Julia Tutorial for optimization and operations research

Part 03 - Julia for Optimization

Prof. Luiz-Rafael Santos - https://lrsantos11.github.io

UFSC, Brazil and MS&E/Stanford, US

Introduction to optimization

The general problem of nonlinear optimization (P) can be written as

$$\begin{aligned} \min f(x) \\ \text{s.t. } \ell \leq c(x) \leq u \end{aligned} \tag{P}$$

where $f : D_f \subset \mathbb{R}^n \to \mathbb{R}$ and $c : D_c \subset \mathbb{R}^n \to \mathbb{R}^m$.

- f is the *objective funcion*
- Since $c(x) = (c_1(x), \ldots, c_m(x))$, we call c_i as *constraints*
- The set $\Omega := \{x \in \mathbb{R}^n \mid \ell \leq c(x) \leq u\}$ is called the * feasibility set* ou *feasible set* of (P).
  - If $\Omega = \mathbb{R}^n$ we call (P) *unrestricted*
  - If $\Omega = \emptyset$, we call (P) é *infeasible*
- In particular, in this tutorial:
- f and each c_i are $C^1(\mathbb{R})$ ($C^2(\mathbb{R})$ if necessary)

Minimizers

- We say $x^* \in \Omega$ is a (global) *global minimizer* of (P) if

$$f(x^*) \leq f(x), \forall x \in \Omega$$

- $$x^* \in \Omega$$

  is a *local minimizer* of (P) is there exists $\delta > 0$ such that

$$f(x^*) \leq f(x), \forall x \in \Omega \cap \mathcal{B}_\delta(x^*)$$

Iterative methods

- Generate (with the computer) a sequence $(x_k)_{k\geq 0}$ such that the approximation x_{k+1} is well defined if

$$f(x_{k+1}) < f(x_k),$$

whenever $\nabla f(x_k) \neq 0$.

Descent direction

> **Definition (Descent direction).** We call $d \in \mathbb{R}^n$ a descent direction from x if there exists $\varepsilon > 0$ such that $f(x + \alpha d) < f(x)$ for all $\alpha \in (0, \varepsilon]$.

- Diretions that form an angle with less than 90 degrees with $\nabla f(x)$ are descent
- For instance, $d = -\nabla f(x)$ os the *stepest (or maximal) descent direcion from* x

Optimality conditions

- In this tutorial we will not discuss deeply the theory of *Otimality conditions *.

> **Theorem (1st order OC).** Let $f : \mathbb{R}^n \to \mathbb{R}$ smooth at x^*. If x^* is a local minimizer of f, thus
>
> $$\nabla f(x^*) = 0. \text{ (Stationary point)}$$

Basic optimization algorithm

- Step 0: Choose x_0; do $k = 0$.
- Step 1: If x_k an approximate *stationary point*, stop; other wise (\ (generic function with 174 methods)nabla f(x_k)>\varepsilon $), go to step 2.
- Step 2: find descent direction d_k and stepsize α_k such that $f(x_k + \alpha_k d_k) < f(x_k)$
- Step 3: compute

$$x_{k+1} \leftarrow x_k + \alpha_k d_k$$

- Step 4: do $k \leftarrow k + 1$ e go to step 1

Minimizing a unrestricted quadratic

- Let us solve the following Quadratic problem (QP)

$$\min_{x\in\mathbb{R}^n} f(x) = \frac{1}{2}x^TQx - q^Tx + q_0. \tag{QP}$$

with $Q \in \mathbb{R}^{n\times n}$ positive definite matrix.

- Note that

$$\nabla f(x) = Qx - q$$

```julia
using LinearAlgebra, Plots, Random
```

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

```julia
begin
    # Estrutura a e funções para a quadratica

    struct Quadratic
        Q::Matrix
        q::Vector
        q₀::Number
        Quadratic(Q, q, q₀) = new(Q, q, q₀)
    end

    obj(quad::Quadratic,x::Vector) = .5*dot(x,quad.Q*x) - dot(quad.q,x) + quad.q₀
    grad(quad::Quadratic,x::Vector) = quad.Q*x - quad.q


end
```

Toy-sample

$$\min_{x\in\mathbb{R}^n} f(x) = \frac{1}{2}x^TQx - q^Tx + q_0$$

$$Q = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, q = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \quad q_0 = 0 \Rightarrow \mathbf{x}^* = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$$

```
[2.0, -8.0]
```

```julia
begin
    Q = Float64[3 2; 2 6]
    q = Float64[2, -8]
end
```

```
xsol = [2.0, -2.0]
```

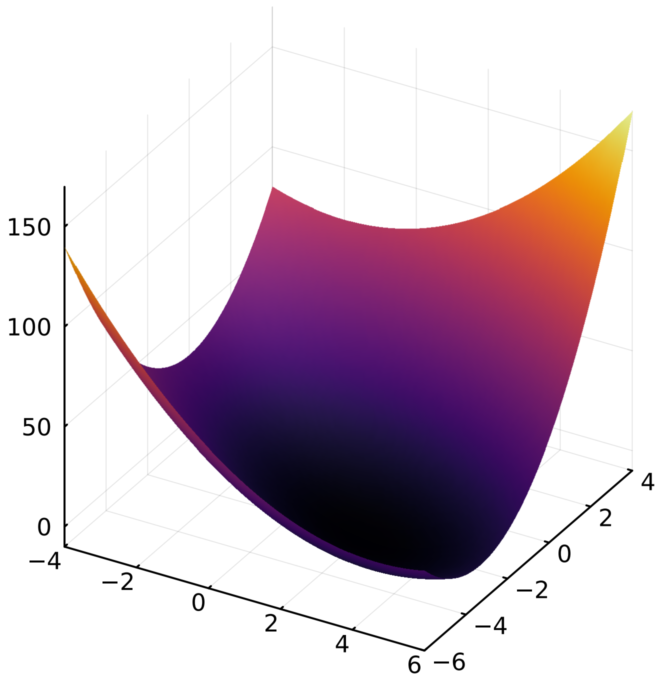
```julia
xsol = Q\q
```

```
quad =  Quadratic(2×2 Matrix{Float64}:, [2.0, -8.0], 0)
                 3.0  2.0
```

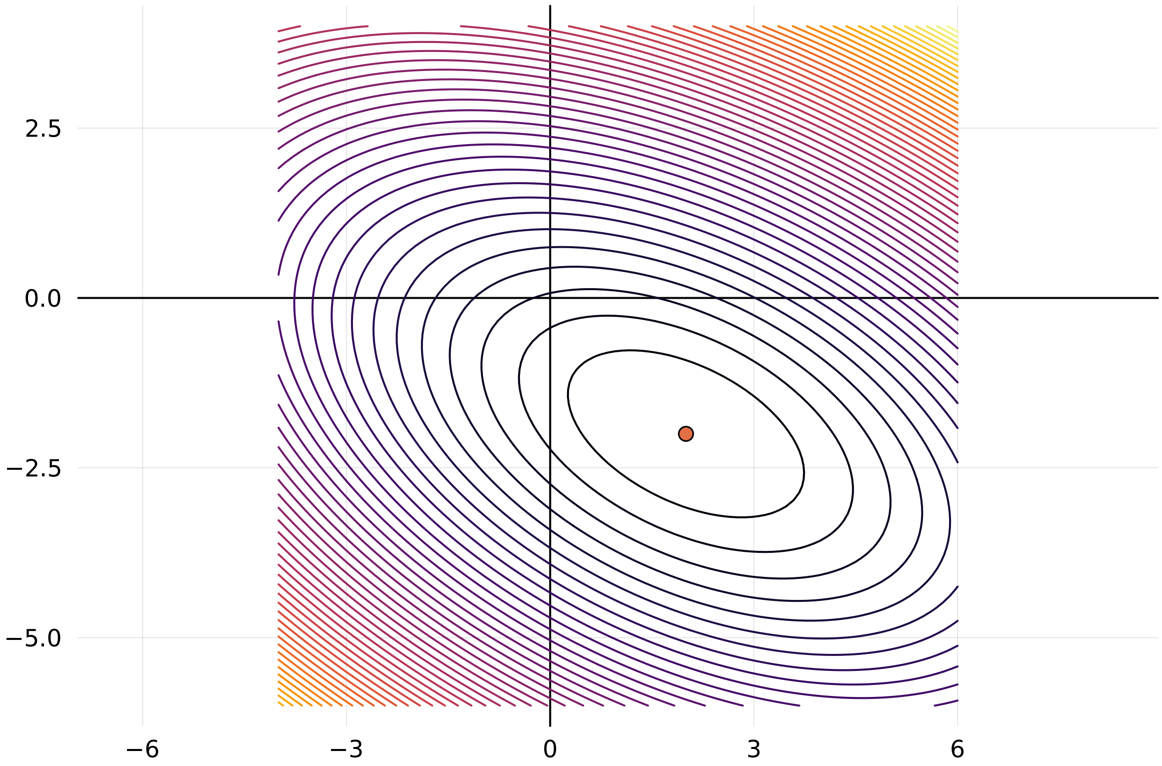
```
quad = Quadratic(Q,q,0)
```

```
quad_R2 (generic function with 1 method)
```

```
quad_R2(x,y) = obj(quad,[x,y])
```




```
begin
    x_test1 = range(-4,stop=6,length = 100)
    y_test1 = range(-6,stop=4,length = 100)
    z_test1 = quad_R2.(x_test1,y_test1)
    plt2 = surface(x_test1,y_test1,quad_R2,leg=false)
end
```




```
begin
    plt =
    contour(x_test1,y_test1,quad_R2,leg=false,framestyle=:zerolines,levels=50,aspect_ratio=:equal)
    scatter!([xsol[1]],[xsol[2]])
end
```

Gradient Descent Method

- In general model we do $d_k = -\nabla f(x_k) = b - Qx$ and we compute α_k such that

$$\alpha_k = \arg\min_{\alpha \geq 0} \varphi(\alpha) = f(x_k + \alpha d_k)$$

- The value of α_k has a closed formula (Why?)

$$\alpha_k = \frac{d_k^T d_k}{d_k^T Q d_k}$$

- **REMARK:** Two consecutive directions are orthogonal
- **Stopping criteria**: $\|d_k\|$ is small enough

Treasure game

- Click here to play

After playing a few times, you have probably noticed that the quickest way to find the deepest point at the bottom of the ocean is to pay attention to the slope found each time you throw your equipment: how steeply the bottom is and in which direction it was inclined. Although you cannot see the bottom and do not have a complete view of what it is like, the slope suggests where to continue the search.

GD Algorithm

- **Passo 0.** Defina x^0, $d^0 = b - Qx^0$ e $k = 0$
- **Passo 1.** Enquanto $d^k \neq 0$ faça
  - $$\alpha_k = \frac{(d^k)^T d^k}{(d^k)^T Q d^k}$$
  - $x^{k+1} = x^k + \alpha_k d^k$
  - $d_{k+1} = d_k - \alpha_k Q d_k$ (why?)
  - $k = k + 1$

iter_gradient (generic function with 1 method)

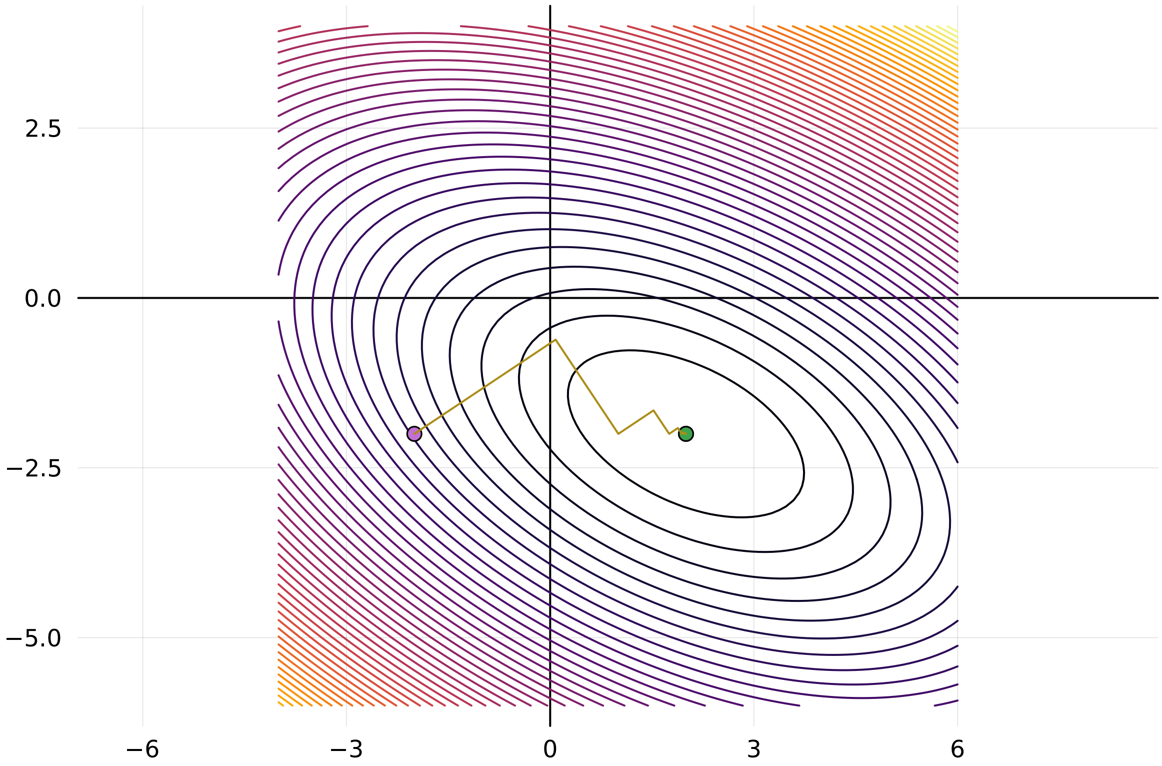
```
function iter_gradient(xₖ, dₖ,dotdₖ, quad)
    """
       Basic iteration of GD
    Parameters:
    xₖ: current iteration
    dₖ: current direction
    dotdₖ: inner product of dₖ
    quad: quadratic of interest
    """
    Qdₖ = quad.Q*dₖ

    αₖ = dotdₖ / dot(dₖ,Qdₖ)

    xₖ = xₖ +  αₖ*dₖ

    dₖ = dₖ - αₖ*Qdₖ

    dotdₖ = dot(dₖ,dₖ)

    return xₖ, dₖ, dotdₖ
end
```

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

```julia
    function gradient(quad::Quadratic,x₀::Vector;itmax::Int = 10, ε::Float64 = 1e-6)
        """
            Method of Gradient Descent
            Parameters:
            quad: Quadratic
            x₀: initial point
            itmax: maximum number of GD iterations
            ε: tolerance
        """
        k = 0
        xₖ = x₀
        dₖ = - grad(quad,xₖ)
        dotdₖ = dot(dₖ,dₖ)
        X =  xₖ
        while k <= itmax && dotdₖ >= ε^2 # equivalente a norm(dₖ) <= ε
            xₖ, dₖ, dotdₖ = iter_gradient(xₖ, dₖ, dotdₖ, quad)
            X = hcat(X,xₖ)
            k += 1 # equivale a k = k + 1
        end
        return X, k
    end
```

```
[2.0, -2.0]
```

```julia
begin
    x₀ = [-2.,-2]
    itmax = 30
    X, k = gradient(quad,x₀,itmax=itmax)
    @show norm(q - quad.Q*X[:,end])
    @show k
    X[:,end]
end
```




```
begin
    scatter!(plt,[xsol[1]],[xsol[2]])
    scatter!(plt,[x₀[1]],[x₀[2]])
    plot!(plt,X[1,:],X[2,:],st=:path)
end
```

Conjugate Gradient Method

Algorithm CG

- **Step 0.** Define x^0, $d^0 = r^0 = b - Qx^0$ and $k = 0$
- **Step 1. while $r^k = b - Qx^k \neq 0$ do**
  - $$\alpha_k \leftarrow \frac{(d^k)^T r^k}{(d^k)^T Q d^k}$$
  - $$x^{k+1} \leftarrow x^k + \alpha_k d^k$$

    where d^k does not makes *zig-zag*
  - $$k \leftarrow k + 1$$

How to find conjugate directions d^k'?

Algorithm CG

(updating residuals)

- **Step 0.** Define x^0, $d^0 = r^0 = b - Qx^0$ and $k = 0$
- **Step 1. while** $r^k = b - Qx^k \neq 0$ **do**
  - $$\alpha_k \leftarrow \frac{(d^k)^T r^k}{(d^k)^T Q d^k}$$
  - $$x^{k+1} \leftarrow x^k + \alpha_k d^k$$
  - $$r^{k+1} = r^k - \alpha^k Q d^k$$
  - $$k \leftarrow k+1$$

- **Proposition.**

$$\mathcal{D}_k := \operatorname{span}\left\{d^0, \ldots, d^k\right\} = \operatorname{span}\left\{r^0, \ldots, r^k\right\},\ k = 0, > \ldots, n-1$$

Lemma. $r^{k+1} \perp \mathcal{D}_k = \operatorname{span}\left\{d^0, \ldots, d^k\right\} = \operatorname{span}\left\{r^0, \ldots, r^k\right\},\ k = 0, \ldots, n-1$

Theorem. $\mathcal{D}_k = \operatorname{span}\left\{r^0, Qr^0, Q^2 r^0, \ldots, Q^k r^0\right\},\ k = 0, \ldots, n-1$

- Left-hand side subspace is called *Krylov subspace of dimension k+1* given by Q and r^0 and it is denoted by $\mathcal{K}_{k+1}(Q, r^0)$

Corollary. $r^k \perp_Q \mathcal{D}_{k-2}$, for $k = 0, \ldots, n-1$, *i.e.*, r^k is Q-conjugate to d^j, $j < k-2$.

Practical CG

- Making $k = k-1$ we get (using $d^k = r^k + \beta_{k-1} d^{k-1}$)

$$\alpha_k = \frac{(d^k)^T r^k}{(d^k)^T Q d^k} = \frac{(r^k)^T r^k}{(d^k)^T Q d^k}$$

$$\beta_k = -\frac{(r^{k+1})^T Q d^k}{(d^k)^T Q d^k} = \frac{(r^{k+1})^T r^{k+1}}{(r^k)^T r^k}$$

Algorithm CG

(updating residuals and conjugate directions)

- **Step 0.** Define x^0, $d^0 = r^0 = b - Qx^0$ and $k = 0$
- **Step 1. while** $r^k = b - Qx^k \neq 0$ **do**
  - $$\alpha_k \leftarrow \frac{(r^k)^T r^k}{(d^k)^T Q d^k}$$
  - $x^{k+1} \leftarrow x^k + \alpha_k d^k$
  - $r^{k+1} \leftarrow r^k - \alpha_k Q d^k$
  - $$\beta_k \leftarrow \frac{(r^{k+1})^T r^{k+1}}{(r^k)^T r^k}$$
  - $d^{k+1} \leftarrow r^{k+1} + \beta_k d^k$
  - $k \leftarrow k + 1$

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

```julia
begin
    function iter_CG(xₖ, rₖ, dotrₖ,dₖ, quad, k)
        """
        Iteração basica de CG
        Parâmetros:
        xₖ: iteração atual
        rₖ: residuo atual
        dotrₖ: prod interno rₖ
        dₖ: direção atual
        quad: quadratica de interesse
        """
        Qdₖ = quad.Q*dₖ

        αₖ = dotrₖ/dot(dₖ,Qdₖ)

        xₖ = xₖ + αₖ*dₖ

        if mod(k,50) != 0
            rₖ = rₖ - αₖ*Qdₖ
        else
            rₖ = -grad(quad,xₖ)
        end

        dotrₖ_old = dotrₖ

        dotrₖ = dot(rₖ,rₖ)

        βₖ = dotrₖ/dotrₖ_old

        dₖ = rₖ +  βₖ*dₖ


        return xₖ, rₖ, dotrₖ, dₖ


    end


end
```

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

```julia
function CG(quad::Quadratic,x₀::Vector;itmax::Int = 10,ε::Float64 = 1e-8)
        """
      Método de Gradientes Conjugados
      Parâmetros:
      quad: Quadratica
      x₀: ponto inicial
      itmax: número max de iterçãoes de CG
      ε: tolerância
     """
      xₖ = x₀
      rₖ = -grad(quad,xₖ)
      dₖ = copy(rₖ)
      k = 0
      X = xₖ
      dotrₖ = dot(rₖ,rₖ)
      while k <= itmax && dotrₖ >= ε^2
          xₖ, rₖ, dotrₖ, dₖ = iter_CG(xₖ, rₖ, dotrₖ,dₖ, quad, k)
          X = hcat(X,xₖ)
          k += 1
      end
      return X, k
  end
```

Exemplos com `BigFloat` e matrizes maiores

```
2.220446049250313e-16
```

```julia
eps()
```

```
π = 3.1415926535897...
```

```julia
pi
```

```
3.141592653589793238462643383279502884197169399375105820974944592307816406286198
```

```julia
BigFloat(pi)
```

```
1.727233711018888925077270372560079914223200072887256277004740694033718360632485e-77
```

```julia
eps(BigFloat)
```

How about non quadratic functions?

- **Example** Rosenbrock functions

$$f(x,y) = (a-x)^2 + b(y-x^2)^2$$

- Local minimizer at (a,a^2) with $f(a,a^2) = 0$
- Quadratic model at x_k

$$m_k(d) = f(x_k) + \nabla f(x_k)^T d + \frac{1}{2}d^T\nabla^2 f(x_k)d$$

- Minimum of f m_k, if $\nabla^2 f(x_k)$ is SPD is the unique solution of linear system

$$\nabla^2 f(x_k)d = -\nabla f(x_k)$$

```
md"""

# How about non quadratic functions?

* **Example**  [Rosenbrock](https://en.wikipedia.org/wiki/Rosenbrock_function)
functions

```math
    f(x,y) = (a-x)^2 + b(y-x^2)^2
```

- Local minimizer at (a,a^2) with $f(a,a^2) = 0$

* Quadratic model at ``x_k``

```math
m_k(d) = f(x_k) + \nabla f(x_k)^Td + \frac{1}{2}d^T\nabla^2f(x_k)d
```


* Minimum of f m_k, if $\nabla^2f(x_k)$ is SPD is the unique solution of linear
system

```math
\nabla^2f(x_k)d = -\nabla f(x_k)
```
"""
```

How to compute derivatives?

- Automatic differentiation package: `ForwardDiff.jl` to compute $\nabla f(x)$ and $\nabla^2 f(x)$

```
md"""
#### How to compute derivatives?

* Automatic differentiation package:  [`ForwardDiff.jl`]
(https://github.com/JuliaDiff/ForwardDiff.jl) to compute $\nabla f(x)$ and
$\nabla^2 f(x)$
"""
```

```julia
using ForwardDiff
```

f (generic function with 1 method)

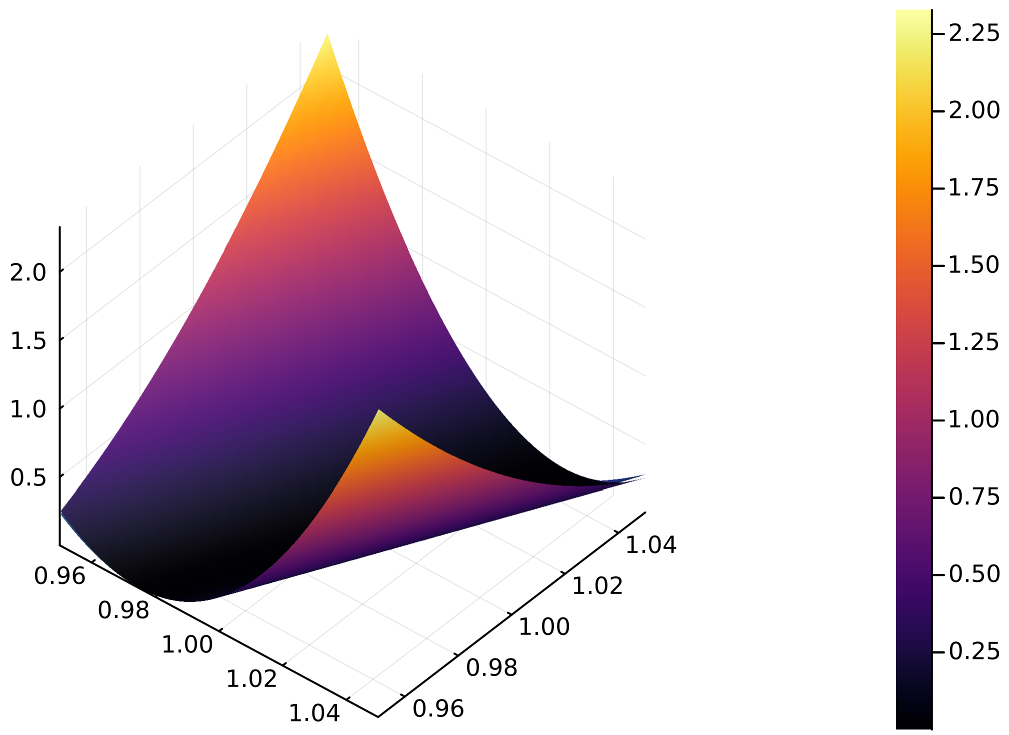
```julia
f(x) = (1-x[1])^2 + 100 * (x[2] - x[1]^2)^2
```

∇f (generic function with 1 method)

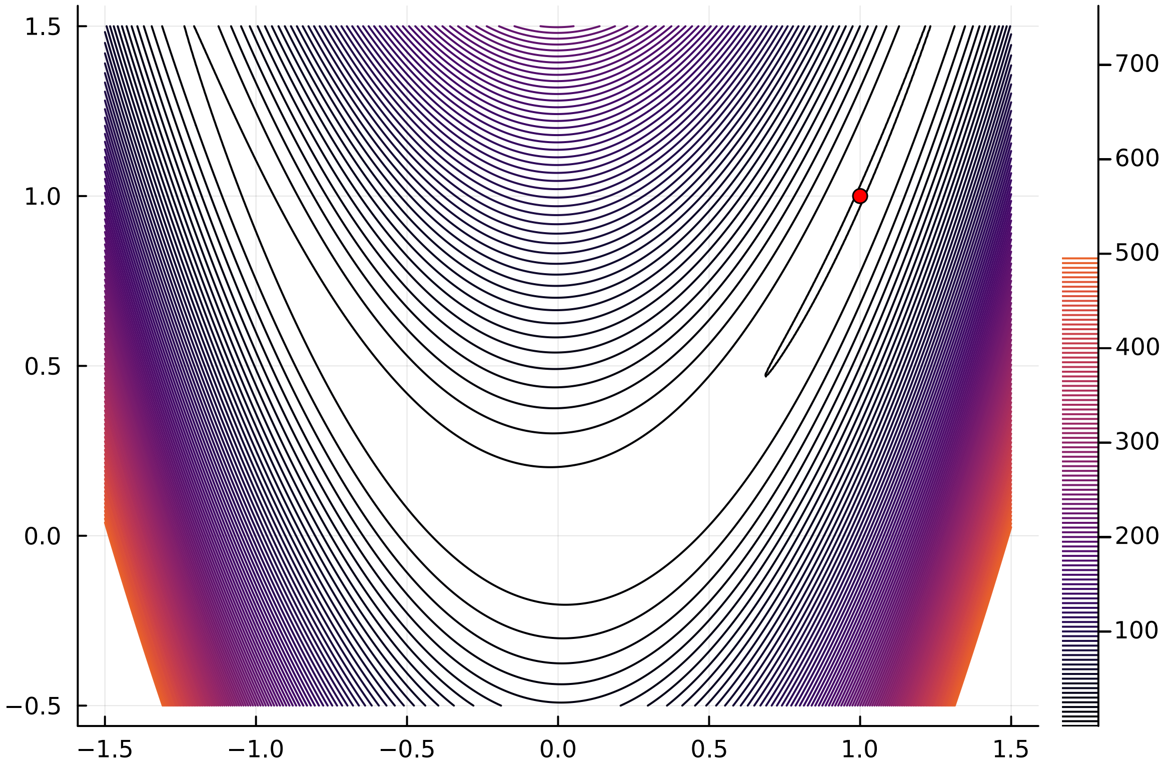
```julia
∇f(x) = ForwardDiff.gradient(f, x)
```

H (generic function with 1 method)

```julia
H(x) = ForwardDiff.hessian(f, x)
```




```julia
let
    x₀ = [1.0; 1.0]

    mₖ(d) = f(x₀) + dot(∇f(x₀), d) + dot(H(x₀) * d, d) / 2
    q(x) = mₖ(x - x₀)

    a, b = 0.95,1.05
    surface(
        range(a,b, length=50),
        range(a, b, length=50),
        (x,y) -> f([x;y]),
        linealpha = 0.3,
        fc=:thermal,
        camera = (40,40))
    surface!(
        range(a, b, length=50),
        range(a, b, length=50),
        (x,y) -> q([x;y]),
    )
end
```




```
let
    x = range(-1.5, 1.5, length=400)
    y = range(-0.5 , 1.5, length=400)
    contour(x,y,(x,y) -> f([x;y]),levels=0.1:5.0:500)
    scatter!([1.0],[1.0],c=:red,label=:false)
end
```

Newton Method

Newton for nonlinear systems of equations $F(x) = 0$

$$x_{k+1} = x_k - J_F(x^k)^{-1}F(x^k)$$

Newton Method for Optimization

- $$x^*$$

  such that $\nabla f(x^*) = 0$. Define $F := \nabla f$ we have

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1}\nabla f(x_k)$$

since $J_{\nabla f}(x) = \nabla^2 f(x)$.

- Direction $d = -(\nabla^2 f(x_k))^{-1}\nabla f(x_k)$ is exactlye the solution of quadratic model
- If $\nabla^2 f(x)$ os PSD, Newton direction is *descent* (Why?)
- Compute α_k by using linear search: exact (as in GD) or inexact search (Armijo)

newton (generic function with 1 method)

```julia
function newton(f, ∇f, H, x₀::Vector; itmax = 10_000,ε = 1e-6)
    k = 0
    xₖ = x₀
    gradₖ = ∇f(xₖ)
    while k <= itmax && norm(gradₖ) >= ε
        d = - (H(xₖ)\gradₖ)
        @info xₖ = xₖ + d
        gradₖ = ∇f(xₖ)
        k += 1
    end
    return xₖ, k
end
```

([1.0, 1.0], 5)

```julia
let
    x₀ = [10,10.]
    xsol, num_iter = newton(f, ∇f, H, x₀)
end
```

```julia
begin

    using PlutoUI
    using PlutoReport
    using HypertextLiteral: @htl, @htl_str

    struct Foldable{C}
        title::String
        content::C
    end

    function Base.show(io, mime::MIME"text/html", fld::Foldable)
        write(io,"<details><summary>$(fld.title)</summary><p>")
        show(io, mime, fld.content)
        write(io,"</p></details>")
    end

    struct TwoColumn{L, R}
        left::L
        right::R
    end

    function Base.show(io, mime::MIME"text/html", tc::TwoColumn)
        write(io, """<div style="display: flex;"><div style="flex: 50%;">""")
        show(io, mime, tc.left)
        write(io, """</div><div style="flex: 50%;">""")
        show(io, mime, tc.right)
        write(io, """</div></div>""")
    end
    apply_css_fixes()
    # @bind _pcon presentation_controls(aside=true)
end
```

```julia
# presentation_ui(_pcon)
```