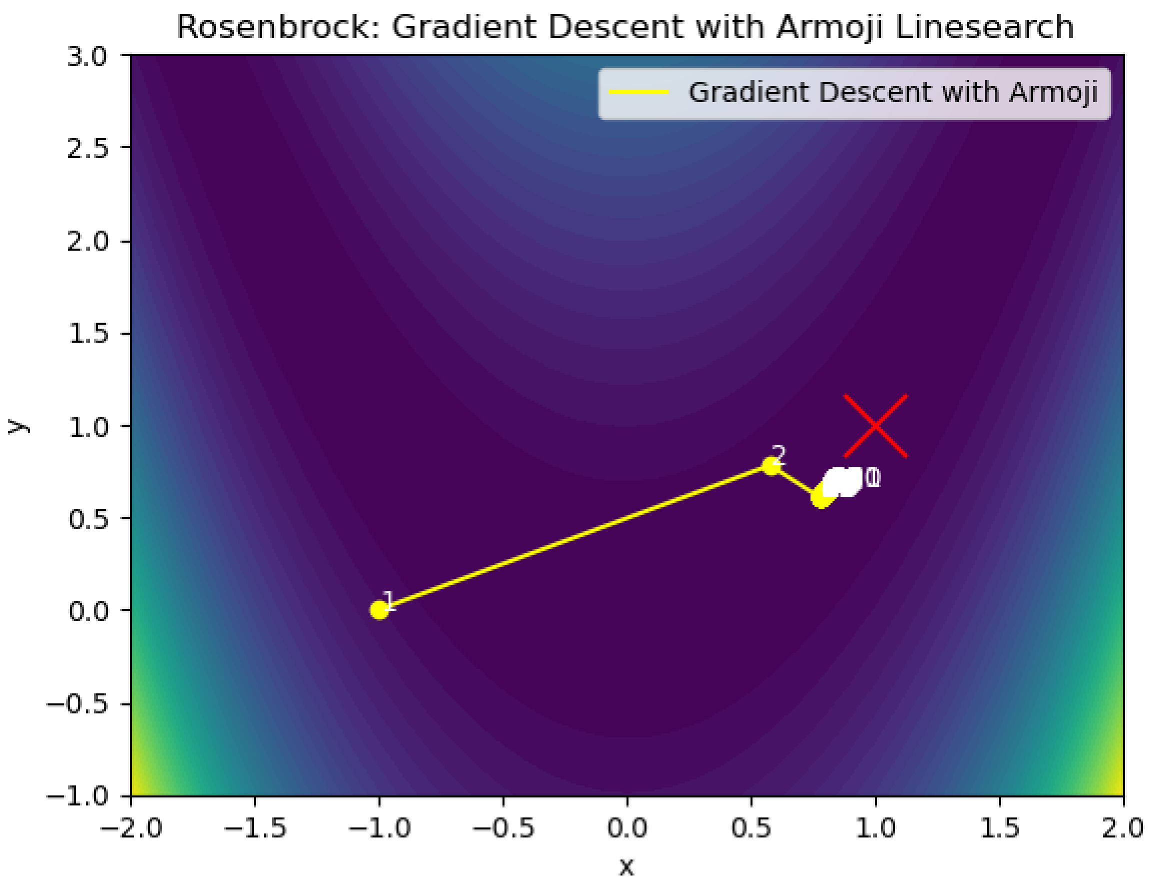
- Same as last week, but with adaptive step size

```
gradient_descent_armijo1 (generic function with 1 method)
```

```
function gradient_descent_armijo1(f, x0, kmax)
    x = x0
    hist = []
    push!(hist, x)
    for k=1:kmax
        x = x + backtracking_linesearch_armijo1(
            f, x, -derivative(f, x), 2, 0.5
        ) * -derivative(f, x)
        push!(hist, x)
    end
    return x, hist
end
```

Rosenbrock: GD with Armijo

- Remember from last week: GD was very sensitive to step width
- Now: Line search automatically choose a valid step size and we have an easy life



Still not the Best Convergence...

```
Float64[0.818009,  0.667712]
```

```
res_gd_2d_rb_arm1[2][end] # still not converged after 100 its 😓
```

Trust-Region Methods

1. Given $x^{(k)}$
2. Replace f by (e.g 2nd order) approximation $\hat{f}$
3. Solve $\hat{x} = \mathrm{argmin}_{x \in D_k} \hat{f}(x)$ for a given thrust region $D_k = \{x \in \mathbb{R}^n \mid \|x - x^{(k)}\|_p \leq \delta\}$
4. Test improvement $\rho = \frac{\text{actual improvement}}{\text{predicted improvement}} = \frac{f(x^k) - f(\hat{x})}{f(x^k) - \hat{f}(\hat{x})}$
5. If $\rho > \rho_{\min}$, set $x^{(k+1)} = \hat{x}$, else decrease thrust region radius $\delta \leftarrow \sigma\delta$

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

```
function trust_region(
    f, fhat, x0, solve_subproblem, kmax, rhomin, delta0, sigma
)
    println("START")
    hist = []
    x = x0
    push!(hist, [x0, 0])
    for k=1:kmax
        @show k
        delta = delta0
        @show delta
        xhatval = nothing
        xhatval = solve_subproblem(x, delta)
        @show xhatval
        rho = (f(x) - f(xhatval)) / (f(x) - fhat(xhatval, x))
        @show rho
        i = 0
        while rho < rhomin && i<10
            delta *= sigma
            @show delta
            xhatval = solve_subproblem(x, delta)
            @show xhatval
            rho = (f(x) - f(xhatval)) / (f(x) - fhat(xhatval, x))
            @show rho
            i += 1
        end
        @show delta
        x = xhatval
        @show x
        push!(hist, [x, delta])
    end
    return x, hist
end
```

Define Problem

- Define objective: $f(x, y) = x^2 + y^2(y^2 - 1)$
- Derive quadratic approximation $\hat{f} = \hat{f}(x) := f(x^{(k)}) + (x - x^{(k)})^T \nabla f(x^{(k)}) + \frac{1}{2}(x - x^{(k)})^T \nabla^2 f(x^{(k)})(x - x^{(k)})$
- Minima are at $(0, \pm 1/\sqrt{6})$, saddle point at $(0, 0)$

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

```
# objective
f(x) = x[1]^2 + x[2]^2 * (x[2]^2 - 1)
```

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

```
# quadratic approximation
fhat(x, x0) = (
    f(x0) + (x-x0)' * derivative(f, x0)
    + 1/2 * (x-x0)' * hessian(f, x0) * (x-x0)
)
```

Define Solution to Subproblem

- Either analytically (see below)
- Or use approximate solutions (Cauchy point, ...)

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

```
solve_subproblem(x, delta) = [
    if (abs(x[1]) <= delta)
        0
    else
        x[1] - sign(x[1])*delta
    end,
    if (x[2] == 0)
        if (abs(x[2]) <= delta)
            delta
        else
            x[2] + sign(x[2]) * delta
        end
    elseif (x[2]^2 >= 1/6)
        if (abs(x[2] - (4*x[2]^3)/(6*x[2]^2-1)) <= delta)
            (4*x[2]^3)/(6*x[2]^2-1)
        else
            x[2] - sign(x[2] - (4*x[2]^3)/(6*x[2]^2-1)) * delta
        end
    else
        nothing
    end
]
```

Test Thrust-Region Method with Saddle Point

- We can escape the saddle point $x^{(0)} = (0, 0)$ 👏

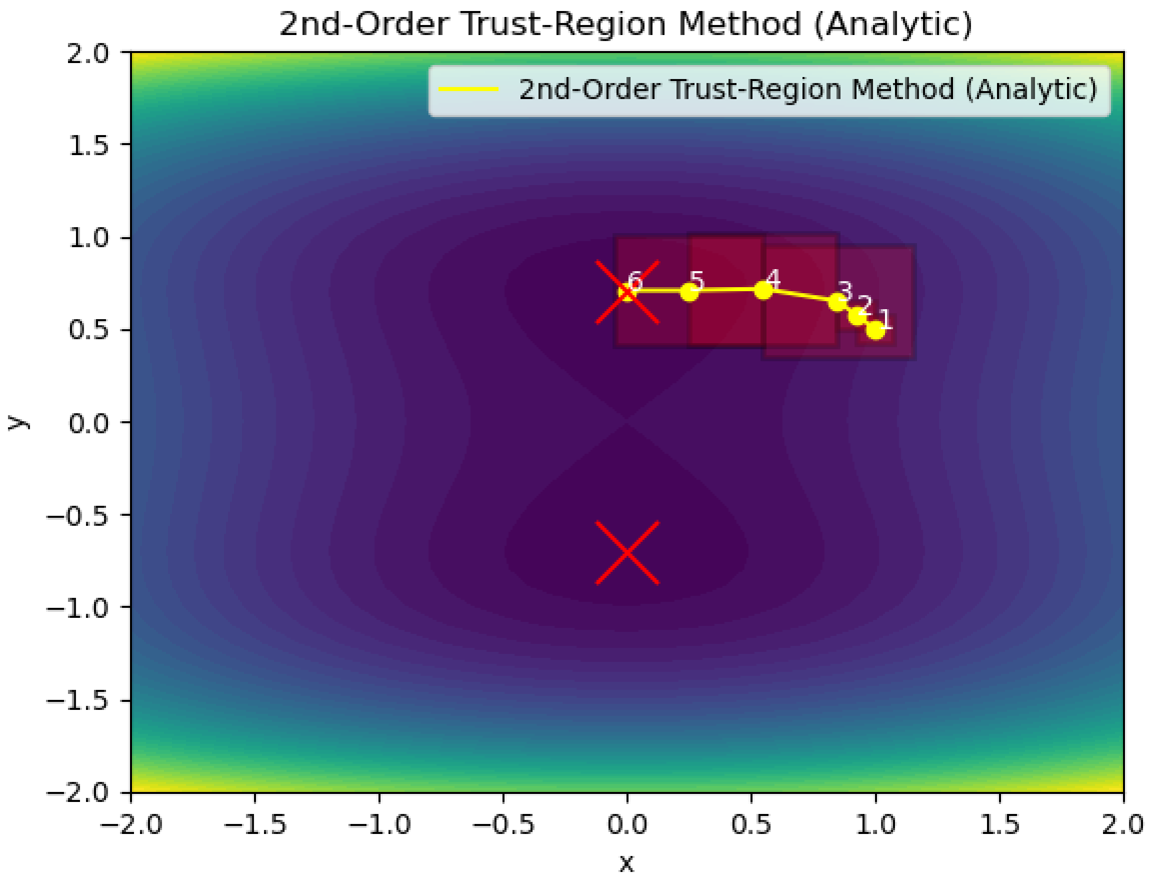
```
(
1:   Float64[0.0, 0.707107]
```

```
2:    Any[
          1:    Any[Float64[0.0,  0.0],  0]
          2:    Any[Float64[0.0,  0.5],  0.5]
          3:    Any[Float64[0.0,  0.75],  0.25]
          4:    Any[Float64[0.0,  0.710526],  0.5]
          5:    Any[Float64[0.0,  0.707131],  0.5]
          6:    Any[Float64[0.0,  0.707107],  0.5]
       ]
)
```

```
trust_region(f, fhat, [0.,0.], solve_subproblem, 5, 0.5, 0.5, 0.5)
```

Trust-Region Method in Action 😎

x01 = 1 x02 = 0,5 k = 5 rhomin = 0,99 delta0 = 0,3 sigma = 0,5




```
let
    # Perform Optimization
    tr = trust_region(f, fhat, [Float64(x01),Float64(x02)], solve_subproblem, k,
rhomin, delta0, sigma)
    tr_x = [
        tr[2][i][1][1] for i=1:length(tr[2])
    ]
    tr_y = [
        tr[2][i][1][2] for i=1:length(tr[2])
    ]
    deltas = [
        tr[2][i][2][1] for i=1:length(tr[2])
    ]

    # Plot annotations
    clf()
    ax = gca()
```

```
    Δ = 0.1
    X=collect(-2:Δ:2)
    Y=collect(-2:Δ:2)
    F=[f([X[j],Y[i]]) for i=1:length(Y), j=1:length(X)]
    contourf(X,Y,F, levels=50)
    PyPlot.title("2nd-Order Trust-Region Method (Analytic)")

    # Trust Regions
    for i=2:length(tr_x)
        ax.add_patch(PyPlot.matplotlib.pyplot.Rectangle((tr_x[i-1]-deltas[i],
tr_y[i-1]-deltas[i]), 2deltas[i], 2deltas[i], facecolor="red", alpha=0.2,
edgecolor="black", linewidth=2.))
    end

    # Trajectory
    PyPlot.plot(tr_x, tr_y, color="yellow", zorder=2)
    scatter(tr_x, tr_y, color="yellow", zorder=2)
    for i=1:length(tr_x)
        annotate(string(i), [tr_x[i], tr_y[i]], color="w", zorder=3)
    end

    # Plot annotations
    legend(["2nd-Order Trust-Region Method (Analytic)"])
    xlabel("x")
    ylabel("y")

    # Mark minima
    scatter(0, 1/sqrt(2), color="r", s=500, zorder=3, marker="x")
    scatter(0, -1/sqrt(2), color="r", s=500, zorder=3, marker="x")

    gcf()
end
```

See you next week ✌️

Questions?