MTH8408 : Méthodes d'optimisation et contrôle optimal

Laboratoire 2: Optimisation sans contraintes

Travail réalisé par Julien Pallage

Matricule: 2012861

1er février 2024

In [ ]:
```
using Pkg
Pkg.activate("..") #Accède au fichier Project.toml
Pkg.add("LDLFactorizations")
Pkg.add("OptimizationProblems")
Pkg.add("Plots")
Pkg.instantiate()
Pkg.status()
```

```
  Activating project at `~/Documents/code/MTH8408-Hiv24`
   Resolving package versions...
  No Changes to `~/Documents/code/MTH8408-Hiv24/Project.toml`
  No Changes to `~/Documents/code/MTH8408-Hiv24/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Documents/code/MTH8408-Hiv24/Project.toml`
  No Changes to `~/Documents/code/MTH8408-Hiv24/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Documents/code/MTH8408-Hiv24/Project.toml`
  No Changes to `~/Documents/code/MTH8408-Hiv24/Manifest.toml`
Status `~/Documents/code/MTH8408-Hiv24/Project.toml`
  [54578032] ADNLPModels v0.7.0
  [b6b21f68] Ipopt v1.6.0
  [4076af6c] JuMP v1.18.1
  [40e66cde] LDLFactorizations v0.10.1
  [b964fa9f] LaTeXStrings v1.3.1
^ [b8f27783] MathOptInterface v1.25.1
  [a4795742] NLPModels v0.20.0
  [f4238b75] NLPModelsIpopt v0.10.1
  [792afdf1] NLPModelsJuMP v0.12.5
  [5049e819] OptimizationProblems v0.7.3
  [91a5bcdd] Plots v1.40.0
  [37e2e46d] LinearAlgebra
Info Packages marked with ^ have new versions available and may be upgradable.
```

In [ ]:
```
using ADNLPModels, LinearAlgebra, NLPModels, Printf
```

On pourra trouver de la documentation sur `ADNLPModels` et `NLPModels` ici:

- juliasmoothoptimizers.github.io/NLPModels.jl/dev/
- juliasmoothoptimizers.github.io/ADNLPModels.jl/dev/

In [ ]:
```
# Problème test:
f(x) = x[1]^2 * (2*x[1] - 3) - 6*x[1]*x[2] * (x[1] - x[2] - 1) # fonction objectif vue en classe
g(x) = 6 * [x[1]^2 - x[1] - 2*x[1]*x[2] + x[2]^2 + x[2]; -x[1]^2 + 2*x[1]*x[2] + x[1]] # le gradient de f
H(x) = 6 * [2*x[1]-1-2*x[2] -2*x[1]+2*x[2]+1; -2*x[1]+2*x[2]+1 2*x[1]] # la Hessienne de f
```

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

Exercice 1: Newton avec recherche linéaire - amélioration du code

Ci-dessous, vous avez le code de deux fonctions qui ont été vues dans le cours, la recherche linéaire qui satisfait Armijo, et une méthode de Newton avec cette recherche linéaire. Le but de ce laboratoire est d'implémenter d'autres méthodes utiles pour résoudre des problèmes de grandes dimensions.

In [ ]:
```
#Amélioration possibles: return also the value of f

function armijo(xk, dk, fk, gk, f)
  slope = dot(gk, dk) #doit être <0
  t = 1.0
  while f(xk + t * dk) > fk + 1.0e-4 * t * slope
    t /= 1.5
  end
  return t
end

function armijo_mod(xk, dk, fk, gk, nlp)
  slope = dot(gk, dk) #doit être <0
  t = 1.0
  while obj(nlp, xk + t * dk) > fk + 1.0e-4 * t * slope
    t /= 1.5
  end
  return t
end
```

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

In [ ]:
```
#Test pour vérifier que la fonction armijo fonctionne correctement.
using Test #le package Test définit (entre autre) la macro @test qui permet de faire des tests unitaires :-)
xk = ones(2) # [1,1]
gk = g(xk) # gradient
dk = - gk # direction is minus the gradient
fk = f(xk) # objective function evaluated at xk
```

```
t  = armijo(xk, dk, fk, gk, f) # armijo to find a t
@test t < 1
@test f(xk + t * dk) <= fk + 1.0e-4 * t * dot(gk,dk)

xk = [1.5, 0.5]
fk = f(xk)
gk = g(xk)
dk = - gk
t  = armijo(xk, dk, fk, gk, f)
@test t < 1
@test f(xk + t * dk) <= f(xk) + 1.0e-4 * t * dot(g(xk),dk)
```

```
Test Passed
```

In [ ]:
```
function newton_armijo(f, g, H, x0; verbose::Bool = true, epsilon_abs = 1.0e-6, epsilon_rel = 1.0e-6, max_iter::Int = 10
  xk  = x0 # initialize xk at x0
  fk  = f(xk) # evaluate the objective function at xk
  gk = g(xk) # get gradient
  gnorm = gnorm0 = norm(gk) # get the norm of the gradient
  k = 0 # round 0
  verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
  verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
  while gnorm > epsilon_abs + epsilon_rel * gnorm0 && k < max_iter # while the stopping conditions is not met
    Hk = H(xk) # get the hessian
    dk = - Hk \ gk # find the direction (just like inv(Hk)@gk)
    slope = dot(dk, gk) # slope= direction@gradient
    λ = 0.0
    while slope ≥ -1.0e-4 * norm(dk) * gnorm
      λ = max(1.0e-3, 10 * λ)
      dk = - ((Hk + λ * I ) \ gk)
      slope = dot(dk, gk)
    end
    t = armijo(xk, dk, fk, gk, f)
    xk += t * dk
    fk = f(xk)
    gk = g(xk)
    gnorm = norm(gk)
    k += 1
    verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
  end
  return xk
end
```

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

In [ ]:
```
sol  = newton_armijo(f, g, H, [1.5, .5])
@test g(sol) ≈ zeros(2) atol = 1.0e-6
```

```
k        fk ||∇f(x)||
0  0.00e+00   4.5e+00
1 -9.49e-01   8.4e-01 1.0e+00
2 -1.00e+00   7.6e-02 1.0e+00
3 -1.00e+00   9.1e-04 1.0e+00
4 -1.00e+00   1.4e-07 1.0e+00
Test Passed
```

On veut améliorer le code de la fonction `newton_armijo` avec les ajouts suivants:

- Changer les paramètre d'entrées de la fonction pour un `nlp` -> DONE
- Avant d'appeler la recherche linéaire, si `slope = dot(dk, gk)` est plus grand que `-1.0e-4 * norm(dk) * gnorm`, on modifie le système. On fait maximum 5 mise à jour de `λ`, sinon on prend l'opposé du gradient.

```
λ = 0.0
while slope ≥ -1.0e-4 * norm(dk) * gnorm
  λ = max(1.0e-3, 10 * λ)
  dk = - ((Hk + λ * I ) \ gk)
  slope = dot(dk, gk)
end
```

Ajouter un compteur sur le nombre de mises à jour de `λ` et ajuster `dk = - gk` si la limite est atteinte. -> DONE

- On veut aussi détecter et éventuellement arrêter la boucle `while` si la fonction objectif `fk` devient trop petite/négative (inférieure à `-1e15`), i.e. le problème est non-bornée inférieurement. -> DONE
- On veut ajouter deux critères d'arrêts supplémentaires:
  - un compteur sur le nombre d'évaluations de f (maximum 1000). Utiliser `neval_obj(nlp)`. ->DONE
  - une limite de temps d'execution, `max_time = 60.0`. Utiliser la fonction `time()`. -> DONE
- Enfin, on voudrait aussi voir un message à l'écran si l'algorithme n'a pas trouvé la solution, i.e. il s'est arrêté à cause de la limite sur le nombre d'itérations, temps, évaluation de fonctions, problème non-borné ->DONE

In [ ]:
```
#SOLUTION: fonction à modifier
function newton_armijo_v2(nlp, x0, verbose::Bool = true, epsilon_abs = 1.0e-6, epsilon_rel = 1.0e-6, max_iter::Int = 100
  historique = [x0]
  start_time = time()
  xk  = x0 # initialize xk at x0
  fk  = obj(nlp, xk) # evaluate the objective function at xk
  gk = grad(nlp, xk) # get gradient
  gnorm = gnorm0 = norm(gk) # get the norm of the gradient
  k = 0 # round 0
  error = false

  verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
  verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
  while gnorm > epsilon_abs + epsilon_rel * gnorm0   # while the stopping conditions is not met
    Hk = hess(nlp, xk) # get the hessian
    dk = - Hk \ gk # find the direction (just like inv(Hk)@gk)
    slope = dot(dk, gk) # slope= direction@gradient
    λ = 0.0
```

```
    lam_counter = 0
    while slope ≥ -1.0e-4 * norm(dk) * gnorm && lam_counter < max_lam # ADDED lam_counter
      λ = max(1.0e-3, 10 * λ)
      dk = - ((Hk + λ * I ) \ gk)
      slope = dot(dk, gk)
      lam_counter += 1
      if lam_counter == 5
        dk = -gk
      end
    end
    t = armijo_mod(xk, dk, fk, gk, nlp)
    xk += t * dk
    push!(historique, xk)
    fk = obj(nlp, xk)
    gk =  grad(nlp, xk)
    gnorm = norm(gk)
    k += 1

    # prints
    if fk <= lower_bound
      xk = -Inf64
      @printf "The problem is unbounded below."
      error = true
      break
    elseif k > max_iter
      @printf "Maximal number of iterations has been reached"
      error = true
      break
    elseif (time() - start_time) >= max_time
      @printf "Timeout has been reached"
      error = true
      break

    elseif neval_obj(nlp) > max_eval
      @printf "Max number of evaluations has been reached"
      error = true
      break
    end
    verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
  end
  if error == false
    println("An optimal solution has been found in $(time() - start_time) seconds")
  else
    println("An error occured during solving")
  end
  return xk, obj(nlp, xk), historique
end
```

```
newton_armijo_v2 (generic function with 9 methods)
```

In [ ]:
```
#Test
f(x) = x[1]^2 * (2*x[1] - 3) - 6*x[1]*x[2] * (x[1] - x[2] - 1) # fonction objectif vue en classe
x0 = zeros(2)
x0[1] = 1.5
x0[2] = 0.5

nlp = ADNLPModel(f, x0)

arg, star, histo = newton_armijo_v2(nlp, x0);

print("argmin est $arg \n")
println("On trouve une fonction objectif de $star ")

print(histo)
```

```
k        fk ||∇f(x)||
0  0.00e+00   4.5e+00
1 -9.49e-01   8.4e-01 1.0e+00
2 -1.00e+00   7.6e-02 1.0e+00
3 -1.00e+00   9.1e-04 1.0e+00
4 -1.00e+00   1.4e-07 1.0e+00
An optimal solution has been found in 0.04477810859680176 seconds
argmin est [1.0000000232305737, 2.3230573680167342e-8]
On trouve une fonction objectif de -0.9999999999999983
[[1.5, 0.5], [1.125, 0.125], [1.0125, 0.012499999999999997], [1.00015243902439, 0.00015243902439024404], [1.0000000232305
737, 2.3230573680167342e-8]]
```

Exercice 2: LDLt-Newton avec recherche linéaire

On va maintenant modifier la méthode de Newton vu précédemment pour utiliser un package qui s'occupe de calculer une factorisation de la matrice hessienne tel que:

$$\nabla^2 f(x) = LDL^T.$$

Ce type de factorisation n'est possible que si la matrice hessienne est définie positive, dans le cas contraire on a besoin de régularisé le système comme dans l'exercice précédent.

Pour résoudre le système linéaire en utilisant cette factorisation, on va utiliser le package `LDLFactorizations` :

In [ ]:
```
using LDLFactorizations, LinearAlgebra
```

Un tutoriel sur l'utilisation de `LDLFactorizations` est disponible sur la documentation du package sur github ou encore à ce lien.

Voici un exemple d'utilisation de ce package. La matrice dont on veut calculer la factorisation doit être de type `Symmetric` .

In [ ]:
```
A = ones(2,2) #cette matrice symétrique, mais pas du type Symmetric
              #à noter que cette matrice n'est pas définie positive.
typeof(A) <: Symmetric #false
```

```
A = Symmetric(A)
typeof(A) <: Symmetric #true :)
display(A)
```

```
2×2 Symmetric{Float64, Matrix{Float64}}:
 1.0  1.0
 1.0  1.0
```

Deuxième étape, le package fait une phase d'analyse de la matrice avec `ldl_analyze` en créant une structure pratique pour les diverses fonctions du package.

In [ ]:
```
A = -rand(2, 2)
sol = rand(2)
b = A*sol #on veut résoudre le système A*x=b
display(A)
# LDLFactorizations va en réalité demander la matrice triangulaire supérieure
A = Symmetric(triu(A), :U)
display(A)

S = ldl_analyze(A)
display(S)

ldl_factorize!(A, S)
display(S)
x = S \ b # x = A \b ça va être résolu par Julia
norm(A * x - b)
```

```
2×2 Matrix{Float64}:
 -0.464877  -0.261397
 -0.714503  -0.0531855
2×2 Symmetric{Float64, Matrix{Float64}}:
 -0.464877  -0.261397
 -0.261397  -0.0531855
LDLFactorizations.LDLFactorization{Float64, Int64, Int64, Int64}(true, false, true, 2, [2, -1], [1, 0], [2, 2], [1, 2], [1, 2], [1, 2, 2], [1, 1, 1], Int64[], [1], [5.0e-324], [8.0e-323, 0.0], [0.0, 4.238951974945878e175], [16, 0], 0.0, 0.0, 0.0, 2)
LDLFactorizations.LDLFactorization{Float64, Int64, Int64, Int64}(true, true, true, 2, [2, -1], [1, 0], [2, 2], [1, 2], [1, 2], [1, 2, 2], [1, 1, 1], Int64[], [2], [0.5622922606330362], [-0.46487678992515935, 0.09379575375708132], [0.0, 0.0], [1, 1], 0.0, 0.0, 0.0, 2)
1.1102230246251565e-16
```

In [ ]:
```
A = [0. 1.; 1. 0.]
```

```
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0
```

In [ ]:
```
A = Symmetric(triu(A), :U)
display(A)
S = ldl_analyze(A)
ldl_factorize!(A, S)
```

```
2×2 Symmetric{Float64, Matrix{Float64}}:
 0.0  1.0
 1.0  0.0
LDLFactorizations.LDLFactorization{Float64, Int64, Int64, Int64}(true, false, true, 2, [2, -1], [0, 0], [1, 2], [1, 2], [1, 2], [1, 2, 2], [1, 1, 1], Int64[], [1], [5.0e-324], [0.0, 0.0], [0.0, 1.0e-323], [241, -1073741824], 0.0, 0.0, 0.0, 2)
```

In [ ]:
```
S.L
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 1 stored entry:
 5.0e-324   ⋅
  ⋅         ⋅
```

La matrice `A` factorisée par LDL^T n'était pas forcément définie positive. On peut le voir sur les valeurs de D.

In [ ]:
```
S.d #c'est le vecteur qui correspond à la matrice diagonale D.
```

```
2-element Vector{Float64}:
 0.0
 0.0
```

Pour l'optimisation, dans le cas où des valeurs de D sont négatives, i.e. `minimum(S.d) <= 0.`, on ajoutera une correction pour être sûr d'obtenir une direction de descente. On pourra choisir un des deux:

- `S.d   = abs.(S.d)`
- `S.d .+= -minimum(S.d) + 1e-6`

Utiliser cette technique pour calculer la direction de descente:

In [ ]:
```
# Solution: modifier le calcul de la direction avec LDLFactorizations
function newton_ldlt_armijo(nlp, x0, verbose::Bool = true)
  historique = [x0]
  xk  = x0
  fk  = obj(nlp, xk)
  gk = grad(nlp, xk)
  gnorm = gnorm0 = norm(gk)
  k = 0
  verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
  verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
  while gnorm > 1.0e-6 + 1.0e-6 * gnorm0 && k < 100 && fk > -1e15
    Hk = Symmetric(triu(hess(nlp, xk)), :U)
    # ... TODO ...
    Sk = ldl_analyze(Hk) # added
    ldl_factorize!(Hk, Sk) # added
    Sk.d = abs.(Sk.d) # added
    dk = - Sk \ gk
    slope = dot(dk, gk)
    t = armijo_mod(xk, dk, fk, gk,nlp)
```

```
    xk += t * dk
    push!(historique, xk)
    fk = obj(nlp, xk)
    gk = grad(nlp, xk)
    gnorm = norm(gk)
    k += 1
    verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
  end
  return xk, obj(nlp, xk), historique
end
```

```
newton_ldlt_armijo (generic function with 2 methods)
```

In [ ]:

```
#Test
f(x) = x[1]^2 * (2*x[1] - 3) - 6*x[1]*x[2] * (x[1] - x[2] - 1)
x0 = zeros(2)
x0[1] = 1.5
x0[2] = 0.5
nlp = ADNLPModel(f, x0)


arg, star = newton_ldlt_armijo(nlp, x0)

print("argmin est $arg \n")
println("On trouve une fonction objectif de $star ")
```

```
k        fk ||∇f(x)||
0  0.00e+00   4.5e+00
1 -9.49e-01   8.4e-01 1.0e+00
2 -1.00e+00   7.6e-02 1.0e+00
3 -1.00e+00   9.1e-04 1.0e+00
4 -1.00e+00   1.4e-07 1.0e+00
argmin est [1.0000000232305737, 2.3230573678432618e-8]
On trouve une fonction objectif de -0.9999999999999983
```

Exercice 3: Méthode quasi-Newton: BFGS

Méthode quasi-Newton: BFGS

Pour des problèmes de très grandes tailles, il est parfois très coûteux d'évaluer la hessienne du problème d'optimisation (et même le produit hessienne-vecteur). La famille des méthode *quasi-Newton* construit une approximation B_k symétrique de la matrice Hessienne en utilisant seulement le gradient et en mesurant sa variation, et permet quand même d'améliorer significativement les performances comparé à la méthode du gradient.

$$s_k = x_{k+1} - x_k, \quad y_k = \nabla f(x_{k+1}) - \nabla f(x_k).$$

Par ailleurs la matrice B_k est aussi construite de façon à ce que l'inverse soit connue, il n'y a donc pas de système linéaire à résoudre.

La méthode la plus connue dans la famille des méthodes quasi-Newton, est la méthode BFGS (Broyden - Fletcher, Goldfarb, and Shanno) où B_k est définir positive ($B_0 = \lambda I,\ \lambda > 0$). La formule suivante calcule l'inverse de B_k que l'on note H_k:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T, \quad \rho_k = \frac{1}{y_k^T s_k}.$$

L'algorithme est presque le même que la méthode de Newton à la différence qu'il n'y a pas de système linéaire à résoudre et la direction d_k est à coup sûr une direction de descente. Ainsi la direction de descente est calculée comme suit:

$$d_k = -H_k \nabla f(x_k).$$

Comment choisir la matrice H_0? On peut éventuellement choisir I. Une alternative est d'utiliser $H_0 = I$ pour la première itération et ensuite mettre H_0 à jour avant de calculer H_1 en utilisant:

$$H_0 = \frac{y_k^T s_k}{y_k^T y_k} I.$$

Important: pour s'assurer que la matrice H_k reste définie positive à toutes les itérations, il faut s'assurer que $y_k^T s_k > 0$. C'est toujours vrai pour des fonctions convexes, mais pas nécessairement dans le cas général. On pourra tester ici la version "skip" qui ne mets pas à jour quand cette condition n'est pas vérifiée.

Figure 2. From left to right: Broyden, Fletcher, Goldfarb, and Shanno.

In [ ]:
```
"""
This function aims to compute the next H matrix in the BFGS method.
Inputs:, sk, yk and Hk
Outputs: H_{k+1}
"""
function nextH(sk, yk, Hk)
    pk = 1 ./(transpose(yk)*sk)
    return (I-pk.*sk*transpose(yk))*Hk*(I-pk.*yk*transpose(sk)) + pk.*sk*transpose(sk)

end


"""Here are some tests"""

a = ones((2,1))
b = ones((2,1))
b[1,1] = b[1,1]* 4
display(nextH(a,b,I))
```

```
2×2 Matrix{Float64}:
  0.28  -0.12
 -0.12   1.48
```

In [ ]:
```
# Solution: copier-coller votre newton_armijo ici et modifier le calcul de la direction avec la méthode de BFGS inverse
"""
BFGS algorithm implemented with armijo linesearch
inputs: nlp object, starting point x0
output: ending point xk, f(xk), search history

"""
function bfgs_quasi_newton_armijo(nlp, x0, verbose::Bool = true, epsilon_abs = 1.0e-6, epsilon_rel = 1.0e-6, max_iter::I
  start_time = time()
  historique = [x0]
  xk  = x0 # initialize xk at x0
  fk  = obj(nlp, xk) # evaluate the objective function at xk
  gk = grad(nlp, xk) # get gradient
  gnorm = gnorm0 = norm(gk) # get the norm of the gradient
  k = 0 # round 0
  Hk = I
  error = false

  verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
  verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
  while gnorm > epsilon_abs + epsilon_rel * gnorm0 && (time() - start_time) <= max_time  # while the stopping conditions
    dk = - Hk * gk
    slope = dot(dk, gk) # slope= direction@gradient

    t = armijo_mod(xk, dk, fk, gk, nlp)
    last_x = xk # save last xk
    last_g = gk # save last gk
    xk += t * dk # get new xk
    push!(historique, xk) # append the history
    fk = obj(nlp, xk) # evaluate the nlp
    gk =  grad(nlp, xk) # evaluate the gradient
    gnorm = norm(gk) # get the norm
    sk = xk - last_x
    yk = gk - last_g
    if k == 0 # update H0
      Hk = (transpose(yk)*sk ./ (transpose(yk)*yk)) *I
    end
    if transpose(yk)*sk > 0 # skip if not respected
      Hk = nextH(sk, yk, Hk) # find next Hk
    end

    k += 1

    # prints
    if fk <= lower_bound
```

```
        xk = -Inf64
        @printf "The problem is unbounded below. \n"
        error = true
        break
      elseif k > max_iter
        @printf "Maximal number of iterations has been reached \n"
        error = true
        break
      elseif (time() - start_time) >= max_time
        @printf "Timeout has been reached"
        error = true
        break

      elseif neval_obj(nlp) > max_eval
        @printf "Max number of evaluations has been reached \n"
        error = true
        break
      end
      verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
    end
    if error == false
      println("An optimal solution has been found in $(time() - start_time) seconds")
    else
      println("An error occured during solving")
    end

    return xk, obj(nlp, xk), historique
  end
```

```
bfgs_quasi_newton_armijo
```

In [ ]:

```
#Test
f(x) = x[1]^2 * (2*x[1] - 3) - 6*x[1]*x[2] * (x[1] - x[2] - 1)
x0 = zeros(2)
x0[1] = 1.5
x0[2] = 0.5
nlp = ADNLPModel(f, x0)

arg, star = bfgs_quasi_newton_armijo(nlp, x0)
print("argmin est $arg \n")
println("On trouve une fonction objectif de $star ")
```

```
 k        fk  ||∇f(x)||
 0  0.00e+00   4.5e+00
 1 -3.73e-01   4.2e+00 8.8e-02
 2 -7.22e-01   1.4e+00 1.0e+00
 3 -8.44e-01   9.3e-01 1.0e+00
 4 -9.83e-01   4.6e-01 1.0e+00
 5 -9.98e-01   2.3e-01 1.0e+00
 6 -1.00e+00   7.9e-02 1.0e+00
 7 -1.00e+00   1.3e-02 1.0e+00
 8 -1.00e+00   9.1e-04 1.0e+00
 9 -1.00e+00   1.3e-04 1.0e+00
10 -1.00e+00   6.7e-06 1.0e+00
11 -1.00e+00   1.1e-07 1.0e+00
An optimal solution has been found in 0.02729201316833496 seconds
argmin est [1.0000000149131603, 1.5213298824643154e-9]
On trouve une fonction objectif de -0.9999999999999994
```

Exercice 4: application à un problème de grande taille

On va ajouter le package `OptimizationProblems` qui contient, comme son nom l'indique, une collection de problème d'optimisation disponible au format de `JuMP` (dans le sous-module `OptimizationProblems.PureJuMP`) et de `ADNLPModel` (dans le sous-module `OptimizationProblems.ADNLPProblems`).

In [ ]:

```
using ADNLPModels, OptimizationProblems.ADNLPProblems, Random # Attention si vous ne faites pas using ADNLPModels avant
```

In [ ]:

```
n = 500
model = genrose(n=n)
@test typeof(model) <: ADNLPModel
```

```
Test Passed
```

Si vous le souhaitez, il est possible d'accéder à certaines informations sur le problème en accédant à son meta:

In [ ]:

```
using OptimizationProblems
OptimizationProblems.genrose_meta
```

```
Dict{Symbol, Any} with 17 entries:
  :has_equalities_only    => false
  :origin                 => :unknown
  :has_inequalities_only  => false
  :defined_everywhere     => missing
  :has_fixed_variables    => false
  :variable_ncon          => false
  :nvar                   => 100
  :is_feasible            => true
  :minimize               => true
  :ncon                   => 0
  :name                   => "genrose"
  :best_known_lower_bound => -Inf
  :objtype                => :other
  :best_known_upper_bound => 405.106
  :has_bounds             => false
  :variable_nvar          => true
  :contype                => :unconstrained
```

Résoudre le problème `genrose` et un autre problème de la collection en utilisant vos algorithmes précédents. Avant d'utiliser l'algorithme on testera que le problème est bien sans contrainte avec:

In [ ]:
```
# TODO
n = 200
nlp = genrose(n = n)
unconstrained(nlp) #qui retourne vrai si `nlp` est un problème sans contraintes.
v = Vector{Float64}(undef,n)
x0 = rand!(v,-20:20)

# Use previous functions to solve genrose.
println("Genrose (Newton Armijo):")
arg1, star1 = newton_armijo_v2(nlp, x0)
print("argmin est $arg1 \n")
println("On trouve une fonction objectif de $star1 ")


println("Genrose (LDLT Armijo):")
arg2, star2 = newton_ldlt_armijo(nlp, x0)
print("argmin est $arg2 \n")
println("On trouve une fonction objectif de $star2 ")

println("Genrose (Quasi Newton Armijo):")
arg3, star3 = bfgs_quasi_newton_armijo(nlp, x0 )
print("argmin est $arg3 \n")
println("On trouve une fonction objectif de $star3 ")
```

```
Genrose (Newton Armijo):
 k        fk  ||∇f(x)||
 0  6.50e+08   1.8e+07
 1  6.44e+08   1.9e+07 8.8e-02
 2  1.62e+08   7.9e+06 1.0e+00
 3  1.13e+08   5.5e+06 4.4e-01
 4  5.34e+07   6.2e+06 1.0e+00
 5  2.06e+07   3.0e+06 1.0e+00
 6  9.88e+06   1.6e+06 1.0e+00
 7  9.70e+06   1.6e+06 2.6e-02
 8  7.17e+06   1.2e+06 4.4e-01
 9  1.40e+06   3.5e+05 1.0e+00
10  1.34e+06   3.6e+05 8.8e-02
11  2.48e+05   9.6e+04 1.0e+00
12  2.37e+05   9.8e+04 5.9e-02
13  5.07e+04   2.7e+04 1.0e+00
14  4.93e+04   2.8e+04 3.9e-02
15  4.33e+04   2.5e+04 1.3e-01
16  3.67e+04   2.1e+04 3.0e-01
17  3.22e+04   1.7e+04 4.4e-01
18  3.07e+04   1.6e+04 8.8e-02
19  7.35e+03   4.6e+03 1.0e+00
20  5.63e+03   8.4e+03 4.4e-01
21  3.96e+03   5.0e+03 4.4e-01
22  1.72e+03   6.6e+02 1.0e+00
23  1.64e+03   1.8e+03 8.8e-02
24  1.62e+03   1.5e+03 2.0e-01
25  1.58e+03   1.3e+03 1.3e-01
26  1.57e+03   1.2e+03 3.0e-01
27  1.57e+03   1.2e+03 7.7e-03
28  1.55e+03   1.1e+03 5.9e-02
29  1.50e+03   1.4e+03 1.0e+00
30  9.71e+02   6.4e+02 1.0e+00
31  9.33e+02   5.8e+02 6.7e-01
32  7.46e+02   5.1e+02 1.0e+00
33  6.91e+02   3.0e+02 1.0e+00
34  6.49e+02   2.7e+02 4.4e-01
35  5.77e+02   1.9e+02 1.0e+00
36  5.17e+02   4.7e+01 1.0e+00
37  5.16e+02   6.2e+01 4.4e-01
38  5.11e+02   6.5e+01 6.7e-01
39  5.10e+02   6.1e+01 8.8e-02
40  5.04e+02   3.6e+01 1.0e+00
41  5.02e+02   3.9e+01 6.7e-01
42  5.00e+02   4.5e+01 6.7e-01
43  4.96e+02   2.9e+01 6.7e-01
44  4.94e+02   2.9e+01 3.0e-01
45  4.93e+02   2.6e+01 4.4e-01
46  4.88e+02   3.9e+01 1.0e+00
```

```
47  4.82e+02   1.4e+01 1.0e+00
An optimal solution has been found in 0.7732281684875488 seconds
argmin est [-0.9932828903652735, 0.9966447129638264, 0.9983176471304948, 0.9991424075191178, 0.9995344750737974, 0.999691
5912409828, 0.999693043138511, 0.9995395750081875, 0.9991537891420228, 0.998341295702879, 0.9966935269592335, 0.993385358
9409855, 0.9867742403175128, 0.9736337187585117, 0.9477725039645136, 0.8978670718416523, 0.8052958375168296, 0.6467349879
095946, 0.4144636408680006, 0.15735385640191468, 0.02241788417905135, 0.010483766565544948, 0.010214114524484192, 0.01020
8541223849833, 0.01020842625581891, 0.01020842388431261, 0.010208423835394338, 0.010208423834385277, 0.01020842383436446,
0.01020842383436403, 0.010208423834364022, 0.010208423834364022, 0.010208423834364022, 0.010208423834364022, 0.0102084238
34364022, 0.010208423834364022, 0.010208423834364022, 0.010208423834364018, 0.010208423834363878, 0.01020842383435707, 0.
010208423834026929, 0.010208423818022077, 0.010208423042123656, 0.010208385427382392, 0.010206561897378503, 0.01011814272
8581785, 0.005792891712573818, -0.3642696749441098, 0.6482592735388433, 0.8149992863292047, 0.9051565699514735, 0.9518388
125925493, 0.9753529156256319, 0.9867387899905489, 0.9915364813974434, 0.9921196848895323, 0.9887849905602163, 0.97990627
8253199, 0.9612416566504492, 0.9243176460326902, 0.8541468457334622, 0.7287561233172954, 0.5297252916609207, 0.2749877878
5419553, 0.06145307417669763, 0.011848842895445204, 0.010241060843305, 0.01013075288855628, 0.006463965111847972, -0.0776
7880733085769, 0.38428119959035273, 0.6607148651074665, 0.8212409869743548, 0.9070448138571334, 0.9502639964452216, 0.969
7805507247175, 0.9747160468167696, 0.9674782113709363, 0.9447964572112301, 0.8968782422836247, 0.8058374364206324, 0.6491
27058470808, 0.42244226423792985, 0.16789789020513363, 0.024182594973411603, 0.010527010993463894, 0.010215009947055704,
0.01020855969563916, 0.010208426636846561, 0.010208423892172264, 0.010208423835556462, 0.01020842383438862, 0.01020842383
4364532, 0.010208423834364032, 0.010208423834364022, 0.010208423834364022, 0.010208423834364022, 0.010208423834364018, 0.
010208423834363874, 0.010208423834356805, 0.010208423834014196, 0.010208423817404793, 0.010208423012198299, 0.01020838397
6632327, 0.010206491566012143, 0.010114731889820422, 0.005624564357668566, -0.38930928455835273, 0.6641170084748947, 0.82
39609026909461, 0.9100220090789392, 0.9545747276278639, 0.9771869407599411, 0.9885312064151673, 0.9941486321938584, 0.996
829102496526, 0.9979146253957285, 0.997950767157189, 0.9969562290305664, 0.9944333096880296, 0.9891268202394706, 0.978432
6542288637, 0.957250938108202, 0.9160645898206454, 0.8385900268956095, 0.7021517218246272, 0.4913545884876071, 0.23346340
880834465, 0.04354702189064238, 0.011105450649651944, 0.01022700768133619, 0.010206861017882998, 0.010114027580653318, 0.
005589424552341758, -0.39451941037721405, 0.6673041271933344, 0.8256434022659579, 0.9107616572826382, 0.9546993385096993,
0.9767756347736406, 0.9874071999565226, 0.9917792698234653, 0.9920575065017286, 0.988387651322436, 0.9789808402407363, 0.
9593527433353383, 0.9206372462993109, 0.8472948263323936, 0.7169764504884419, 0.5123004834292533, 0.2553634936069118, 0.0
5245121737784621, 0.01144712164488409, 0.01023420920193613, 0.010208955822951482, 0.010208434808007167, 0.010208424060722
981, 0.010208423839033243, 0.010208423834460337, 0.010208423834366006, 0.010208423834363932, 0.010208423834357626, 0.0102
08423834053864, 0.010208423819327872, 0.010208423105427297, 0.010208388496276723, 0.010206710675480258, 0.010125358475815
095, 0.006150188707456294, -0.2981023024301227, 0.5902570154773636, 0.7652441072162659, 0.8536188924252163, 0.88333199580
47661, 0.8660734774652394, 0.7975909928568344, 0.6626192319458905, 0.45283216417684474, 0.2098621870315052, 0.03916686834
0012366, -0.17357493081532083, 0.5296729076456302, 0.7486469633780386, 0.8702664395360791, 0.9341937923416064, 0.96689161
64483824, 0.9833764524165834, 0.9915953987113978, 0.9956044168567053, 0.9974044915009964, 0.9978989491876682, 0.997336723
6219282, 0.9954363329876105, 0.9912492542906182, 0.9827061183049111, 0.9656727090488683, 0.9322595911617111, 0.8683622327
848413, 0.7520878489698086, 0.560689066461377]
On trouve une fonction objectif de 482.1693130015966
Genrose (LDLT Armijo):
 k        fk ||∇f(x)||
 0  6.50e+08   1.8e+07
 1  6.16e+08   1.7e+07 5.9e-02
 2  5.84e+08   1.7e+07 1.3e-01
 3  1.16e+08   5.2e+06 1.0e+00
 4  8.03e+07   3.7e+06 4.4e-01
 5  1.21e+07   9.5e+05 1.0e+00
 6  1.16e+07   2.2e+06 1.0e+00
 7  4.59e+06   9.9e+05 1.0e+00
 8  4.56e+06   9.9e+05 7.7e-03
 9  9.06e+05   2.9e+05 1.0e+00
10  9.02e+05   3.1e+05 5.9e-02
11  1.69e+05   8.0e+04 1.0e+00
12  1.63e+05   8.1e+04 3.9e-02
13  4.82e+04   2.8e+04 1.0e+00
14  4.66e+04   3.2e+04 5.9e-02
15  4.32e+04   2.9e+04 8.8e-02
16  4.23e+04   3.2e+04 1.0e+00
17  4.03e+04   3.1e+04 3.9e-02
18  3.92e+04   3.0e+04 3.9e-02
19  8.70e+03   8.7e+03 1.0e+00
20  7.25e+03   8.0e+03 2.0e-01
21  6.98e+03   7.3e+03 8.8e-02
22  6.92e+03   7.1e+03 3.9e-02
23  6.85e+03   7.0e+03 7.7e-03
24  5.80e+03   5.8e+03 2.0e-01
25  4.45e+03   4.8e+03 1.0e+00
26  3.21e+03   4.7e+03 4.4e-01
27  3.00e+03   4.3e+03 8.8e-02
28  2.92e+03   4.0e+03 5.9e-02
29  2.91e+03   4.0e+03 7.7e-03
30  2.39e+03   3.3e+03 2.0e-01
31  2.20e+03   2.8e+03 2.0e-01
32  2.19e+03   2.6e+03 5.9e-02
33  1.08e+03   5.4e+02 1.0e+00
34  9.80e+02   1.4e+03 3.0e-01
35  7.48e+02   2.4e+02 1.0e+00
36  7.28e+02   2.6e+02 2.6e-02
37  6.79e+02   5.2e+02 2.0e-01
38  6.69e+02   4.9e+02 5.9e-02
39  6.17e+02   4.1e+02 4.4e-01
40  6.12e+02   3.9e+02 1.3e-01
41  5.68e+02   4.5e+02 4.4e-01
42  5.31e+02   4.2e+02 2.6e-02
43  5.26e+02   5.4e+02 6.7e-01
44  4.75e+02   4.4e+02 3.0e-01
45  4.16e+02   3.8e+02 1.0e+00
46  4.11e+02   3.6e+02 1.3e-01
47  3.68e+02   2.6e+02 4.4e-01
48  3.19e+02   1.8e+02 1.0e+00
49  3.14e+02   1.6e+02 8.8e-02
50  3.07e+02   1.4e+02 2.0e-01
51  2.93e+02   7.2e+01 1.0e+00
52  2.88e+02   5.1e+01 5.9e-02
53  2.88e+02   5.1e+01 2.6e-02
54  2.82e+02   2.7e+01 1.0e+00
55  2.81e+02   3.5e+01 1.2e-02
56  2.81e+02   4.6e+01 5.9e-02
```

```
57  2.81e+02   4.5e+01 1.7e-02
58  2.80e+02   4.7e+01 1.3e-01
59  2.79e+02   6.3e+01 6.7e-01
60  2.79e+02   7.0e+01 8.8e-02
61  2.79e+02   7.0e+01 6.8e-04
62  2.78e+02   6.9e+01 8.8e-02
63  2.77e+02   7.2e+01 2.6e-02
64  2.76e+02   7.0e+01 8.8e-02
65  2.75e+02   6.8e+01 1.3e-01
66  2.72e+02   7.7e+01 6.7e-01
67  2.68e+02   4.2e+01 3.4e-03
68  2.67e+02   4.1e+01 8.8e-02
69  2.64e+02   3.3e+01 1.0e+00
70  2.61e+02   4.7e+01 4.4e-01
71  2.59e+02   4.2e+01 2.6e-02
72  2.58e+02   3.3e+01 1.3e-01
73  2.57e+02   3.5e+01 2.0e-01
74  2.56e+02   3.2e+01 1.3e-01
75  2.56e+02   3.4e+01 2.3e-03
76  2.56e+02   3.0e+01 1.3e-01
77  2.54e+02   3.2e+01 2.0e-01
78  2.53e+02   3.9e+01 5.9e-02
79  2.53e+02   4.4e+01 1.7e-02
80  2.51e+02   4.9e+01 6.7e-01
81  2.51e+02   5.2e+01 1.5e-03
82  2.51e+02   5.2e+01 2.6e-02
83  2.49e+02   6.3e+01 4.4e-01
84  2.49e+02   6.1e+01 5.9e-02
85  2.48e+02   7.3e+01 3.0e-01
86  2.46e+02   7.1e+01 1.3e-01
87  2.44e+02   6.7e+01 5.9e-02
88  2.35e+02   1.1e+02 1.0e+00
89  2.30e+02   1.8e+02 7.7e-03
90  2.03e+02   1.7e+02 3.9e-02
91  1.79e+02   1.0e+02 6.7e-01
92  1.77e+02   8.9e+01 2.0e-01
93  1.75e+02   8.3e+01 5.9e-02
94  1.74e+02   8.1e+01 5.9e-02
95  1.68e+02   4.7e+01 6.7e-01
96  1.68e+02   4.9e+01 8.8e-02
97  1.67e+02   4.8e+01 1.2e-02
98  1.67e+02   4.8e+01 3.0e-01
```

```
99  1.66e+02   4.1e+01 1.3e-01
100  1.64e+02   4.7e+01 1.0e+00
argmin est [-0.993286085736001, 0.9966510445947183, 0.9983302597123248, 0.9991676178139778, 0.9995849614280562, 0.9997927
948339016, 0.9998960107452057, 0.9999467175027879, 0.999970518088447, 0.9999794306890849, 0.9999779553586345, 0.999965344
704306, 0.9999352211072035, 0.9998723350249551, 0.9997447796526415, 0.9994875393873922, 0.9989683007392863, 0.99791523038
78897, 0.9957584054074409, 0.9912600210948799, 0.9815863725030773, 0.9597053872912419, 0.9068518324385488, 0.779139372133
6481, 0.511618878865695, 0.17138301906399223, 0.022805473283432402, 0.010501487028260958, 0.010214485669452178, 0.0102085
47973828756, 0.010208382367087685, 0.010206288635304553, 0.01010510575779217, 0.005664162526188954, -0.00787984947725325
9, 0.13500487971788327, 0.5323755366678744, 0.7519047265136001, 0.8712709377472251, 0.9346744384079455, 0.967154386495188
5, 0.983556967412389, 0.991787156836633, 0.9959025775862699, 0.997956937781878, 0.9989815733112496, 0.9994924055861844, 0
.9997470274246242, 0.9998739290937939, 0.999937172649613, 0.9999686903043469, 0.9999843970373299, 0.9999922243920173, 0.9
999961250939415, 0.9999980689755747, 0.9999990376916784, 0.9999995204425279, 0.9999997610169654, 0.9999998809049981, 0.99
99999406500619, 0.9999999704234085, 0.9999999852605769, 0.9999999926543366, 0.9999999963385363, 0.999999998173697, 0.9999
999990865746, 0.9999999995381723, 0.9999999997565472, 0.9999999998519783, 0.9999999998726585, 0.9999999998290312, 0.99999
99996990647, 0.9999999994171258, 0.9999999988408356, 0.9999999976791671, 0.999999995345478, 0.9999999906612549, 0.9999999
812609643, 0.9999999623974555, 0.9999999245446406, 0.9999998485867844, 0.9999996961649122, 0.9999993903049442, 0.99999877
65434298, 0.9999975449140265, 0.9999950733763289, 0.9999901135317809, 0.9999801595636413, 0.9999601803085143, 0.999920068
4927367, 0.9998394962229572, 0.9996774872657355, 0.9993510734856803, 0.9986907946845842, 0.9973447939589731, 0.9945607365
638193, 0.9886527006831685, 0.9755821611224202, 0.9446561082015994, 0.8677415900216361, 0.6861019174886764, 0.35901370472
576133, 0.07649574407670662, 0.012770166367628634, 0.010262547843878728, 0.010209540841428739, 0.010208446875679192, 0.01
0208424309649104, 0.010208423844168134, 0.01020842383457415, 0.010208423834750976, 0.010208423852920985, 0.01020842473398
1862, 0.010208467446888294, 0.010210538121024256, 0.010310916137236177, 0.015162861122563667, 0.21961468851794938, 0.4994
109640827561, 0.7344855948406854, 0.8652894713170908, 0.9320068533999769, 0.9656463924926566, 0.9827835966963744, 0.99139
96072423671, 0.9957090285246165, 0.9978604097534881, 0.9989334486705306, 0.9994684179712536, 0.9997350720804039, 0.999867
9709149026, 0.9999342033582698, 0.999967210563583, 0.9999836596168048, 0.9999918569040086, 0.9999959419594029, 0.99999797
77121601, 0.9999989922115116, 0.9999994977781613, 0.9999997497229031, 0.9999998752777406, 0.9999999378478448, 0.999999969
0311123, 0.999999984575092, 0.9999999923294932, 0.9999999962102885, 0.9999999981772795, 0.9999999992237967, 0.99999999987
83311, 1.0000000004714225, 1.000000001302582, 1.0000000027915439, 1.0000000056902298, 1.0000000114624592, 1.0000000230231
387, 1.0000000462101333, 1.000000092731756, 1.000000186076918, 1.0000003733667748, 1.000000749109609, 1.0000015027612066,
1.0000030137302358, 1.0000060402816784, 1.0000120915893613, 1.000024146031022, 1.0000479781959875, 1.0000943544388388, 1.
0001814505457052, 1.0003309402599527, 1.000541475870837, 1.0007850528561417, 1.0010469199499963, 1.0014403841718285, 1.00
09759961298057, 1.0005321465282906, 1.000275037610075, 1.000139296443137, 1.0000699451544286, 1.0000349840877338, 1.00001
74652441929, 1.000008711345854, 1.0000043431183379, 1.0000021648111057, 1.0000010789022742, 1.0000005376331684, 1.0000002
678180087, 1.0000001332386406, 1.0000000659420396, 1.000000031945785, 1.0000000140823118, 1.0000000033305305, 0.999999994
2600938, 0.9999999822874827, 0.9999999613546257, 0.9999999208424982, 0.9999998400995326, 0.9999996775741212, 0.9999993480
685576, 0.999998672639764, 0.9999972599093936]
On trouve une fonction objectif de 163.88092453875953
Genrose (Quasi Newton Armijo):
 k        fk ||∇f(x)||
 0  6.50e+08   1.8e+07
 1  1.23e+08   7.0e+06 1.2e-05
 2  7.40e+06   5.9e+05 1.0e+00
 3  4.70e+06   4.1e+05 1.0e+00
 4  1.56e+06   1.7e+05 1.0e+00
 5  6.92e+05   8.8e+04 1.0e+00
 6  3.08e+05   4.8e+04 1.0e+00
 7  1.60e+05   3.1e+04 1.0e+00
 8  9.36e+04   2.2e+04 1.0e+00
 9  6.36e+04   1.7e+04 1.0e+00
10  5.22e+04   1.5e+04 1.0e+00
11  4.85e+04   1.4e+04 1.0e+00
12  4.28e+04   1.3e+04 1.0e+00
13  3.02e+04   9.9e+03 1.0e+00
14  1.64e+04   6.1e+03 1.0e+00
15  9.17e+03   3.6e+03 1.0e+00
16  6.89e+03   2.9e+03 1.0e+00
17  6.37e+03   3.1e+03 1.0e+00
18  6.20e+03   3.2e+03 1.0e+00
19  5.96e+03   3.2e+03 1.0e+00
20  5.49e+03   2.9e+03 1.0e+00
21  4.81e+03   2.3e+03 1.0e+00
22  4.19e+03   1.9e+03 1.0e+00
23  3.75e+03   1.9e+03 1.0e+00
24  3.55e+03   2.0e+03 1.0e+00
25  3.47e+03   2.0e+03 1.0e+00
26  3.39e+03   2.0e+03 1.0e+00
27  3.21e+03   1.8e+03 1.0e+00
28  2.91e+03   1.5e+03 1.0e+00
29  2.57e+03   9.6e+02 1.0e+00
30  2.37e+03   8.7e+02 1.0e+00
31  2.22e+03   8.4e+02 1.0e+00
32  2.15e+03   8.2e+02 1.0e+00
33  2.11e+03   8.1e+02 1.0e+00
34  2.09e+03   8.1e+02 1.0e+00
35  2.08e+03   8.1e+02 1.0e+00
36  2.05e+03   8.0e+02 1.0e+00
37  2.01e+03   7.8e+02 1.0e+00
38  1.91e+03   7.2e+02 1.0e+00
39  1.74e+03   6.2e+02 1.0e+00
40  1.55e+03   5.4e+02 1.0e+00
41  1.44e+03   5.1e+02 1.0e+00
42  1.41e+03   4.9e+02 1.0e+00
43  1.40e+03   4.7e+02 1.0e+00
44  1.39e+03   4.6e+02 1.0e+00
45  1.39e+03   4.4e+02 1.0e+00
46  1.37e+03   4.3e+02 1.0e+00
47  1.34e+03   4.2e+02 1.0e+00
48  1.29e+03   4.3e+02 1.0e+00
49  1.19e+03   4.5e+02 1.0e+00
50  1.13e+03   4.5e+02 1.0e+00
51  1.11e+03   4.3e+02 1.0e+00
52  1.10e+03   4.0e+02 1.0e+00
53  1.09e+03   4.0e+02 1.0e+00
54  1.09e+03   4.0e+02 1.0e+00
55  1.08e+03   4.0e+02 1.0e+00
56  1.06e+03   4.1e+02 1.0e+00
57  1.02e+03   4.3e+02 1.0e+00
58  9.48e+02   4.1e+02 1.0e+00
59  8.88e+02   3.5e+02 1.0e+00
60  8.58e+02   3.1e+02 1.0e+00
```

```
61  8.51e+02   3.0e+02 1.0e+00
62  8.49e+02   2.9e+02 1.0e+00
63  8.47e+02   2.9e+02 1.0e+00
64  8.42e+02   2.7e+02 1.0e+00
65  8.28e+02   2.6e+02 1.0e+00
66  7.98e+02   2.4e+02 1.0e+00
67  7.44e+02   2.5e+02 1.0e+00
68  6.94e+02   2.9e+02 1.0e+00
69  6.71e+02   2.8e+02 1.0e+00
70  6.62e+02   2.6e+02 1.0e+00
71  6.61e+02   2.4e+02 1.0e+00
72  6.60e+02   2.4e+02 1.0e+00
73  6.57e+02   2.3e+02 1.0e+00
74  6.52e+02   2.2e+02 1.0e+00
75  6.39e+02   2.2e+02 1.0e+00
76  6.15e+02   2.1e+02 1.0e+00
77  5.80e+02   2.1e+02 1.0e+00
78  5.50e+02   2.0e+02 1.0e+00
79  5.36e+02   1.8e+02 1.0e+00
80  5.34e+02   1.8e+02 1.0e+00
81  5.34e+02   1.8e+02 1.0e+00
82  5.32e+02   1.8e+02 1.0e+00
83  5.30e+02   1.8e+02 1.0e+00
84  5.25e+02   2.0e+02 1.0e+00
85  5.13e+02   2.3e+02 1.0e+00
86  4.81e+02   2.9e+02 1.0e+00
87  4.24e+02   3.0e+02 1.0e+00
88  4.01e+02   3.0e+02 1.0e+00
89  3.85e+02   2.6e+02 1.0e+00
90  3.78e+02   2.4e+02 1.0e+00
91  3.74e+02   2.1e+02 1.0e+00
92  3.70e+02   1.9e+02 1.0e+00
93  3.68e+02   1.8e+02 1.0e+00
94  3.67e+02   1.9e+02 1.0e+00
95  3.66e+02   1.9e+02 1.0e+00
96  3.64e+02   2.0e+02 1.0e+00
97  3.59e+02   2.1e+02 1.0e+00
98  3.47e+02   2.3e+02 1.0e+00
99  3.23e+02   2.5e+02 1.0e+00
100  2.87e+02   2.3e+02 1.0e+00
Maximal number of iterations has been reached
An error occured during solving
argmin est [0.5236939701726346, -0.01436222079369176, 0.07312833139375927, -0.033474325734938214, 0.0055197922815156605,
-0.02155150870196297, -0.0030760233930257354, 0.014322603716315564, 0.028524910161210844, 0.033596747273575533, -0.023527
84190412148, 0.041122456866105994, -0.010700912906135776, 0.021033239684342266, 0.036003767144578716, -0.0154040796720123
11, 0.02088753968550955, -0.0029875078709729885, 0.010092039703664265, 0.018306763104473732, 0.03142655649045304, -0.0608
18131893858485, -0.11421845099023947, -0.14411736019349028, -0.11742058061608379, -0.04167908758546507, -0.08140499072771
93, 0.056568269065529406, -0.012481536635450806, 0.010425962903166879, -0.016113445311476814, -0.0131267620393795, 0.0314
4917565131148, 0.0437287473367633, 0.04096343547508383, -0.01940169640290422, 0.03322785921893892, -0.021446474165504523,
-0.018561083309686064, -0.015765022881064322, -0.023064837998345543, -0.0014476123131031728, 0.0021877747860682105, 0.031
307314964612906, -0.01921719620379666, 0.034546422565065085, 0.03799244378002553, 0.03712352703287802, 0.0416079041788326
3, 0.0628288629891779, 0.03672622706671087, 0.030937937287643925, 0.0064861298456851246, -0.008295849153456183, -0.003154
6135707123547, 0.02657224579830546, 0.02525776354465844, 0.04646284953292143, -0.019277392848382976, 0.035748579198004726
, 0.0025166815054470364, 0.055923808929540325, 0.06497661210841103, 0.03535238848293404, 0.02294923247930586, -0.01242649
1644560063, -0.028849718172229977, 0.07047641808224339, 0.029028386909212875, -0.007066992783092998, 0.04325000891449001,
-0.020687139296779344, 0.003143482368605101, 0.037855669828951805, -0.03332157494438579, 0.018759793735022817, 0.02281088
879361188, 0.014834826301503068, 0.028076454109207317, -0.005915612612900273, -0.017624900340688684, 0.029857995968614257
, 0.031003802168703607, -0.03558294494554824, 0.0787087892606856, 0.025836982166779876, 0.032467996348032976, -0.00627843
6603626508, -0.0236472171460781, -0.0279305763990126, 0.02944286798887267, 0.0021437767656090 76, 0.0355845547995519, 0.03
578215112040978, -0.013929054579586443, -0.021478457204183385, 0.03473057781538075, -0.006651096192239065, -0.03165633740
272327, 0.04041765754457672, -0.021433074239145398, 0.00841000615527171, -0.019350682354617586, 0.02207486480028748, -0.0
21924065044848885, -0.022483402696436516, 0.04301463106951778, -0.012452760512180226, 0.041045728037161326, 0.02642419678
4459272, -0.010591510278218589, -0.023937602446413176, 0.03117844275418685, -0.008257868115719168, 0.03715472811231103, 0
.025175223006507638, -0.031858937858807945, 0.022193305241876675, 0.06691849935490685, 0.0007778991064981344, -0.03032872
39631884, 0.02746162946756823, -0.03045544256776727, 0.027627425570324257, 0.017527587243943106, -0.0006178083654129305,
0.03641720732420116, -0.006607147322013871, -0.02068582774054335, 0.044111171555888735, 0.03751131170432693, -0.010978521
283924202, 0.015110771725494982, -0.028051413019027957, 0.014150239359177079, 0.020710294486423243, -0.05386477888164653
6, -0.016311335682928765, -0.020863551598693617, -0.02515654749180911, 0.056028280049164186, 0.019156071722561063, 0.0146
7795417180183, 0.0370790908103934, 0.011880820053362901, 0.02043686339371715, 0.01650668744657263, 0.0015950940769924177,
0.04349986237161314, 0.007000163000974687, 0.029568290092365085, 0.012759230463231002, 0.016903102535202892, -0.026240245
173966305, 0.002774245768101621, 0.04554015349697596, -0.0025631853735968435, 0.015851992775731745, -0.02473222401994288
5, -0.023229060553789406, -0.005269697738773292, -0.009624066910075703, 0.042386461386766834, 0.04046994486176943, 0.0428
37376258499094, -0.0020738805987672183, -0.03494355937098011, -0.06355600547658725, 0.033951562717554534, 0.0282013520226
83094, 0.03066867982860857, -0.023324286139380892, 0.031840508438215065, -0.017640077440719876, 0.005912422011160549, -0.0
27271591426143618, -0.009898459349287262, -0.022545973223792315, 0.03170226541250874, -0.0014359847648903368, -0.01646712
844194851, -0.020632659507554444, 0.0386280271853185, 0.009442715264158743, 0.03341401257947455, -0.005357120724571795,
-0.01113740836292237, 0.05755554509404026, 0.030859882514869028, 0.023897357642653373, 0.008947803137285423, 0.0571345620
2679151, -0.020522202458026585, 0.026447279302087845, -0.012292536848771502, -0.016900828520707734, 0.002163310996199014,
0.07457406204697822, -0.24221668041222444, -0.04782478728481945]
On trouve une fonction objectif de 236.5491639526762
```

In [ ]:

```julia
n = 100
nlp = tridia(n = n)
unconstrained(nlp) #qui retourne vrai si `nlp` est un problème sans contraintes.
v = Vector{Float64}(undef,n)
x0 = rand!(v,-3:3)

# Use previous functions to solve genrose.
println("Tridia (Newton Armijo):")
arg1, star1 = newton_armijo_v2(nlp, x0)
print("argmin est $arg1 \n")
println("On trouve une fonction objectif de $star1 \n \n")


println("Tridia (LDLT Armijo):")
arg2, star2 = newton_ldlt_armijo(nlp, x0)
print("argmin est $arg2 \n")
println("On trouve une fonction objectif de $star2  \n \n")

println("Tridia (Quasi Newton Armijo):")
```

```
arg3, star3 = bfgs_quasi_newton_armijo(nlp, x0)
print("argmin est $arg3 \n")
println("On trouve une fonction objectif de $star3  \n \n")
```

```
Tridia (Newton Armijo):
 k        fk ||∇f(x)||
 0  1.19e+05   1.5e+04
 1  2.28e-27   1.7e-12 1.0e+00
An optimal solution has been found in 0.003070831298828125 seconds
argmin est [1.0000000000000004, 0.5000000000000004, 0.25, 0.12499999999999956, 0.0625, 0.03125000000000022, 0.01562499999
9999988, 0.0078125, 0.003906249999998426, 0.001953124999999556, 0.0009765624999995559, 0.0004882812500004441, 0.00024414
0625, 0.0001220703125003632, 6.103515625021111e-5, 3.051757812546735e-5, 1.525878906338818e-5, 7.629394531734198e-6, 3.8
14697266069089e-6, 1.9073486333267193e-6, 9.536743172944284e-7, 4.768371590913034e-7, 2.384185795456517e-7, 1.19209289994
87046e-7, 5.960464533393435e-8, 2.980232327587373e-8, 1.4901162082026076e-8, 7.450581041013038e-9, 3.725290742551124e-9,
1.8626456183577465e-9, 9.313227966600834e-10, 4.656615093523442e-10, 2.3283019956465978e-10, 1.1641487773772496e-10, 5.82
0766091346741e-11, 2.9104274545943554e-11, 1.4551582161459464e-11, 7.275735569578501e-12, 3.637756762486788e-12, 1.818989
4035458565e-12, 9.099387909827783e-13, 4.550025315038751e-13, 2.276129836953263e-13, 1.141309269314661e-13, 5.70654634657
3305e-14, 2.886579864025407e-14, 1.4654943925052066e-14, 7.549516567451064e-15, 3.552713678800501e-15, 1.3322676295501878
e-15, 6.661338147750939e-16, 4.440892098500626e-16, 2.220446049250313e-16, 0.0, -4.440892098500626e-16, -1.11022302462515
65e-16, 1.1102230246251565e-16, 1.1102230246251565e-16, 4.440892098500626e-16, 1.1102230246251565e-16, 0.0, 0.0, 0.0, 0.0
, -4.440892098500626e-16, 0.0, -2.220446049250313e-16, -4.440892098500626e-16, 4.440892098500626e-16, -2.5100728968820273
e-16, -8.881784197001252e-16, -5.019465355918274e-16, -4.440892098500626e-16, -2.220446049250313e-16, 2.220446049250313
e-16, 8.881784197001252e-16, 4.440892098500626e-16, 4.440892098500626e-16, 1.1102230246251565e-16, 2.220446049250313e-16,
4.440892098500626e-16, -4.440892098500626e-16, 4.440892098500626e-16, 8.881784197001252e-16, 4.440892098500626e-16, 4.440
892098500626e-16, 4.440892098500626e-16, 4.440892098500626e-16, 8.881784197001252e-16, 4.440892098500626e-16, 2.220446049
250313e-16, 3.3306690738754696e-16, 4.440892098500626e-16, 0.0, 4.440892098500626e-16, 4.440892098500626e-16, 6.661338147
750939e-16, 4.440892098500626e-16, 4.440892098500626e-16, 0.0]
On trouve une fonction objectif de 2.2843398701057977e-27


Tridia (LDLT Armijo):
 k        fk ||∇f(x)||
 0  1.19e+05   1.5e+04
 1  1.96e-27   1.2e-12 1.0e+00
argmin est [1.0000000000000004, 0.5000000000000004, 0.25000000000000044, 0.12500000000000044, 0.06250000000000044, 0.0312
5000000000022, 0.01562500000000019, 0.007812500000000222, 0.003906250000000294, 0.001953125000000222, 0.00097656250000044
4, 0.00048828125, 0.000244140625, 0.0001220703125000496, 6.103515624996666e-5, 3.0517578124874307e-5, 1.52587890625e-5,
7.6293945312354065e-6, 3.814697265180911e-6, 1.9073486326156932e-6, 9.5367431640625e-7, 4.768371577590358e-7, 2.384185786
574733e-7, 1.1920928955078125e-7, 5.96046451434426e-8, 2.980232283178452e-8, 1.4901160749758446e-8, 7.450580152834618e-9,
3.725290742551124e-9, 1.8626451331575591e-9, 9.313223525708736e-10, 4.6566106526313433e-10, 2.3283064365386963e-10, 1.164
1576591614466e-10, 5.820766091346741e-11, 2.9103830456733704e-11, 1.4551582161459464e-11, 7.275735569578501e-12, 3.637090
628672013e-12, 1.8181012251261564e-12, 9.090506125630782e-13, 4.54785195495902e-13, 2.2760161090946e-13, 1.14130926931466
1e-13, 5.750955267558311e-14, 2.864375403532904e-14, 1.4210854715202004e-14, 6.661338147750939e-15, 3.1086244689504383e-1
5, 1.7763568394002505e-15, 4.440892098500626e-16, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 4.440892098500626e-16, 5.6
59162945429592e-16, 1.3322676295501878e-15, 1.3322676295501878e-15, 4.440892098500626e-16, 4.440892098500626e-16, 4.44089
2098500626e-16, 4.440892098500626e-16, 8.881784197001252e-16, 5.582993569023956e-16, 4.440892098500626e-16, 3.34616331801
2322e-16, 2.220446049250313e-16, 3.3306690738754696e-16, 4.440892098500626e-16, 8.881784197001252e-16, 8.881784197001252
e-16, 4.440892098500626e-16, 4.440892098500626e-16, 6.661338147750939e-16, 8.881784197001252e-16, 8.881784197001252e-16,
8.881784197001252e-16, 4.440892098500626e-16, 2.220446049250313e-16, 2.220446049250313e-16, 4.440892098500626e-16, 4.4408
92098500626e-16, 4.440892098500626e-16, 0.0, 1.1102230246251565e-16, 1.1102230246251565e-16, 2.220446049250313e-16, 0.0,
0.0, 0.0, 2.220446049250313e-16, 0.0, -2.220446049250313e-16, -4.440892098500626e-16]
On trouve une fonction objectif de 1.9645811831380393e-27


Tridia (Quasi Newton Armijo):
 k        fk ||∇f(x)||
 0  1.19e+05   1.5e+04
 1  7.04e+04   1.3e+04 1.5e-03
 2  1.02e+04   2.5e+03 1.0e+00
 3  5.55e+03   1.6e+03 1.0e+00
 4  1.45e+03   9.0e+02 1.0e+00
 5  8.58e+02   6.2e+02 1.0e+00
 6  4.27e+02   3.5e+02 1.0e+00
 7  3.02e+02   2.8e+02 1.0e+00
 8  2.09e+02   2.1e+02 1.0e+00
 9  1.59e+02   1.7e+02 1.0e+00
10  1.22e+02   1.4e+02 1.0e+00
11  1.00e+02   1.1e+02 1.0e+00
12  8.53e+01   9.0e+01 1.0e+00
13  7.53e+01   7.8e+01 1.0e+00
14  6.74e+01   7.2e+01 1.0e+00
15  6.06e+01   6.8e+01 1.0e+00
16  5.42e+01   6.5e+01 1.0e+00
17  4.84e+01   6.3e+01 1.0e+00
18  4.28e+01   6.4e+01 1.0e+00
19  3.68e+01   6.7e+01 1.0e+00
20  3.00e+01   7.2e+01 1.0e+00
21  2.27e+01   7.1e+01 1.0e+00
22  1.63e+01   6.1e+01 1.0e+00
23  1.18e+01   5.0e+01 1.0e+00
24  8.64e+00   4.3e+01 1.0e+00
25  6.30e+00   3.8e+01 1.0e+00
26  4.54e+00   3.1e+01 1.0e+00
27  3.40e+00   2.5e+01 1.0e+00
28  2.70e+00   2.0e+01 1.0e+00
29  2.22e+00   1.7e+01 1.0e+00
30  1.86e+00   1.5e+01 1.0e+00
31  1.57e+00   1.4e+01 1.0e+00
32  1.31e+00   1.4e+01 1.0e+00
33  1.04e+00   1.4e+01 1.0e+00
34  7.69e-01   1.3e+01 1.0e+00
35  5.42e-01   1.2e+01 1.0e+00
36  3.74e-01   9.8e+00 1.0e+00
37  2.56e-01   8.2e+00 1.0e+00
38  1.74e-01   6.7e+00 1.0e+00
39  1.22e-01   5.2e+00 1.0e+00
40  9.19e-02   3.9e+00 1.0e+00
41  7.32e-02   3.3e+00 1.0e+00
42  5.93e-02   2.9e+00 1.0e+00
43  4.89e-02   2.4e+00 1.0e+00
44  4.19e-02   2.0e+00 1.0e+00
45  3.71e-02   1.7e+00 1.0e+00
46  3.33e-02   1.6e+00 1.0e+00
47  3.01e-02   1.4e+00 1.0e+00
```

```
48  2.76e-02   1.2e+00 1.0e+00
49  2.59e-02   1.0e+00 1.0e+00
50  2.46e-02   9.1e-01 1.0e+00
51  2.35e-02   8.5e-01 1.0e+00
52  2.25e-02   7.8e-01 1.0e+00
53  2.17e-02   7.2e-01 1.0e+00
54  2.10e-02   7.0e-01 1.0e+00
55  2.03e-02   7.1e-01 1.0e+00
56  1.96e-02   7.0e-01 1.0e+00
57  1.90e-02   6.5e-01 1.0e+00
58  1.84e-02   6.1e-01 1.0e+00
59  1.79e-02   6.0e-01 1.0e+00
60  1.74e-02   6.0e-01 1.0e+00
61  1.69e-02   5.9e-01 1.0e+00
62  1.64e-02   5.8e-01 1.0e+00
63  1.59e-02   6.0e-01 1.0e+00
64  1.53e-02   6.6e-01 1.0e+00
65  1.47e-02   7.0e-01 1.0e+00
66  1.40e-02   6.9e-01 1.0e+00
67  1.33e-02   6.6e-01 1.0e+00
68  1.27e-02   6.8e-01 1.0e+00
69  1.20e-02   7.2e-01 1.0e+00
70  1.13e-02   7.5e-01 1.0e+00
71  1.04e-02   7.8e-01 1.0e+00
72  9.52e-03   8.3e-01 1.0e+00
73  8.45e-03   9.1e-01 1.0e+00
74  7.24e-03   9.4e-01 1.0e+00
75  6.03e-03   9.0e-01 1.0e+00
76  4.94e-03   8.5e-01 1.0e+00
77  3.93e-03   8.3e-01 1.0e+00
78  2.96e-03   8.1e-01 1.0e+00
79  2.12e-03   7.0e-01 1.0e+00
80  1.55e-03   5.5e-01 1.0e+00
81  1.19e-03   4.6e-01 1.0e+00
82  9.06e-04   4.3e-01 1.0e+00
83  6.62e-04   3.9e-01 1.0e+00
84  4.72e-04   3.3e-01 1.0e+00
85  3.39e-04   2.8e-01 1.0e+00
86  2.43e-04   2.4e-01 1.0e+00
87  1.70e-04   2.1e-01 1.0e+00
88  1.20e-04   1.7e-01 1.0e+00
89  8.57e-05   1.4e-01 1.0e+00
90  6.11e-05   1.2e-01 1.0e+00
91  4.26e-05   1.0e-01 1.0e+00
92  2.99e-05   8.3e-02 1.0e+00
93  2.14e-05   7.2e-02 1.0e+00
94  1.46e-05   6.6e-02 1.0e+00
95  8.86e-06   5.8e-02 1.0e+00
96  4.94e-06   4.4e-02 1.0e+00
97  2.83e-06   3.1e-02 1.0e+00
98  1.75e-06   2.3e-02 1.0e+00
99  1.12e-06   1.8e-02 1.0e+00
100  7.43e-07   1.4e-02 1.0e+00
An optimal solution has been found in 0.016509056091308594 seconds
argmin est [0.9999929941052751, 0.4999846166199124, 0.25003878885361003, 0.12497640848676961, 0.062459777782592825, 0.031
247426922738894, 0.015655763110695487, 0.007849573226338273, 0.003953572719658264, 0.001962588980726668, 0.00094858559349
94696, 0.0004549662327795867, 0.0002139154007133493, 7.578201954056958e-5, 3.0949811547299093e-5, 1.5849853530103356e-5,
3.1311634986154932e-6, -3.6440140959765144e-6, -2.04719456295014e-5, -2.8164523874444137e-5, -3.903774352972876e-5, -4.20
8761275831031e-5, -4.37449436782306e-5, -3.2361607299786276e-5, -3.1734180224066476e-5, -1.4580248323853167e-5, -1.268739
0007131413e-5, 2.6948079278496868e-6, 1.5369545713598077e-6, 1.1030096601693634e-5, 8.000394381798769e-7, 3.4764828084031
995e-6, 8.361803915164032e-6, 9.407399791962483e-6, 2.8042863703542773e-6, -2.5499164433661854e-6, -1.5837660993690581
e-6, -2.8144471811607987e-6, -5.329766302171853e-6, 3.4969753693920293e-6, 5.852725834971847e-6, 2.28382607823165e-6, 8.5
0582243417623e-6, 6.880446833560525e-6, -3.0017085084869816e-6, -1.934447645037781e-6, 3.6210345715094457e-6, 5.749048715
081301e-6, 8.808557915926936e-6, 1.21748447990043e-5, 8.612691528256538e-6, 4.056495081241347e-6, 5.984720792851214e-6,
6.51863544698516e-6, 5.900548460964439e-6, 8.101567275428789e-6, 1.2056642151193256e-5, 1.070565002345469e-5, 6.322093060
041326e-6, 6.225220250805928e-6, 9.68823231975542e-6, 1.1987367769060233e-5, 9.808803981184992e-6, 4.334076774386059e-6,
2.7799570853124295e-6, 5.382234074493404e-6, 5.087112191017313e-6, 1.3381610315150703e-6, -2.7284227699505767e-7, 1.84942
9931813794e-6, 2.8449917608140012e-6, 2.829414098741117e-6, 3.1821154128061064e-6, 3.1518491171182477e-6, 5.8518962402102
09e-7, -2.3636942119481918e-6, -2.8113960835237384e-6, -9.996100269248849e-7, -3.3137342370339475e-7, -1.4254701682506933
e-6, -2.2654483182819354e-6, -1.4937960505819134e-6, 1.1859720532677508e-6, 3.213178909884004e-6, 2.737211374296495e-6,
2.5208316892827063e-7, -1.8691203454773425e-6, -2.3499522318754615e-6, -1.4956183433883668e-6, -2.5005273736221306e-6,
-3.8889730142741194e-6, -3.7163168469217343e-6, -6.802474772399441e-7, 2.0976185543479207e-6, 2.638489670571905e-6, -5.24
9473065797279e-7, -4.6294799672551705e-6, -6.880018193656062e-6, -6.843900122533843e-6, -5.1124551827689094e-6]
On trouve une fonction objectif de 7.430884403012082e-7
```

Devoir

Exercice 1

En partant de votre fonction BFGS codé pendant le lab (à finir), écrire une méthode adapté au cas quadratique convexe

$$f(x) = 0.5x^\top Ax - b^\top x$$

Réponse:

On reprend la fonction BFGS codé en lab et on modifie le pas de la recherche linéaire pour satisfaire l'expression

$$\alpha = -\frac{\nabla f(x)^\top d}{d^\top Ad}$$

Dans la fonction `bfgs_quadratic_versionJulien` , on utilise le fait que la fonction soit quadratique, donc que son hessien est exactement la matrice A. On calcule une première fois le Hessien au début de la fonction pour obtenir A et on l'utilise par après pour obtenir la direction et obtenir le α optimal.

Exercice 1.1

In [ ]:
```
function bfgs_quadratic(nlp, x0, verbose::Bool = true, epsilon_abs = 1.0e-6, epsilon_rel = 1.0e-6, max_iter::Int = 100,
    start_time = time()
    xk  = x0 # initialize xk at x0
    fk  = obj(nlp, xk) # evaluate the objective function at xk
    gk = grad(nlp, xk) # get gradient
    gnorm = gnorm0 = norm(gk) # get the norm of the gradient
    A = hess(nlp, xk)

    k = 0 # round 0
    Hk = I
    error = false

    verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
    verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
    while gnorm > epsilon_abs + epsilon_rel * gnorm0 && (time() - start_time) <= max_time  # while the stopping conditio
      dk = - Hk * gk
      slope = dot(dk, gk) # slope= direction@gradient

      alpha = - transpose(gk) * dk ./ (transpose(dk)*A*dk)
      last_x = xk
      last_g = gk
      xk += alpha * dk
      fk = obj(nlp, xk)
      gk =  grad(nlp, xk)
      gnorm = norm(gk)


      sk = xk - last_x
      yk = gk - last_g
      if k == 0 # update H0
        Hk = (transpose(yk)*sk ./ (transpose(yk)*yk)) *I
      end
      if transpose(yk)*sk > 0 # skip if not respected
        Hk = nextH(sk, yk, Hk) # find next Hk
      end

      k += 1


      # prints
      if fk <= lower_bound
        xk = -Inf64
        @printf "The problem is unbounded below. \n"
        error = true
        break
      elseif k > max_iter
        @printf "Maximal number of iterations has been reached \n"
        error = true
        break
      elseif (time() - start_time) >= max_time
        @printf "Timeout has been reached"
        error = true
        break

      elseif neval_obj(nlp) > max_eval
        @printf "Max number of evaluations has been reached \n"
        error = true
        break
      end
      verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
    end
    if error == false
      println("An optimal solution has been found in $(time() - start_time) seconds")
    else
      println("An error occured during solving")
    end
    return xk, obj(nlp, xk), Hk
  end
```

```
bfgs_quadratic (generic function with 18 methods)
```

In [ ]:
```
function bfgs_quadratic_versionJulien(nlp, x0, A_inv, verbose::Bool = true, epsilon_abs = 1.0e-6, epsilon_rel = 1.0e-6,
    start_time = time()
    xk  = x0 # initialize xk at x0
    fk  = obj(nlp, xk) # evaluate the objective function at xk
    gk = grad(nlp, xk) # get gradient
    gnorm = gnorm0 = norm(gk) # get the norm of the gradient
    A = hess(nlp, xk)

    k = 0 # round 0
    Hk = A_inv
    error = false

    verbose && @printf "%2s %9s %9s\n" "k" "fk" "||∇f(x)||"
    verbose && @printf "%2d %9.2e %9.1e\n" k fk gnorm
    while gnorm > epsilon_abs + epsilon_rel * gnorm0 && (time() - start_time) <= max_time  # while the stopping conditio
      dk = - Hk * gk
      slope = dot(dk, gk) # slope= direction@gradient

      alpha = - transpose(gk) * dk ./ (transpose(dk)*A*dk)
      last_x = xk
      last_g = gk
      xk += alpha * dk
      fk = obj(nlp, xk)
      gk =  grad(nlp, xk)
```

```
      gnorm = norm(gk)
      Hk = A
      k += 1

      # prints
      if fk <= lower_bound
        xk = -Inf64
        @printf "The problem is unbounded below. \n"
        error = true
        break
      elseif k > max_iter
        @printf "Maximal number of iterations has been reached \n"
        error = true
        break
      elseif (time() - start_time) >= max_time
        @printf "Timeout has been reached"
        error = true
        break

      elseif neval_obj(nlp) > max_eval
        @printf "Max number of evaluations has been reached \n"
        error = true
        break
      end
      verbose && @printf "%2d %9.2e %9.1e %7.1e \n" k fk gnorm t
    end
    if error == false
      println("An optimal solution has been found in $(time() - start_time) seconds")
    else
      println("An error occured during solving")
    end
    return xk, obj(nlp, xk)
  end
```

```
bfgs_quadratic_versionJulien (generic function with 18 methods)
```

Exercice 1.2

Tester la méthode sur la matrice A et le vecteur b donné.

In [ ]:

```
n = 10
A = diagm(-1 => ones(n-1), 0 => 4*ones(n), 1 => ones(n-1))
b = A*[1:n;]

display(A)

display(inv(A))

display(b)
```

```
10×10 Matrix{Float64}:
 4.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0  4.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  1.0  4.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  4.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  4.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  4.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  4.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  4.0  1.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  4.0  1.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  4.0
10×10 Matrix{Float64}:
  0.267949     -0.0717968     0.0192379    …   7.08317e-6   -1.77079e-6
 -0.0717968     0.287187     -0.0769515       -2.83327e-5    7.08317e-6
  0.0192379    -0.0769515     0.288568         0.000106248  -2.65619e-5
 -0.00515478    0.0206191    -0.0773216       -0.000396657   9.91644e-5
  0.00138122   -0.00552487    0.0207183        0.00148038   -0.000370096
 -0.000370096   0.00148038   -0.00555143   …  -0.00552487    0.00138122
  9.91644e-5   -0.000396657   0.00148747       0.0206191    -0.00515478
 -2.65619e-5    0.000106248  -0.000398428     -0.0769515     0.0192379
  7.08317e-6   -2.83327e-5    0.000106248      0.287187     -0.0717968
 -1.77079e-6    7.08317e-6   -2.65619e-5      -0.0717968     0.267949
10-element Vector{Float64}:
  6.0
 12.0
 18.0
 24.0
 30.0
 36.0
 42.0
 48.0
 54.0
 49.0
```

On initialise x0 de façon random.

In [ ]:

```
f(x) = 0.5 * transpose(x) * A * x - transpose(b)*x # fonction objectif vue en classe
v = Vector{Float64}(undef,n)
x0 = rand!(v,-5:5)

nlp = ADNLPModel(f, x0);
arg, star = bfgs_quadratic_versionJulien(nlp, x0, inv(A))

arg2, star2, H2 = bfgs_quadratic(nlp, x0)

print(norm(H2 - inv(A)))
```

```
 k        fk ||∇f(x)||
 0  4.75e+02   1.3e+02
 1 -1.10e+03   7.9e-15 8.8e-02
An optimal solution has been found in 0.021844863891601562 seconds
 k        fk ||∇f(x)||
 0  4.75e+02   1.3e+02
 1 -1.08e+03   1.0e+01 8.8e-02
 2 -1.10e+03   2.1e+00 8.8e-02
 3 -1.10e+03   6.4e-01 8.8e-02
 4 -1.10e+03   1.4e-01 8.8e-02
 5 -1.10e+03   3.0e-02 8.8e-02
 6 -1.10e+03   4.1e-03 8.8e-02
 7 -1.10e+03   1.3e-03 8.8e-02
 8 -1.10e+03   2.3e-04 8.8e-02
 9 -1.10e+03   3.8e-05 8.8e-02
An optimal solution has been found in 0.0002257823944091797 seconds
0.18617958365213627
```

On remarque que d'utiliser directement inv(A) dans l'algorithme converge beaucoup plus vite. On remarque aussi que la méthode BFGS avec le alpha demandé converge en au plus n itérations et que le H obtenu à la fin est presqu'exactement le inv(A): comme vu en cours.

Exercice 1.3

Justifier pourquoi α est toujours bien défini théoriquement, i.e. $d^\top Ad > 0$?

Puisque $A \succ 0$, peu importe la valeur de $d \in \mathbb{R}^n$, cette expression est supérieure à 0.

Une façon intuitive de le voir, est que c'est la même forme qu'une fonction quadratique $\mathbb{R}^n \to \mathbb{R}$

$$f(d) = d^\top Ad$$

qui est strictement positive si $A \succ 0$, par définition.

Si A était $\succeq 0$, on n'aurait pas cette garantie.

Exercice 2

Dans cet exercice, on veut vérifier la convergence théorique et comparer les algorithmes codés précédemment sur la minimisation de la fonction Himmelblau , i.e. fH(x) = (x[2]+x[1].^2-11).^2+(x[1]+x[2].^2-7).^2.

In [ ]:
```julia
# fonction Himmelblau
f(x) = (x[2] + x[1].^2 - 11).^2 + (x[1] + x[2].^2 - 7).^2


# initialisation random de x0
n = 2
v = Vector{Float64}(undef,n)
x0 = rand!(v,-10:10)
display(x0)

# definition du nlp
nlp = ADNLPModel(f, x0);
```

```
2-element Vector{Float64}:
 -10.0
   8.0
```

On test Newton Armijo.

In [ ]:
```julia
println("Newton armijo")
arg1, star1, histo1 = newton_armijo_v2(nlp, x0)
```

```
Newton armijo
 k        fk ||∇f(x)||
 0  1.16e+04   4.1e+03
 1  2.02e+03   1.2e+03 1.0e+00
 2  3.01e+02   3.3e+02 1.0e+00
 3  3.22e+01   8.3e+01 1.0e+00
 4  1.67e+00   1.6e+01 1.0e+00
 5  1.43e-02   1.4e+00 1.0e+00
 6  1.63e-06   1.5e-02 1.0e+00
 7  2.22e-14   1.7e-06 1.0e+00
An optimal solution has been found in 0.042073965072631836 seconds
([-2.805118113094295, 3.1313125198536147], 2.2241294024570164e-14, [[-10.0, 8.0], [-6.834505743934033, 5.61727058981271
7], [-4.824632822547817, 4.169393320019825], [-3.6265783366086026, 3.4194770364368554], [-3.020562225136527, 3.1667400083
839614], [-2.8259698683049166, 3.132795818880928], [-2.8053417691757967, 3.131326226645438], [-2.805118113094295, 3.13131
25198536147]])
```

On teste LDLt armijo.

In [ ]:
```julia
println("LDLt armijo")
arg2, star2, histo2 = newton_ldlt_armijo(nlp, x0);
```

```
LDLt armijo
 k        fk ||∇f(x)||
 0  1.16e+04   4.1e+03
 1  2.02e+03   1.2e+03 1.0e+00
 2  3.01e+02   3.3e+02 1.0e+00
 3  3.22e+01   8.3e+01 1.0e+00
 4  1.67e+00   1.6e+01 1.0e+00
 5  1.43e-02   1.4e+00 1.0e+00
 6  1.63e-06   1.5e-02 1.0e+00
 7  2.22e-14   1.7e-06 1.0e+00
```

On teste notre version modifiée du BFGS Armijo.

In [ ]:
```
println("BFGS newton armijo")
arg3, star3, histo3 = bfgs_quasi_newton_armijo(nlp, x0)
```

```
BFGS newton armijo
 k        fk ||∇f(x)||
 0  1.16e+04   4.1e+03
 1  6.04e+03   2.9e+03 5.1e-03
 2  1.12e+02   6.4e+01 1.0e+00
 3  1.02e+02   6.6e+01 1.0e+00
 4  9.11e+01   6.7e+01 1.0e+00
 5  7.99e+01   6.7e+01 1.0e+00
 6  6.86e+01   6.6e+01 1.0e+00
 7  5.00e+01   1.1e+02 2.0e-01
 8  2.11e+01   3.9e+01 1.0e+00
 9  1.12e+01   1.6e+01 1.0e+00
10  9.74e+00   1.1e+01 1.0e+00
11  9.37e+00   9.6e+00 1.0e+00
12  6.41e+00   1.4e+01 1.0e+00
13  4.17e+00   1.8e+01 6.7e-01
14  2.07e+00   1.5e+01 1.0e+00
15  9.49e-01   1.1e+01 1.0e+00
16  1.70e-01   3.5e+00 1.0e+00
17  8.95e-03   1.1e+00 1.0e+00
18  4.52e-06   2.5e-02 1.0e+00
19  5.06e-10   2.9e-04 1.0e+00
An optimal solution has been found in 0.0001800060272216797 seconds
([2.9999971310233233, 1.9999978570739463], 5.055739241625803e-10, [[-10.0, 8.0], [9.453342892249566, -0.724716384321121
5], [1.3481744817400703, 0.18978136916413402], [1.4999590357443788, 0.23807186191710256], [1.657852676064072, 0.285097435
2791462], [1.819685073941883, 0.3304531515342577], [1.9827328187698234, 0.37382784018192633], [4.104538158779709, 0.91667
19202991473], [2.7951998689181385, 0.6226313780502017], [3.1407717429230817, 0.7359096041225623], [3.3365700757602768, 0.
8338846562837952], [3.302132109222859, 0.8562950657868973], [3.1120319928793205, 1.1662169585204052], [2.616295631797075,
2.2780596491404577], [2.9338383885987285, 1.6786717523002914], [2.908489956723726, 1.8462292779430811], [2.93419246401249
23, 2.083623306133875], [2.983319009235124, 2.0053267778959376], [3.0003793430441568, 1.9998284810066815], [2.99999713102
33233, 1.9999978570739463]])
```

À partir des historiques trouvés, on obtient les métriques demandées dans l'exercice en plus d'une approximation de l'odre de convergence (q).

Order estimation [edit]

A practical method to calculate the order of convergence for a sequence is to calculate the following sequence, which converges to q:[6]

$$q \approx \frac{\log\left|\frac{x_{k+1}-x_k}{x_k-x_{k-1}}\right|}{\log\left|\frac{x_k-x_{k-1}}{x_{k-1}-x_{k-2}}\right|}.$$

In [ ]:
```
"""
Function that returns all of the required metrics,
i.e., euclidian distance between xk and x_last, the linear convergence, the quadratic convergence.
"""

function get_distance_from_histo(histo)
    x_star = histo[end]
    dist = []
    conv = []
    conv_quad = []
    q_est = []
    for x in histo
        push!(dist, norm(x - x_star, 2))
    end

    for i in 1:(length(dist) -1)
        push!(conv, dist[i+1]./dist[i])
        push!(conv_quad, dist[i+1]./(dist[i]).^2)
    end

    for i in 3:(length(histo)-1)
        a = log.(norm((histo[i+1] - histo[i]))./norm((histo[i] - histo[i-1])))
        b = log.(norm((histo[i] - histo[i-1]))./norm((histo[i-1] - histo[i-2])))
        q = a./b
        push!(q_est, q)

    end
    return dist, conv, conv_quad, q_est

end
```

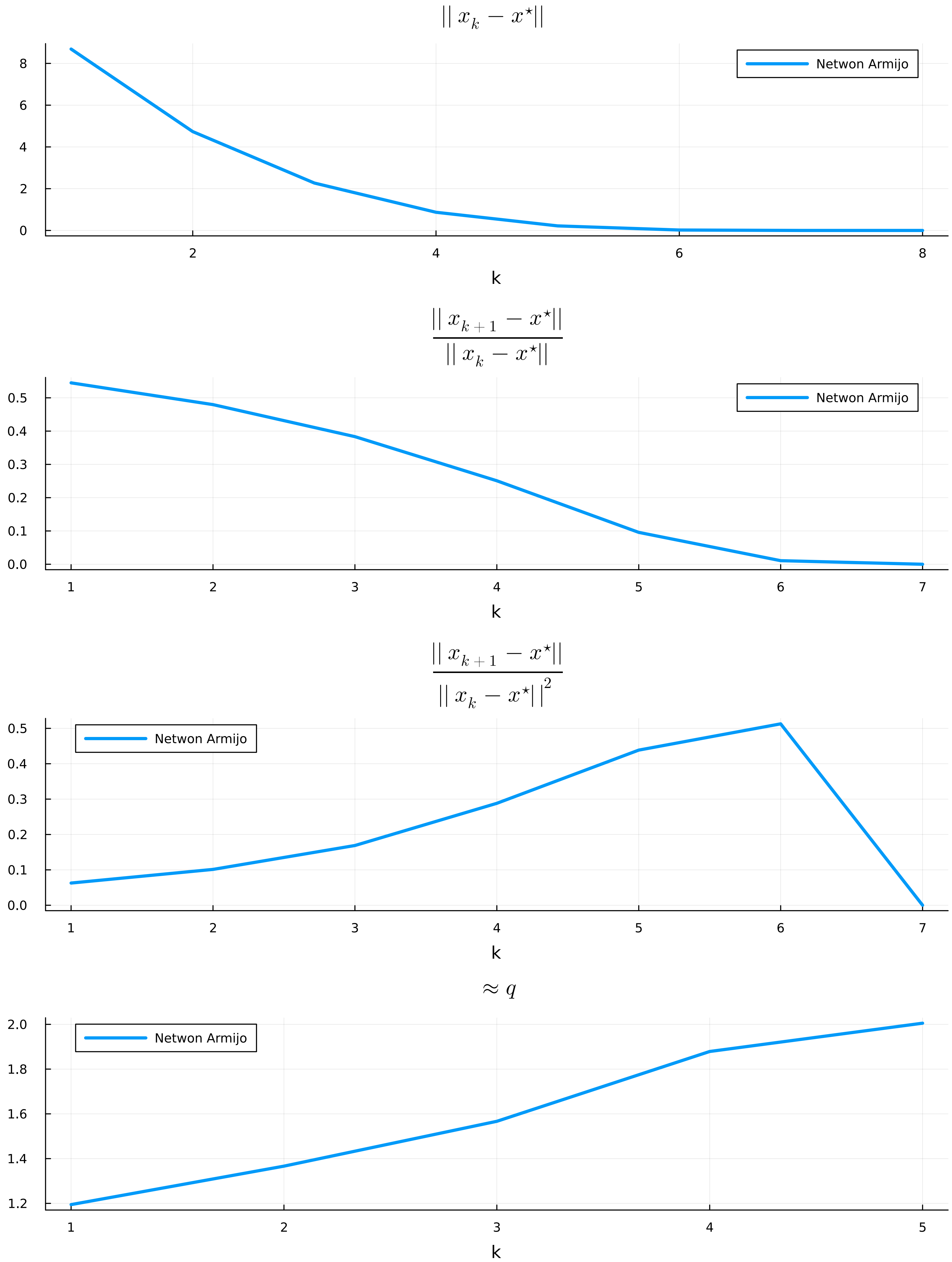
```
get_distance_from_histo (generic function with 1 method)
```

In [ ]:
```
dist1, conv1, conv_quad1, q1 = get_distance_from_histo(histo1);

dist2, conv2, conv_quad2, q2 = get_distance_from_histo(histo2);

dist3, conv3, conv_quad3, q3 = get_distance_from_histo(histo3);
```

On trace les graphiques pour Newton Armijo:

In [ ]:
```
k = 1:length(dist1)
l = 1:length(conv1)
m = 1:length(q1)

p1=plot(k, [dist1 ], title=L"$||x_{k} - x^{\star} ||$", label="Netwon Armijo", linewidth=3, xlabel="k")

p2 = plot(l, [conv1  ], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||}$", label="Netwon Armijo", linewi
p3 = plot(l, [conv_quad1 ], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||^2}$",label="Netwon Armijo", l
```

```
p4 = plot(m, q1, title=L"$\approx q$",label="Netwon Armijo", linewidth=3, xlabel="k")

plot!(size=(900,1200))

plot(p1, p2, p3, p4, layout=(4,1), legend=true)
```



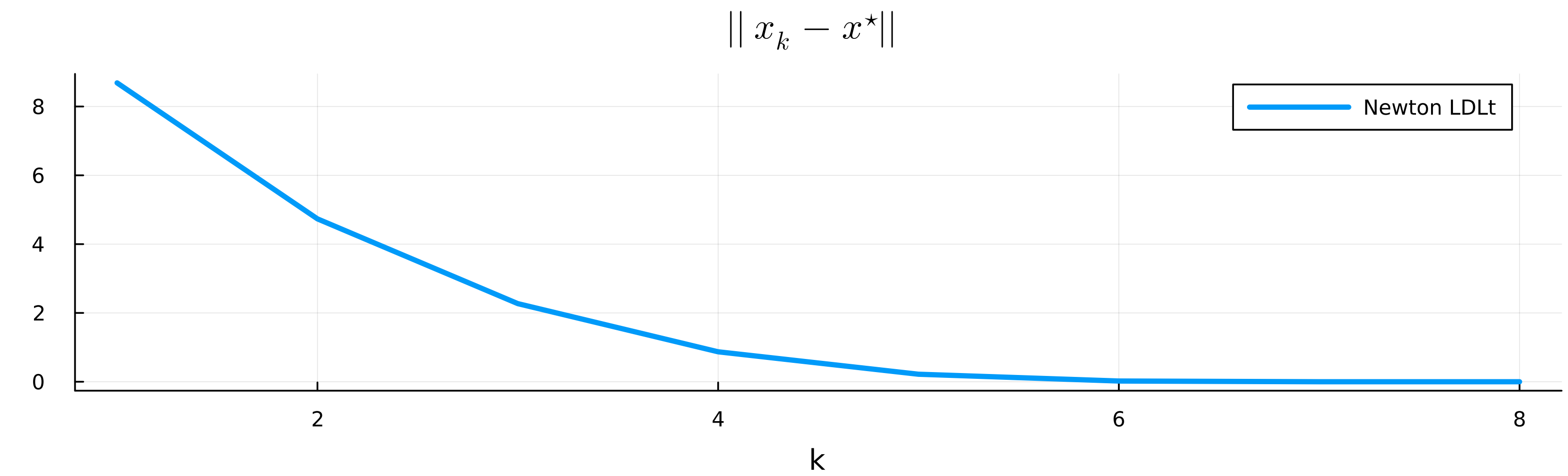
Difficile de faire une analyse graphique avec aussi peu d'itérations, mais on remarque que la méthode converge bel et bien à un optimum. On sait que si la fonction est différenciable trois fois, que la matrice hessienne est définie positive, que le gradient au point optimal est nul et qu'on utilise newton sans linesearch, alors on devrait observer une convergence quadratique.

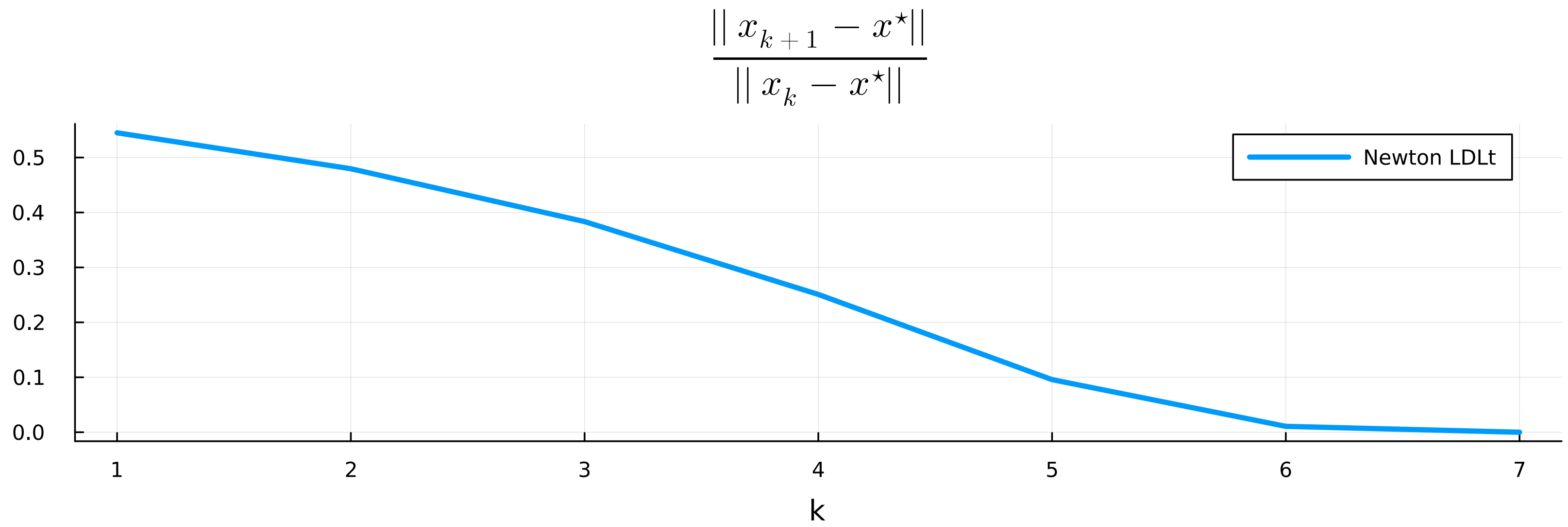
Dans ce cas-ci, on utilise une recherche d'Armijo alors le théorème de convergence n'est pas nécessairement respecté. Cependant, on remarque lors de la résolution que t = 1 pour chaque itération. Donc, on devrait observer une convergence quadratique en théorie. On remarque que notre estimation de q oscille entre 1 et 2 au cours de la résolution, ce qui semble indiquer une convergence expériomentale supralinéaire potentiellement quadratique. Il faudrait plus d'itérations pour pouvoir vraiment se prononcer.

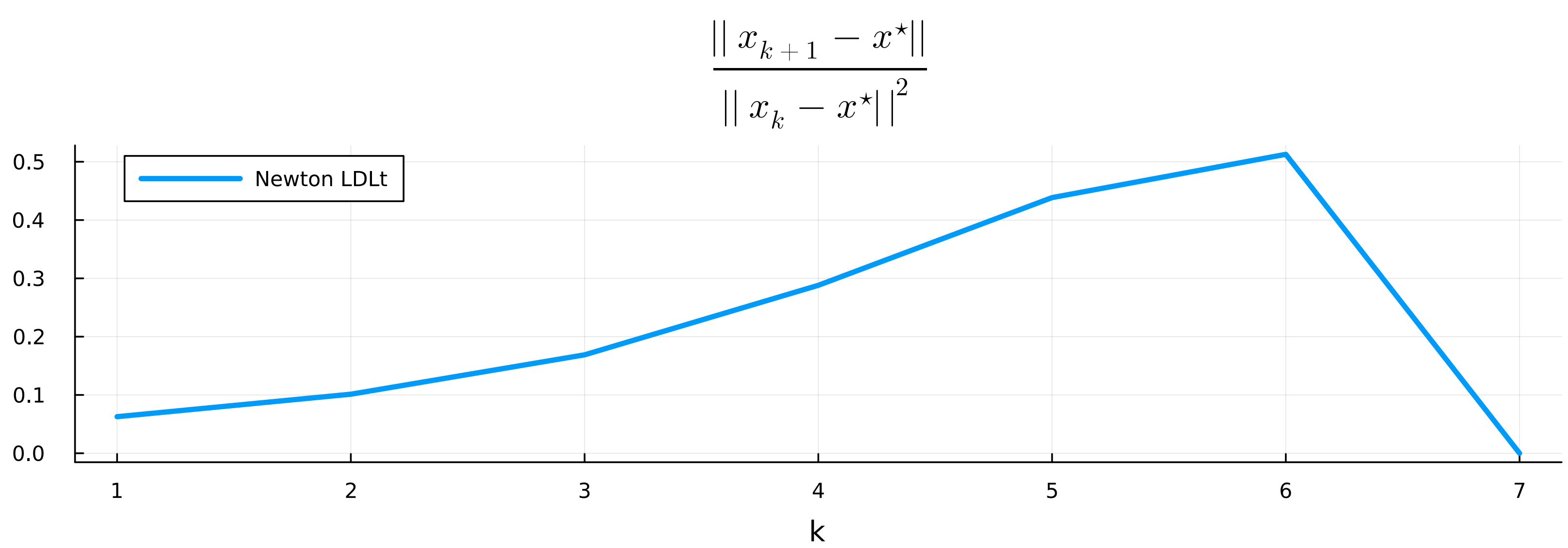
On trace les graphiques pour Newton LDLt:

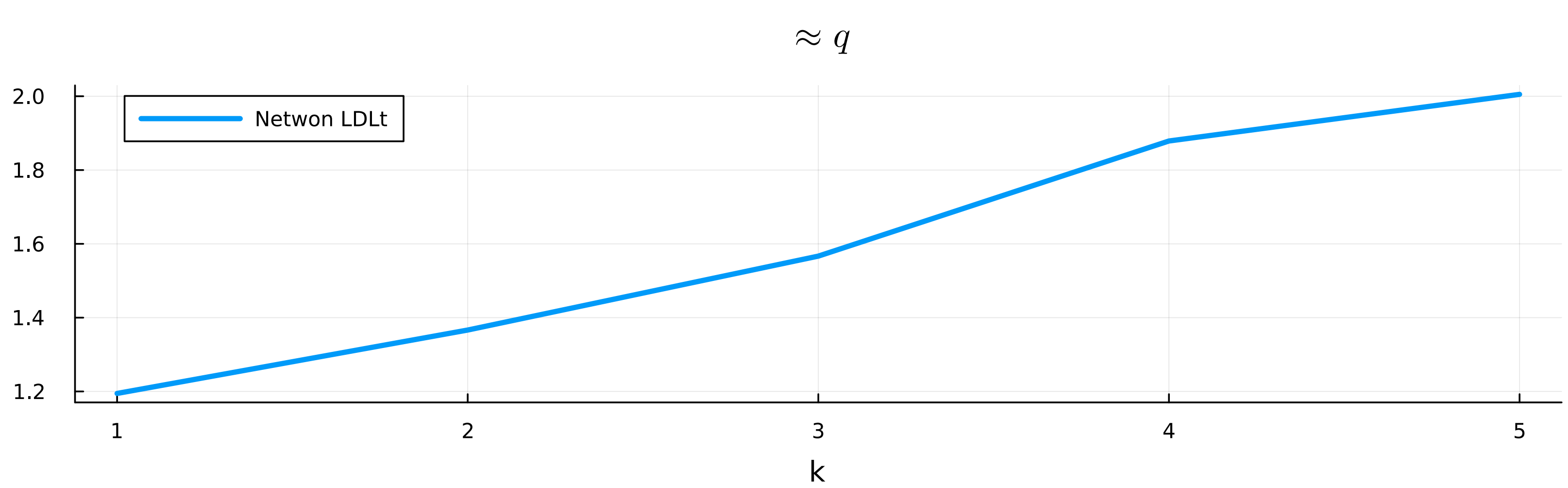
In [ ]:
```
k = 1:length(dist2)
l = 1:length(conv2)
m = 1:length(q2)
p1=plot(k, [dist2], title=L"$||x_{k} - x^{\star} ||$", label="Newton LDLt" , linewidth=3, xlabel="k")
p2 = plot(l, [conv2], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||}$", label="Newton LDLt" , linewidth
p3 = plot(l, [conv_quad2], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||^2}$",label="Newton LDLt" , lin
p4 = plot(m, q2, title=L"$\approx q$",label="Newton LDLt", linewidth=3, xlabel="k")

plot!(size=(900,1200))
```

```
plot(p1, p2, p3, p4, layout=(4,1), legend=true)
```





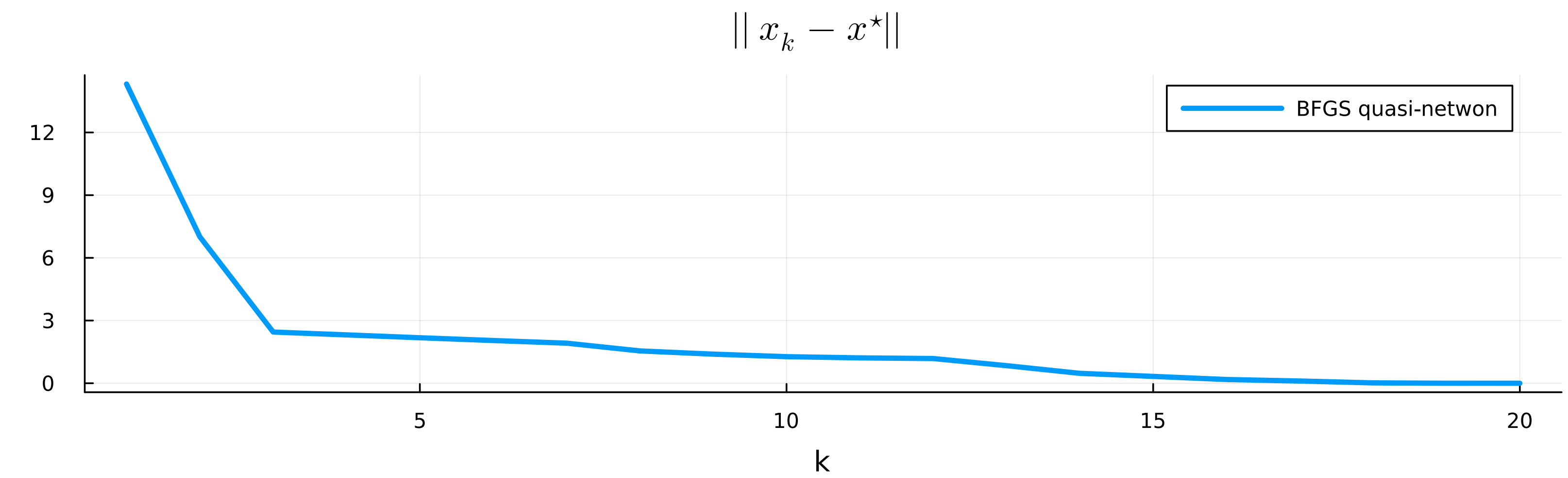


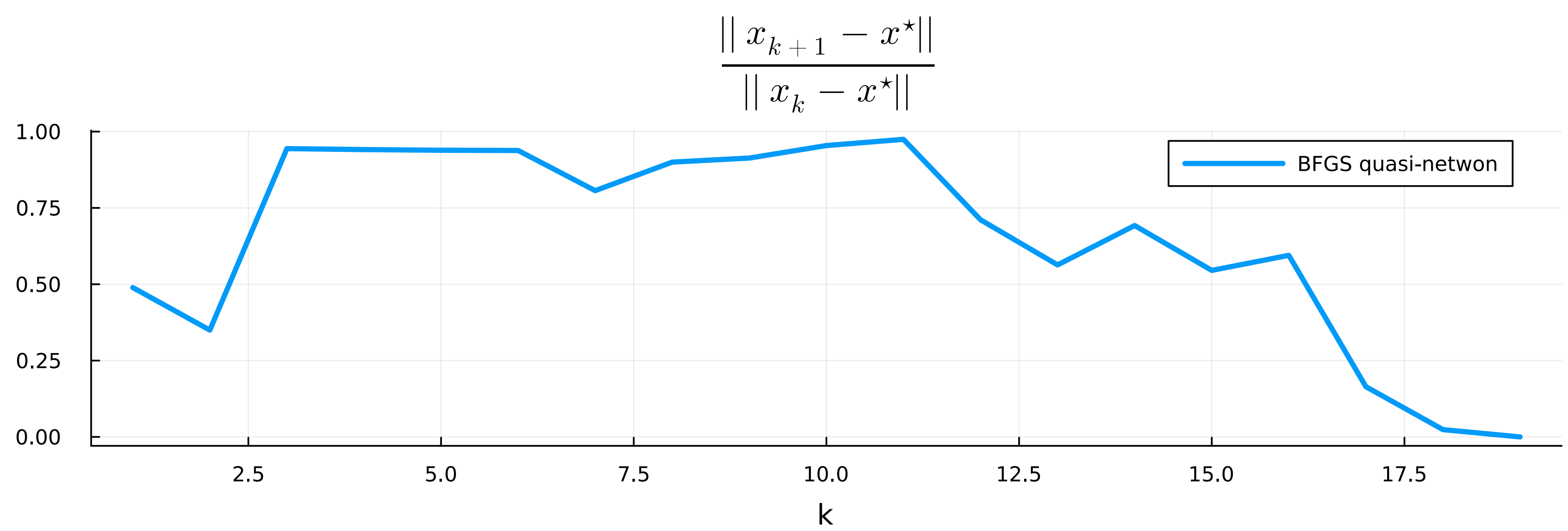


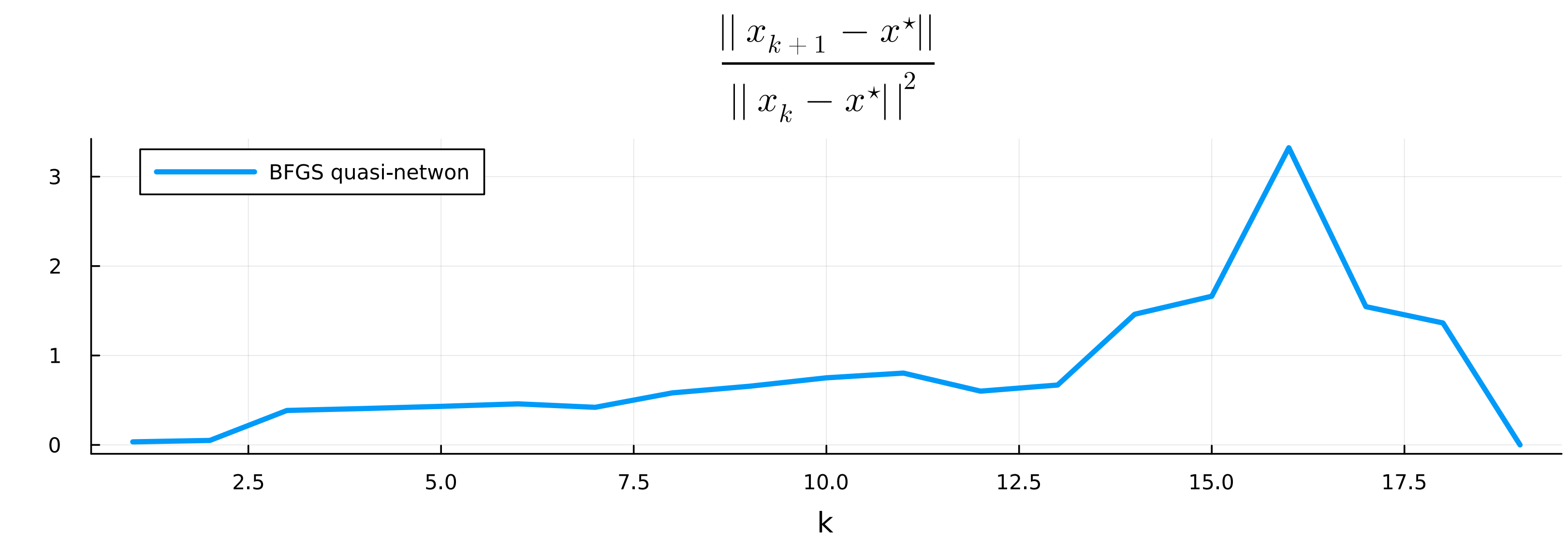
On remarque vraisemblablement la même convergence que pour la méthode précédente de Newton Armijo, soit supralinéaire pour la majorité de la recherche puis quadratique à la toute fin.

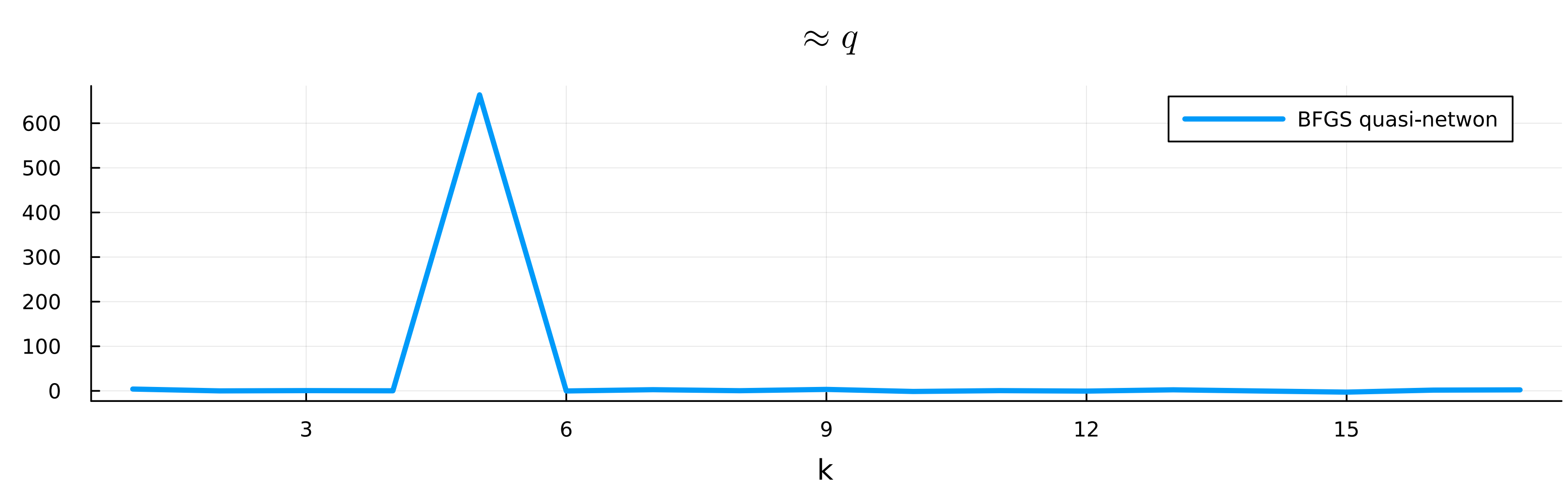
On trace les graphiques pour BFGS.

In [ ]:
```
k = 1:length(dist3)
l = 1:length(conv3)
m = 1:length(q3)
p1=plot(k, [dist3], title=L"$||x_{k} - x^{\star} ||$", label="BFGS quasi-netwon", linewidth=3, xlabel="k")
p2 = plot(l, [conv3], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||}$", label="BFGS quasi-netwon", line
p3 = plot(l, [conv_quad3], title=L"$\frac{||x_{k+1} - x^{\star} ||}{||x_{k} - x^{\star}||^2}$",label="BFGS quasi-netwon"
p4 = plot(m, q3, title=L"$\approx q$",label="BFGS quasi-netwon", linewidth=3, xlabel="k")

plot!(size=(900,1200))

plot(p1, p2, p3, p4, layout=(4,1), legend=true)
```









In [ ]:

```
print(q3)
```

```
Any[4.096362833391191, -0.008573268147973026, 0.5913312414323877, 0.19358471212360986, 663.6489265364356, -0.19108512237078298, 2.6654269842854728, 0.3886348665419723, 3.2972762681884005, -1.3031399413585825, 0.554256234987103, -0.48395991944725303, 2.3715351369255444, -0.24723714249974357, -2.7677140271448906, 1.7283365367032197, 2.291207862045402]
```

On sait que si les conditions de Dennis-Moré sont respectées, alors la convergence est supralinéaire. Puisque BFGS satisfie ces conditions théoriquement, c'est ce qu'on devrait observer. Ici, la convergence linéaire semble bornée par 1, ce qui indiquerait plutôt une convergence linéaire. Notre approximation de q ne fonctionne pas particulièrement bien non plus.