present

Constrained Optimization: Penalty & Barrier Methods

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

System Setup for Binder

- See **@lamBOOO/teaching on Github**

```
begin
    ENV["MPLBACKEND"]="Agg"

    import Pkg
    Pkg.activate(mktempdir())

    # No python env to use Conda.jl
    ENV["PYTHON"]=""
    Pkg.add("PyCall")
    Pkg.build("PyCall")

    Pkg.add("PyPlot")
    using PyPlot
    Pkg.add("Calculus")
    using Calculus
    Pkg.add("PlutoUI")
    using PlutoUI
end
```

Define optimization problem

$$\min_{x \in \mathbb{R}^n} f(x) \text{ s.t. } \begin{cases} g_j(x) \leq 0 \text{ for } j = 1, \dots, m \\ h_i(x) = 0 \text{ for } i = 1, \dots, q \end{cases}$$

```
struct ConstrainedMinimizationProblem
    f::Function
    g::Array{Function,1}
    h::Array{Function,1}
end
```

```
p =
  ConstrainedMinimizationProblem(#617 (generic function with 1 method), Function[#618, #
```

```
p = ConstrainedMinimizationProblem(
    x -> 4*x[1]^2 - x[1] - x[2] - 2.5,
    [
        x -> -(x[2]^2 -1.5*x[1]^2 + 2*x[1] - 1),
        x -> +(x[2]^2 +2*x[1]^2 - 2*x[1] - 4.25),
    ],
    [x -> 5*(x[1]+x[2])],
)
```

Power Penalty Function

$$P_p(x, \alpha) = f(x) + \alpha r_p(x)$$

with

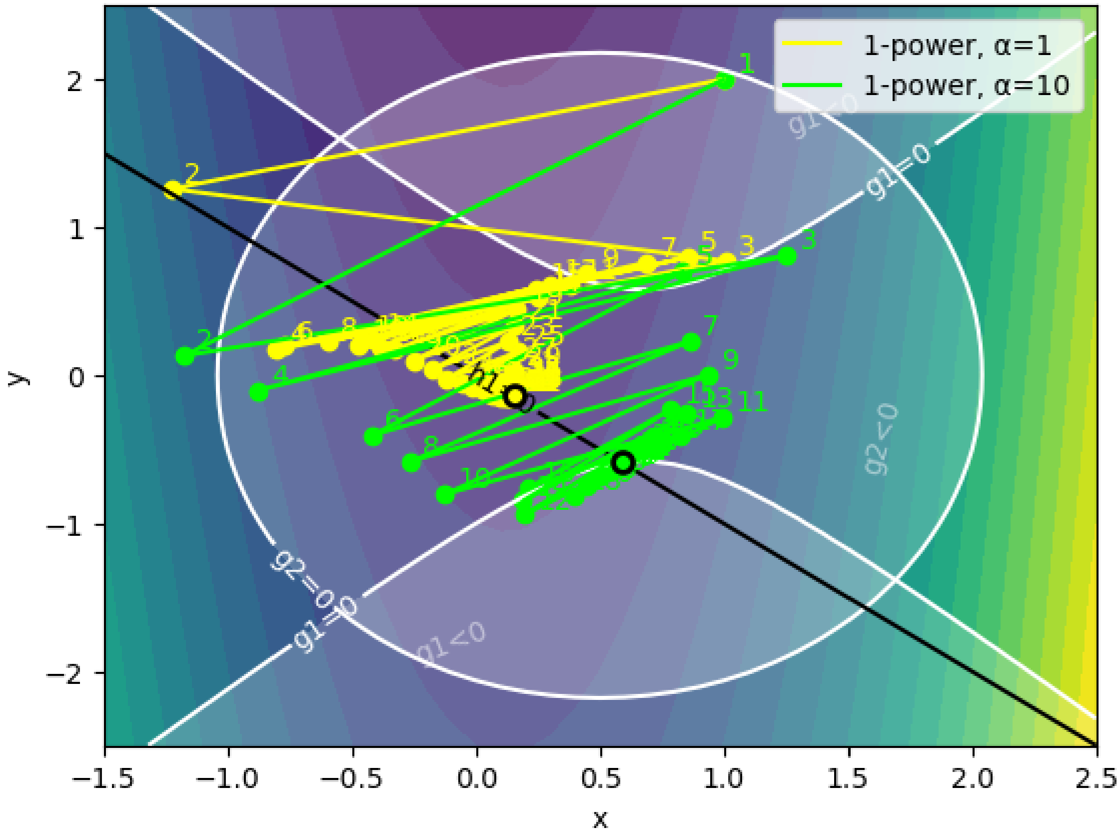
$$r_p(x) = \sum_{i=1}^{q} |h_i(x)|^p + \sum_{j=1}^{m} |\max(0, g_j(x))|^p$$

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

```
function P(x, p::ConstrainedMinimizationProblem, α::Number, pow::Int)
    @assert α>0
    r = (
        reduce(+, [abs(p.h[i](x))^pow for i ∈ 1:length(p.h)], init=0)
        + reduce(+, [max(0, p.g[i](x))^pow for i ∈ 1:length(p.g)], init=0)
    )
    return p.f(x) + α * r
end
```

Solve and Visualize Convergence History

steps = 50, penalty = ☐, αp = 1




```
visualize([
        gradient_descent_wolfe(x->P(x, p, 1, 1), [1,2], steps)[2],
        gradient_descent_wolfe(x->P(x, p, 10, 1), [1,2], steps)[2],
], p, legend = ["1-power, α=1", "1-power, α=10"],
    showotherfunction = if penalty x->P(x, p, αp, 1) else nothing end
)
```

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

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

$$\text{i)} \quad f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k),$$

$$\text{ii)} \quad -\mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq -c_2 \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k),$$

- Also assert that second Wolfe condition is fulfilled

wolfe1 (generic function with 1 method)

```
wolfe1(f, d, x, α) = f(x + α*d) <= f(x) + 1E-4 * α * derivative(f, x)' * d
```

wolfe2 (generic function with 1 method)

```
wolfe2(f, d, x, α) = derivative(f, x+α*d)' * d >= 0.99 * derivative(f, x)' * d
```

backtracking_linesearch_wolfe (generic function with 1 method)

```
function backtracking_linesearch_wolfe(f, x, d, αmax, β)
    # @assert wolfe2(f, d, x, backtracking_linesearch(f, x, d, αmax, wolfe1, β))
    return backtracking_linesearch(f, x, d, αmax, wolfe1, β)
end
```

Use Backtracking Algorithm in Gradient Descent

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), 1, 0.9
        ) * -derivative(f, x)
        push!(hist, x)
    end
    return x, hist
end
```

TODO: Implement Barrier Methods

See you NOT next week ✌️

Questions?

Exercises over

This was the last exercise on Wednesday

Exam Questions Session

15.03.21 15:00 Exam Questions Session (see Moodle)