Romain Pujol

Matricule 2343161

LAB 3

Exercice 1

On va coder la méthode limited-BFGS qui prend racine dans la méthode BFGS mais qui nécessite moins d'allocation et de calcul.

In [49]:
```julia
using Pkg
Pkg.activate(".") #Accède au fichier Project.toml
Pkg.instantiate()
Pkg.status()
```

In [50]:
```julia
using LinearAlgebra, NLPModels, Printf
using JSOSolvers, BenchmarkTools, ADNLPModels

fH(x) = (x[2]+x[1].^2-11).^2+(x[1]+x[2].^2-7).^2
x0H = [10., 20.]
himmelblau = ADNLPModel(fH, x0H)

problem2 = ADNLPModel(x->-x[1]^2, ones(3))

roz(x) = 100 *  (x[2] - x[1]^2)^2 + (x[1] - 1.0)^2
rosenbrock = ADNLPModel(roz, [-1.2, 1.0])

f(x) = x[1]^2 * (2*x[1] - 3) - 6*x[1]*x[2] * (x[1] - x[2] - 1)
pb_du_cours = ADNLPModel(f, [-1.001, -1.001]) #ou [1.5, .5] ou [.5, .5]
```

```
ADNLPModel - Model with automatic differentiation backend ADModelBackend{
  ForwardDiffADGradient,
  ForwardDiffADHvprod,
  EmptyADbackend,
  EmptyADbackend,
  EmptyADbackend,
  ForwardDiffADHessian,
  EmptyADbackend,
}
  Problem name: Generic
   All variables: ████████████████████ 2      All constraints: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            free: ████████████████████ 2                 free: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0              low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
          infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            nnzh: (  0.00% sparsity)   3               linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                    nonlinear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                         nnzj: (------% sparsity)

  Counters:
             obj: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 grad: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 cons: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
        cons_lin: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0             cons_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 jcon: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           jgrad: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                  jac: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0              jac_lin: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         jac_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                jprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0            jprod_lin: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
       jprod_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               jtprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0           jtprod_lin: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
      jtprod_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 hess: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                hprod: ⋅⋅⋅⋅
⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           jhess: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               jhprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
```

In [51]:
```julia
using SolverCore
```

In [52]:
```julia
using LinearOperators
```

In [53]:
```julia
function armijo(xk, dk, fk, gk, slope, nlp :: AbstractNLPModel; τ1 = 1.0e-4, t_update = 1.5)
    t = 1.0
    fk_new = obj(nlp, xk + dk) # t = 1.0
    while fk_new > fk + τ1 * t * slope
      t /= t_update
      fk_new = obj(nlp, xk + t * dk)
    end
    return t, fk_new
```

```
end
```

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

In [54]:

```
function limited_bfgs(nlp      :: AbstractNLPModel;
    x        :: AbstractVector = nlp.meta.x0,
    atol     :: Real = √eps(eltype(x)),
    rtol     :: Real = √eps(eltype(x)),
    max_eval :: Int = -1,
    max_time :: Float64 = 30.0,
    f_min    :: Float64 = -1.0e16,
    verbose  :: Bool = true,
    mem      :: Int = 5)

start_time = time()
elapsed_time = 0.0

T = eltype(x)
n = nlp.meta.nvar

xt = zeros(T, n)
∇ft = zeros(T, n)

f = obj(nlp, x)
∇f = grad(nlp, x)

##################################################
H = InverseLBFGSOperator(n,mem)
##################################################

∇fNorm = norm(∇f) #nrm2(n, ∇f)
ϵ = atol + rtol * ∇fNorm
iter = 0

@info log_header([:iter, :f, :dual, :slope, :bk], [Int, T, T, T, T],
 hdr_override=Dict(:f=>"f(x)", :dual=>"‖∇f‖", :slope=>"∇fᵀd"))

optimal = ∇fNorm ≤ ϵ
unbdd = f ≤ f_min
tired = neval_obj(nlp) > max_eval ≥ 0 || elapsed_time > max_time
stalled = false
status = :unknown

while !(optimal || tired || stalled || unbdd)

##################################################
d = H*(-∇f)
##################################################
slope = dot(d, ∇f)
if slope ≥ 0
@error "not a descent direction" slope
status = :not_desc
stalled = true
continue
end

# Perform improved Armijo linesearch.
t, ft = armijo(x, d, f, ∇f, slope, nlp)

@info log_row(Any[iter, f, ∇fNorm, slope, t])

# Update L-BFGS approximation.
xt = x + t * d
∇ft = grad(nlp, xt) # grad!(nlp, xt, ∇ft)
##################################################
push!(H,xt-x,∇ft-∇f)
##################################################

# Move on.
x = xt
f = ft
∇f = ∇ft

∇fNorm = norm(∇f) #nrm2(n, ∇f)
iter = iter + 1

optimal = ∇fNorm ≤ ϵ
unbdd = f ≤ f_min
elapsed_time = time() - start_time
tired = neval_obj(nlp) > max_eval ≥ 0 || elapsed_time > max_time
end

@info log_row(Any[iter, f, ∇fNorm])
```

```julia
if optimal
status = :first_order
elseif tired
if neval_obj(nlp) > max_eval ≥ 0
status = :max_eval
elseif elapsed_time > max_time
status = :max_time
end
elseif unbdd
status = :unbounded
end

return GenericExecutionStats(
nlp,
status=status,
solution=x,
objective=f,
dual_feas=∇fNorm,
iter=iter,
elapsed_time=elapsed_time,
)
end
```

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

In [55]:

```julia
using Test
# Demander le test secret pour lbfgs
@testset begin
    #Unit/Validation Tests
    using Logging, Test
    stats = with_logger(NullLogger()) do
        limited_bfgs(himmelblau)
    end
    @test stats.status == :first_order
    @test stats.solution ≈ [3.584428266659278, -1.8481265666485827] atol = 1e-6
    @show (stats.status, stats.solution)
    stats = with_logger(NullLogger()) do
        limited_bfgs(problem2)
    end
    @test stats.status == :unbounded
    @show (stats.status, stats.solution)
    stats = with_logger(NullLogger()) do
        limited_bfgs(rosenbrock)
    end
    @test stats.solution ≈ [1., 1.] atol = 1e-6
    @show (stats.status, stats.solution)
    stats = with_logger(NullLogger()) do
        limited_bfgs(pb_du_cours, x = [-1.001, -1.001])
    end
    @test stats.status == :unbounded
    @show (stats.status, stats.solution)
    stats = with_logger(NullLogger()) do
        limited_bfgs(pb_du_cours, x = [1.5, .5])
    end
    @test stats.status == :first_order
    @test stats.solution ≈ [1., 0.] atol = 1e-6
    @show (stats.status, stats.solution)
    stats = with_logger(NullLogger()) do
        limited_bfgs(pb_du_cours, x = [.5, .5])
    end
    @test stats.status == :first_order
    @test stats.solution ≈ [1., 0.] atol = 1e-6
    @show (stats.status, stats.solution)
end
```

```
Test.DefaultTestSet("test set", Any[], 9, false, false, true, 1.707683086045e9, 1.707683086662e9, false, "c:\\Users\\romai\\POLYMTL\\MTH8408-Hiv24\\lab3\\lab3.ipynb")
```

In [56]:

```julia
@benchmark limited_bfgs(himmelblau)
```

```
BenchmarkTools.Trial: 3282 samples with 1 evaluation.
 Range (min … max):  806.400 μs …   9.159 ms  ┊ GC (min … max): 0.00% … 79.27%
 Time  (median):     1.438 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.516 ms ± 478.136 μs  ┊ GC (mean ± σ):  1.05% ±  3.80%
```

```
 806 μs          Histogram: frequency by time        2.52 ms <

 Memory estimate: 152.89 KiB, allocs estimate: 2202.
```

In [57]:

```julia
@benchmark lbfgs(himmelblau)
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  19.500 μs …  8.475 ms  ┊ GC (min … max): 0.00% … 98.77%
 Time  (median):     20.700 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   22.666 μs ± 84.604 μs  ┊ GC (mean ± σ):  3.69% ±  0.99%

  19.5 μs      Histogram: log(frequency) by time        36 μs <

 Memory estimate: 10.77 KiB, allocs estimate: 269.
```

La fonction 'lbfgs' du modèle requiert moins de temps de calcul (21μs contre $1.4ms$) et d'allocation (269 contre 2202) pour résoudre le problème Himmelblau. La fonction 'lbfgs' doit être codée de manière plus efficace.

Pour tester "facilement" plusieurs valeurs de τ_1 dans la fonction armijo, il suffit de le placer en tant que Keyword Arguments et non pas en tant qu'Optional Argument. L'entête de la fonction armijo deviendrait : function armijo(xk, dk, fk, gk, slope, nlp :: AbstractNLPModel, τ_1 ; t_update = 1.5).

Exercice 2

On va maintenant coder la méthode du gradient conjugué.

In [58]:
```julia
function cg_optim(H, ∇f)
    #setup the tolerance:
    n∇f = norm(∇f)
#####################################
    ϵk = minimum([0.5,sqrt(n∇f)])*n∇f
####################################
    n = length(∇f)
    z = zeros(n)
    r = ∇f
    d = -r

    j = 0
    while norm(r) ≥ ϵk && j < 3 * n
###############################################
        if dot(d, H * d) ≤ 0
            if j==0
                return -∇f
            else
                return z
            end
        end
##############################################
        α = dot(r,r)/dot(d, H*d)
##############################################
        z += α * d
        nrr2 = dot(r, r)
        r += α * H * d
##############################################
        if nrr2<ϵk
            return z
        end
        β  = dot(r,r)/dot(r-α * H * d,r-α * H * d)
##############################################
        d  = -r + β * d
        j += 1
    end
    return z
end
```

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

In [59]:
```julia
function armijo_Newton_cg(nlp      :: AbstractNLPModel;
    x        :: AbstractVector = nlp.meta.x0,
    atol     :: Real = √eps(eltype(x)),
    rtol     :: Real = √eps(eltype(x)),
    max_eval :: Int = -1,
    max_time :: Float64 = 30.0,
    f_min    :: Float64 = -1.0e16)

start_time = time()
elapsed_time = 0.0

T = eltype(x)
n = nlp.meta.nvar

f = obj(nlp, x)
∇f = grad(nlp, x)
#################################################
H = hess_op(nlp,x)
```

```julia
##################################################

∇fNorm = norm(∇f) #nrm2(n, ∇f)
ϵ = atol + rtol * ∇fNorm
iter = 0

@info log_header([:iter, :f, :dual, :slope, :bk], [Int, T, T, T, T],
hdr_override=Dict(:f=>"f(x)", :dual=>"‖∇f‖", :slope=>"∇fᵀd"))

optimal = ∇fNorm ≤ ϵ
unbdd = f ≤ f_min
tired = neval_obj(nlp) > max_eval ≥ 0 || elapsed_time > max_time
stalled = false
status = :unknown

while !(optimal || tired || stalled || unbdd)

    d = cg_optim(H, ∇f)

    slope = dot(d, ∇f)
    if slope ≥ 0
    @error "not a descent direction" slope
    status = :not_desc
    stalled = true
    continue
    end

    # Perform improved Armijo linesearch.
    t, f = armijo(x, d, f, ∇f, slope, nlp)

    @info log_row(Any[iter, f, ∇fNorm, slope, t])

    # Update L-BFGS approximation.
    x += t * d
    ∇f = grad(nlp, x)
    ##################################################
    H = hess_op(nlp,x)
    ##################################################

    ∇fNorm = norm(∇f) #nrm2(n, ∇f)
    iter = iter + 1

    optimal = ∇fNorm ≤ ϵ
    unbdd = f ≤ f_min
    elapsed_time = time() - start_time
    tired = neval_obj(nlp) > max_eval ≥ 0 || elapsed_time > max_time
end

@info log_row(Any[iter, f, ∇fNorm])

if optimal
status = :first_order
elseif tired
if neval_obj(nlp) > max_eval ≥ 0
status = :max_eval
elseif elapsed_time > max_time
status = :max_time
end
elseif unbdd
status = :unbounded
end

return GenericExecutionStats(nlp, status = status, solution=x, objective=f, dual_feas=∇fNorm,
         iter=iter, elapsed_time=elapsed_time)
end
```

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

In [60]:

```julia
stats = with_logger(NullLogger()) do
    armijo_Newton_cg(himmelblau)
end
@test stats.status == :first_order
@test stats.dual_feas ≤ 1e-6 + 1e-6 * norm(grad(himmelblau, himmelblau.meta.x0))
```

Test Passed

In [61]:

```julia
stats = with_logger(NullLogger()) do
    armijo_Newton_cg(problem2)
end
@test stats.status == :unbounded
```

Test Passed

In [62]:

```julia
stats = with_logger(NullLogger()) do
    armijo_Newton_cg(rosenbrock)
```

```julia
end
@test stats.solution ≈ [1., 1.] atol = 1e-5
```

Test Passed

In [63]:
```julia
stats = with_logger(NullLogger()) do
    armijo_Newton_cg(pb_du_cours, x = [.5, .5])
end
@test stats.status == :first_order
@test stats.solution ≈ [1., 0.] atol = 1e-6
```

Test Passed

Les tests sont réussis !

Maintenant que les implémentations sont faites, on peut passer aux exercices du PDF.

PDF Exercice 1

On va résoudre le problème "genrose" que j'ai résolu au dernier laboratoire comme ça je sais vers quoi on doit converger : le vecteur rempli de 1.

In [64]:
```julia
using OptimizationProblems
using ADNLPModels, OptimizationProblems.ADNLPProblems

n = 100
model = genrose(n=n)
@test unconstrained(model) #on vérifie qu'on est bien dans un modèle sans contrainte
```

Test Passed

In [65]:
```julia
limited_bfgs(model).solution
```

```
100-element Vector{Float64}:
 1.0000000001795355
 1.0000000002407166
 1.0000000003020546
 1.0000000003505547
 1.0000000004277545
 1.000000000414482
 1.0000000003459237
 1.0000000000797196
 0.9999999998438878
 0.9999999995206177
 ⋮
 0.9999999998406989
 0.9999999997229829
 0.9999999994597167
 0.9999999989519792
 0.9999999978546621
 0.9999999956862801
 0.9999999912861994
 0.9999999825027955
 0.9999999648957821
```

Ce qui est le cas ! On converge bien vers le point attendu.

PDF Exercice 2

Même combat que l'exercice précédent, je vais prendre un problème résolu au dernier lab, le problème tridia où la solution est $v = (\frac{1}{2^{i-1}})_{i \in [1,n]}$ où n est la taille du problème.

In [66]:
```julia
n=100
model2=tridia(n=n)
@test unconstrained(model2)
```

Test Passed

In [67]:
```julia
armijo_Newton_cg(model2).solution
```

```
100-element Vector{Float64}:
  1.0000074265509098
  0.5000042348966984
  0.25000230542583446
  0.12500122746441464
  0.0625006450737
  0.031250336119284856
  0.01562517408047534
  0.00781258975340332
  0.0039062961147050875
  0.001953148627818381
  ⋮
  1.071478929090357e-9
 -1.5501026627733622e-9
  2.1098051585679768e-9
 -2.6797278095709528e-9
  3.138352395940997e-9
 -3.3245337811783996e-9
  3.075454847120296e-9
 -2.2922553730372632e-9
  1.0177602433463937e-9
```

C'est à peu près le cas, les erreurs numériques se voient beaucoup pour la fin de la solution mais ça reste tolérable car on a toujours une certaine tolérance de l'ordre de 10^{-6} en général.

PDF Exercice 3

In [72]:

```julia
using NLPModels, NLPModelsJuMP, OptimizationProblems, OptimizationProblems.PureJuMP, SolverBenchmark
using ADNLPModels

n = 50 # taille des problemes, 4 < n < 101 pour satisfaire skip_if

solvers=Dict(
    :limited_bfgs_1 => model -> limited_bfgs(model, mem=1),
    :limited_bfgs_5 => model -> limited_bfgs(model, mem=5),
    :limited_bfgs_20 => model -> limited_bfgs(model, mem=20),
    )

ad_problems = (eval(Meta.parse(problem))(;n) for problem ∈ OptimizationProblems.meta[!, :name])

stats = bmark_solvers(
  solvers, ad_problems,
  skipif=prob -> (!unconstrained(prob) || get_nvar(prob) > 100 || get_nvar(prob) < 5),
  )
```

```
UndefVarError: `AMPGO02` not defined
```

In [ ]:

```
Malheureusement je n'ai pas réussi à faire fonctionner le benchmark...
```

PDF Exercice 4

Processing math: 100%