present

Line Search Stepsize Control and Trust-Region Methods

- Mathe 3 (CES)
- WS21
- 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
    @show α
    return α
end
```

Wolfe Stepsize Conditon

- We need to specify a conditon for the backtracking algorithm
- Use Wolfe conditions

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

armijo (generic function with 1 method)

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

```
function backtracking_linesearch_wolfe(f, x, d, αmax, β)  #TODO
    return backtracking_linesearch(f, x, d, αmax, (f, d, x, α)->(armijo(f, d, x, α)&&curvature(f, d, x, α)), β)
end
```

Use Backtracking Algorithm in Gradient Descent

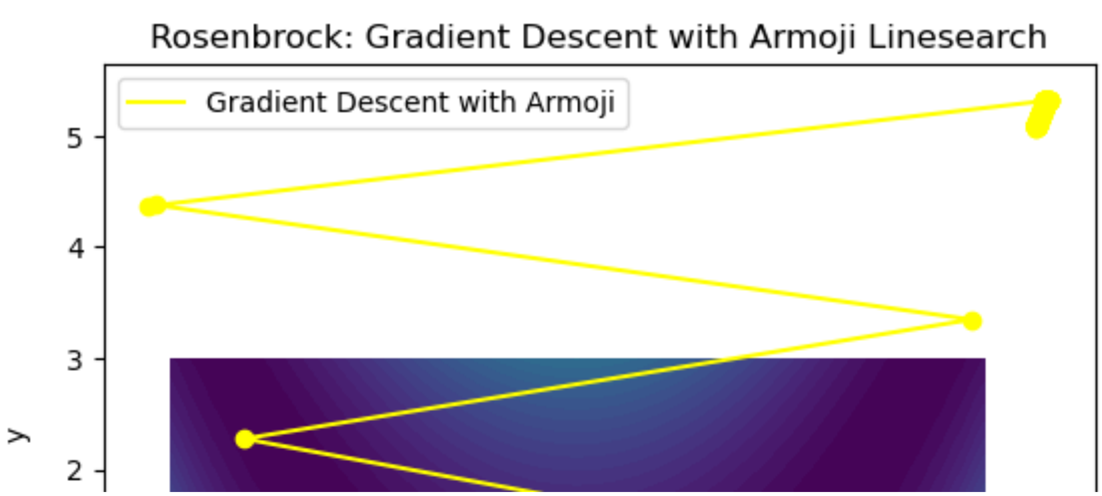
- Same as last week, but with adaptive step size

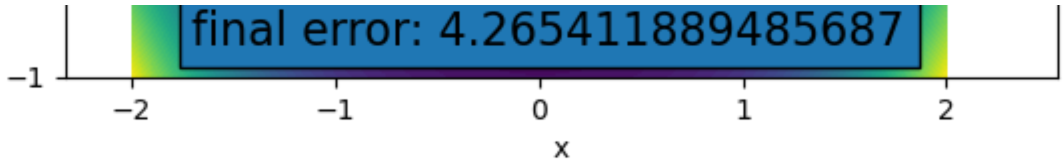
```
gradient_descent_wolfe (generic function with 1 method)
```

```
function gradient_descent_wolfe(f, x0, kmax)
    x = x0
    hist = []
    push!(hist, x)
    for k=1:kmax
        x = x + backtracking_linesearch_wolfe(
            f, x, -derivative(f, x), 1000, 0.95
        ) * -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
  - Even divergence for most stepsizes
- Now: Line search automatically choose a valid step size and we have an easy life






```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]

    # Gradient Descent with Armijo1
    res_gd_2d_rb_arm1 = gradient_descent_wolfe(h, x0, 1000)
    res_gd_2d_rb_arm1_x = [
        res_gd_2d_rb_arm1[2][i][1] for i=1:length(res_gd_2d_rb_arm1[2])
    ]
    res_gd_2d_rb_arm1_y = [
        res_gd_2d_rb_arm1[2][i][2] for i=1:length(res_gd_2d_rb_arm1[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 with Armoji Linesearch")

    # res_gd_2d_rb
    PyPlot.plot(res_gd_2d_rb_arm1_x, res_gd_2d_rb_arm1_y, color="yellow")
    scatter(res_gd_2d_rb_arm1_x, res_gd_2d_rb_arm1_y, color="yellow")
    # for i=1:length(res_gd_2d_rb_arm1_x)
    #   annotate(string(i), [res_gd_2d_rb_arm1_x[i], res_gd_2d_rb_arm1_y[i]],
color="w", zorder=2)
    # end

    legend(["Gradient Descent with Armoji"])

```

```
([2.25428, 5.07683], [[-1.0, 0.0], [1.47574, 1.22561], [-1.63813, 2.27885], [1.93862, 3.3
```

```julia
res_gd_2d_rb_arm1 # still not very fast (x*=[1,1]) 😓 (but robust!)
```

Trust-Region Methods

1. Given $x^{(k)}$
2. Replace f by (e.g 2nd order) approximation $\hat{f}$
3. Solve $\hat{x} = \text{argmin}_{x \in D_k} \hat{f}(x)$ 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 $\rho > \rho_{\text{min}}$, set $x^{(k+1)} = \hat{x}$, else decrease thrust region radius $\delta \leftarrow \sigma\delta$

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

```julia
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
```

- Define objective
- 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, ...)

```
([0.0, 0.707107], Any[[  more], 0], [[  more], 0.523018], Any[Float64[  more], 0.5230
```

```
trust_region(f, fhat, [0.,0.], solve_subproblem, 10, 0.5, 1.5, 0.9) # TODO, fixme
<=> implement solve_subproblem better
```

Trust-Region Method in Action 😎

x01 = 1.0 x02 = 0.5 k = 5 rhomin = 0.91 delta0 = 0.3 sigma = 0.51

```
    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)
```