

Rapport de projet - phase 2

MTH8408

Oihan Cordelier, Oussama Mouhtal

Lien GitHub

Ce projet est accessible sur le dépôt GitHub au lien suivant : https://github.com/oihanc/mth8408_projet.

Contenu

Ce rapport est divisé en deux parties :

- Une première partie consacrée à la comparaison des performances de **LBFGS** (pour différentes tailles de mémoire) avec celles de **CG** et **DIOM**.
- Une seconde partie portant sur la résolution de problèmes non linéaires.

Retour sur la problématique

Le gradient conjugué est une méthode répandue pour résoudre les modèles quadratiques des algorithmes de région de confiance, cependant le CG souffre de perte d'orthogonalisation. Bourhis et al. (2019) ont démontré que CG revient à faire L-BFGS avec une mémoire de 1. L'idée est donc de valider si L-BFGS avec une mémoire supérieure à 1 aide l'efficacité des algorithmes de région de confiance. Bourhis et al. ont déjà démontré qu'augmenter la mémoire aide à résoudre les problèmes quadratiques avec moins d'itérations. De plus, une autre idée serait de tester un sous-solveur similaire à CG, cependant apportant des corrections pour l'orthogonalisation, telle que DIOM.

Résolution de quadratique convexe

Cette partie consiste à comparer CG et DIOM à LBFGS. Le tableau ci-dessous récapitule l'ensemble des matrices SDP utilisées dans cette section.

Nom	n	Condition number
494_bus	494	2.415×10^6
1138_bus	1138	8.572×10^6
662_bus	662	7.941×10^5
bcsstk06	420	7.570×10^6

```
"""
    get_mm()
Charge une matrice depuis la SuiteSparse Matrix Collection
"""
function get_mm(matrix_name)
    ssmc = ssmc_db()
    pb = ssmc_matrices(ssmc, "", matrix_name)
    fetch_ssmc(pb, format="MM")
end
```

```

pb_path = fetch_ssmc(pb, format="MM")
path_mtx = pb_path[1]
A = MatrixMarket.mmread(joinpath(path_mtx, matrix_name * ".mtx"))
#b = MatrixMarket.mmread(joinpath(path_mtx, matrix_name * "_b.mtx"))
return A
end

"""
    memory(n, p)
Génère des indices équidistants pour mémoire limitée
"""
function memory(n, p)
    @assert 1 ≤ p ≤ n "p doit être entre 1 et n"
    indices = [floor{Int, i*n/p} for i in 1:p]
    indices = unique(sort(indices))
    return indices
end

"""
    cg_lbfgs(A, b, name, listmem, atol, rtol)
Compare CG et L-BFGS pour différentes tailles de mémoire, et sauvegarde un graphique
"""
function cg_lbfgs(A, b, name, listmem, atol, rtol)

    (xcg,statscg) = cg(A, b; atol=atol, rtol=rtol,history=true)
    (xlbfgs,statslbfgs) = lbfgs(A, b; atol=atol, rtol=rtol, mem = 1)
    gr()
    plot(statscg.residuals, label="||r|| cg ", lw=1, yaxis=:log, linestyle = :dot, xlabel="Itération", legend=:none)
    plot!(statslbfgs.residuals, label="||r|| lbfgs $(m = 1)", lw=1, linestyle = :dash)

    for mem in listmem
        (xlbfgs,statslbfgs) = lbfgs(A, b; atol=atol, rtol=rtol, mem = mem)
        p = round{Int64, 100 * mem / n}
        if p > 0 && p < 100
            plot!(statslbfgs.residuals, label="||r|| lbfgs $(m = p)%", lw=1,linestyle = :dash)
        else
            plot!(statslbfgs.residuals, label="||r|| lbfgs $(m = p)%", lw=1,linestyle = :dot)
        end
    end

    end

    savefig("CG_versus_lbfgs_$(name).pdf")
end

"""
    diom_lbfgs(A, b, name, listmem, atol, rtol)
Compare DIOM et L-BFGS pour différentes tailles de mémoire, et sauvegarde un graphique
"""
function diom_lbfgs(A, b, name, listmem, atol, rtol)

    (xlbfgs,statslbfgs) = lbfgs(A, b; atol=atol, rtol=rtol, mem = 2)
    (xdiom,statsdiom) = diom(A, b; memory=2, atol=atol, rtol=rtol,history=true)

```

```

gr()
plot(statslbfgs.residuals, label="||r|| lbfgs $(m = 2)", lw=1, yaxis=:log, linestyle = :dash, xlabel="It",
plot!(statsdiom.residuals, label="||r|| diom $(m = 2)", linestyle = :dot, lw=1)
for mem in listmem
    p = round{Int64, 100 * mem / n}
    if mem > 2 && p > 0
        (xlbfgs, statslbfgs) = lbfgs(A, b; atol=atol, rtol=rtol, mem = mem)
        (xdiom, statsdiom) = diom(A, b; memory = mem, atol=atol, rtol=rtol, history=true)
        plot!(statslbfgs.residuals, label="||r|| lbfgs $(m = p)%", linestyle = :dash, lw=1)
        plot!(statsdiom.residuals, label="||r|| diom $(m = p)%", linestyle = :dot, lw=1)
    end
end
savefig("Diom_versus_lbfgs_$(name).pdf")
end

"""
    orthogonality_loss(A, b; maxiter, mem)
Calculer la perte d'orthogonalité associée aux directions L-BFGS pour plusieurs itérations

## Les tolérances dans lbfgs sont ceux par défaut atol=1e-18, rtol=1e-18
"""

function orthogonality_loss(A, b; maxiter=10, mem=1)
    Loss = Float64[]
    for i in 2:maxiter
        (xlbfgs, statslbfgs) = lbfgs(A, b; mem=mem, itmax = i)
        U = tril(statslbfgs.PAP, -1) # partie triangulaire inférieure stricte
        I_U = I + U # U + I
        loss = norm(U) # || (U + I)^(-1) * U ||_2
        push!(Loss, loss)
    end
    return Loss
end

```

orthogonality_loss (generic function with 1 method)

Comparaison CG à LBFGS

Dans cette partie, nous comparons la méthode du gradient conjugué (CG) à LBFGS, en faisant varier la taille de la mémoire utilisée. Nous traçons la norme du résidu en fonction des itérations. Voici les remarques principales :

- CG génère des résidus très similaires à ceux de LBFGS avec une mémoire de taille 1. Dans les deux cas, la convergence n'est pas atteinte en n itérations, en raison de la perte d'orthogonalité.
- En augmentant la taille de la mémoire, la convergence de LBFGS s'améliore. Avec la mémoire complète, la solution est retrouvée en n itérations (étant donné que $\text{atol} = \text{rtol} = 10^{-9}$). Cependant, on observe que lorsque la norme du résidu devient très faible, LBFGS peine à continuer la réduction du résidu.
- La matrice utilisée est 494_bus.

```

A = get_mm("494_bus")
n,n = size(A)
b = randn(eltype(A), n)

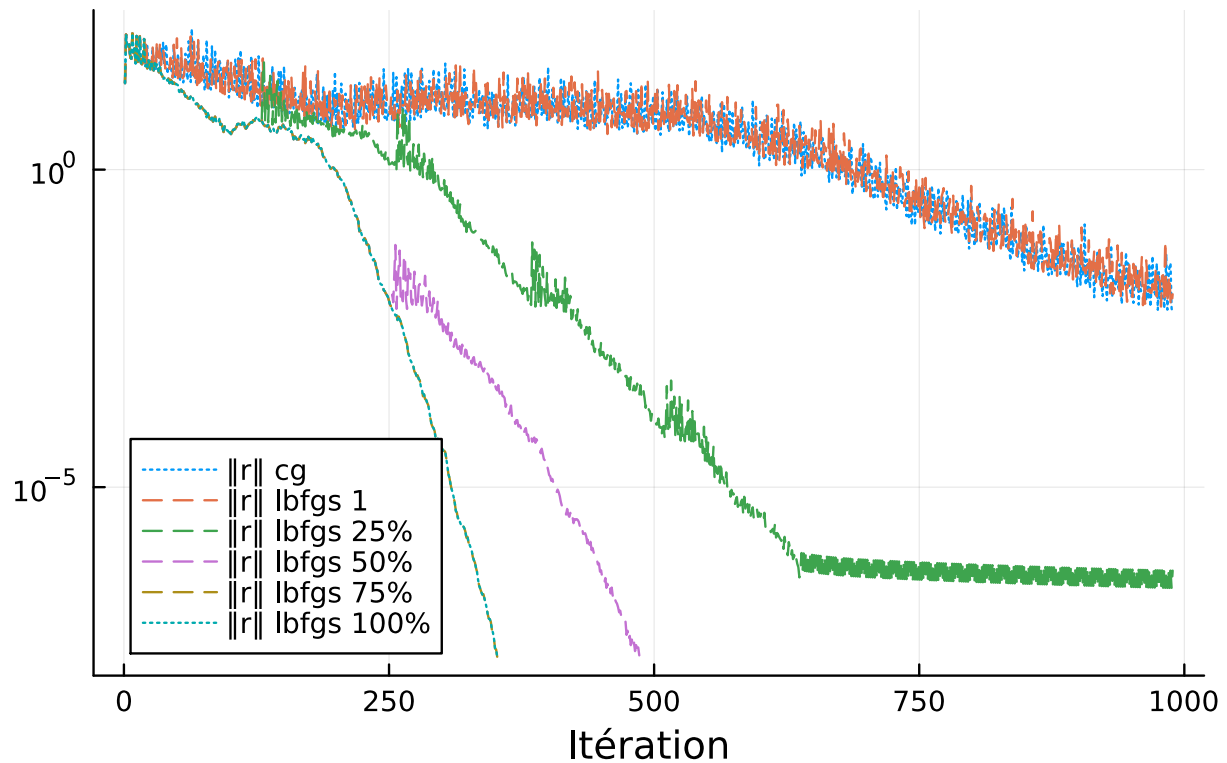
```

```

atol = 1e-9
rtol = 1e-9
p=4
listmem = memory(n, p)
cg_lbfgs(A, b, "494_bus", listmem, atol, rtol)

```

"/home/corde/mth8408/projet/phase_2/CG_versus_lbfgs_494_bus.pdf"



Perte de conjugaison dans les directions de LBFGS

- Dans cette partie on vise à analyser la perte de conjugaison des directions générées. Soit $P_k \in \mathbb{R}^{n \times k}$ la matrice contenant les k premières directions normalisées d_i . Si la famille $\{d_i\}$ est une base A -conjuguée, alors on devrait avoir :

$$P_k^\top A P_k = I.$$

Comme cette matrice est symétrique, on peut la décomposer en une somme :

$$P_k^\top A P_k = U_k^\top + D_k + U_k.$$

Une façon de mesurer la perte de conjugaison consiste donc à évaluer la norme de U_k , qui devrait être nulle si la conjugaison est parfaite. La figure ci dessous illustre l'évolution de la norme de U_k au cours des itérations pour différentes valeur du mémoire. La mtrice utiliser dans cette partie est **bcsstk06**.

- Il est important de souligner qu'une amélioration significative de la perte de conjugaison est obtenue on augmentant la mémoire de la matrice **lbfgs**. Meme on peut remarquer que dès une itération supérieur à la taille de la mémoire on perte la propriété de conjugaison des directions (ce n'est pas une généralisation).

```

A = get_mm("bcsstk06")
n,n = size(A)

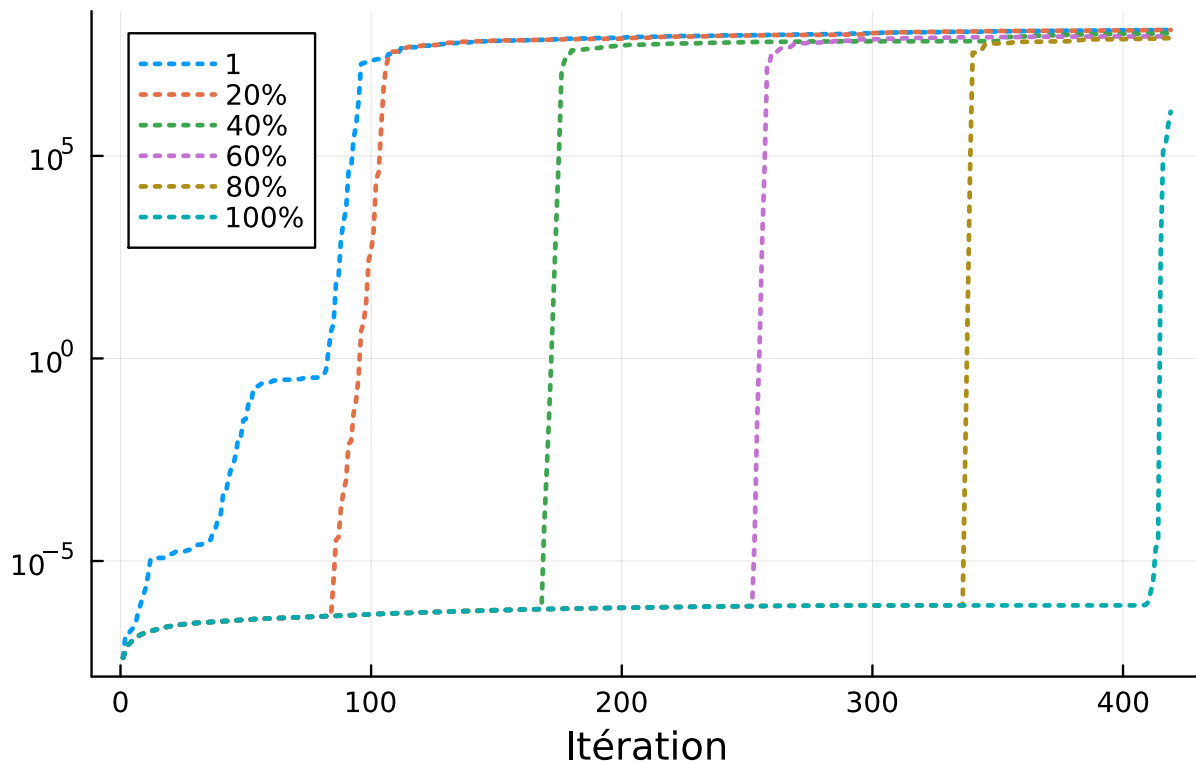
```

```

b = randn(eltype(A), n)
p=5
listmem = memory(n, p)
Loss = orthogonality_loss(A, b; maxiter = n)
plot(Loss, label="$m = 1)", lw=2, yaxis=:log, linestyle = :dot, xlabel="Itération", legend = :topleft)
for mem in listmem
    Loss = orthogonality_loss(A, b; maxiter = n, mem=mem)
    p = round{Int64, 100 * mem / n}
    if p > 0
        plot!(Loss, label="$m = p)", linestyle = :dot, lw=2)
    end
end
end
savefig("Loss_conjugacy.pdf")

```

"/home/corde/mth8408/projet/phase_2/Loss_conjugacy.pdf"



Comparaison DIOM à LBFGS

- Cette partie est dédiée à la comparaison entre DIOM et LBFGS. La méthode DIOM est implémentée de manière à conserver en mémoire les vecteurs de la base générée au cours du processus de Krylov. En arithmétique exacte, cet algorithme est équivalent à la méthode du gradient conjugué (CG) pour les matrices SDP. De plus, en pratique, DIOM se comporte comme CG avec réorthogonalisation.
- On remarque que LBFGS et DIOM génèrent des résidus presque identiques au début. Toutefois, à partir d'un certain nombre d'itérations — lorsque la norme du résidu devient très faible — DIOM surpasse LBFGS, ce dernier n'étant plus capable de faire progresser la réduction du résidu.

```

A = get_mm("494_bus")
n,n = size(A)

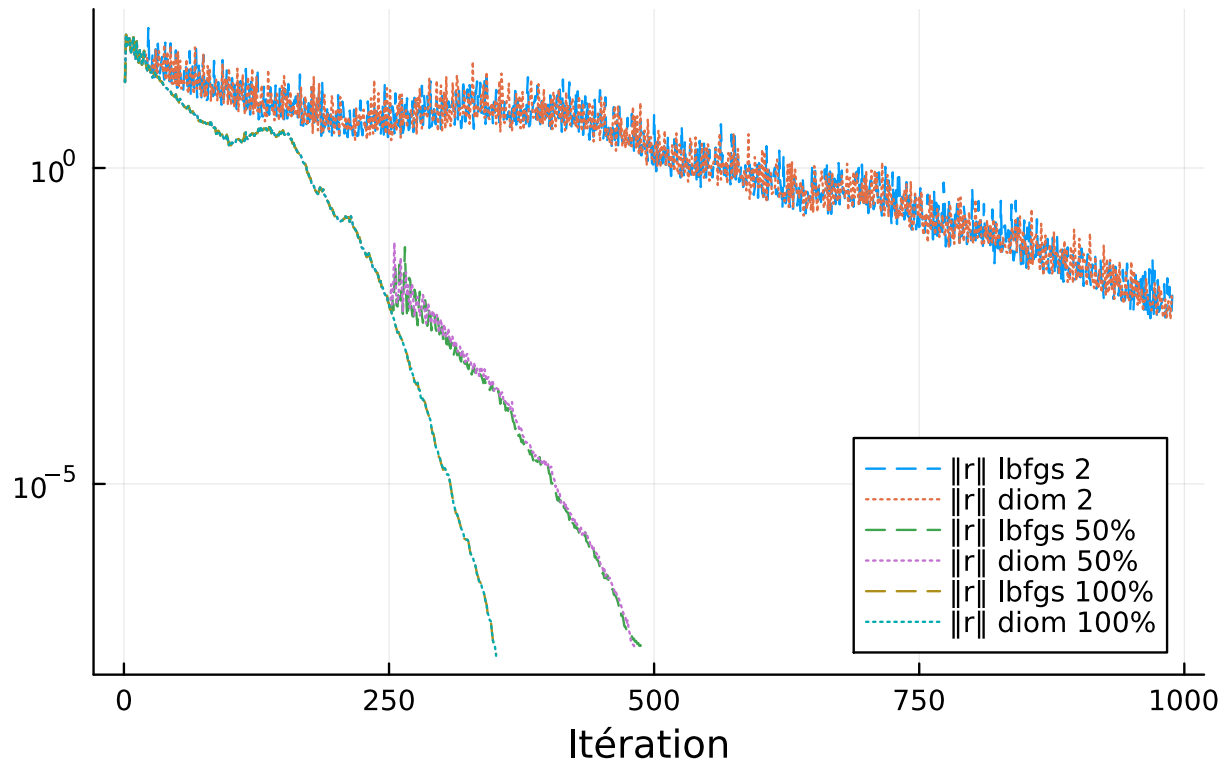
```

```

b = randn(eltype(A), n)
p=2
listmem = memory(n, p)
diom_lbfgs(A, b, "494_bus", listmem, atol, rtol)

```

"/home/corde/mth8408/projet/phase_2/Diom_versus_lbfgs_494_bus.pdf"

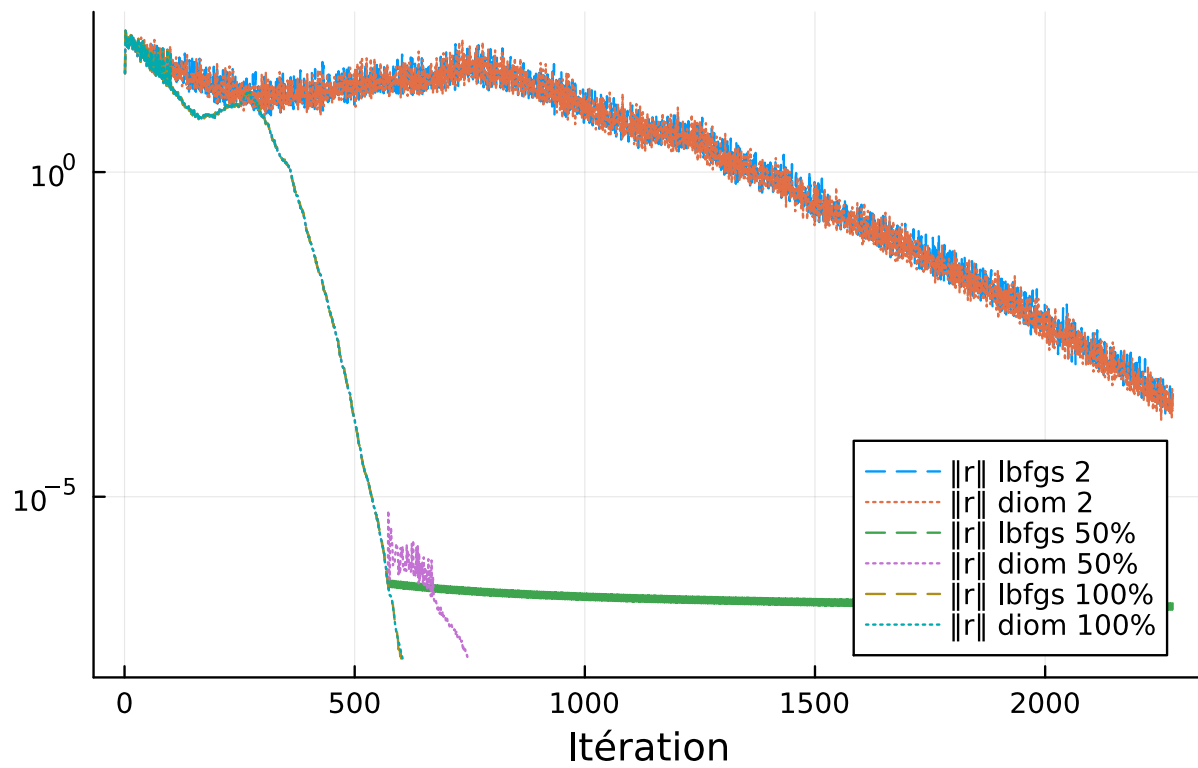


```

A = get_mm("1138_bus")
n,n = size(A)
b = randn(eltype(A), n)
p=2
listmem = memory(n, p)
diom_lbfgs(A, b, "1138_bus", listmem, atol, rtol)

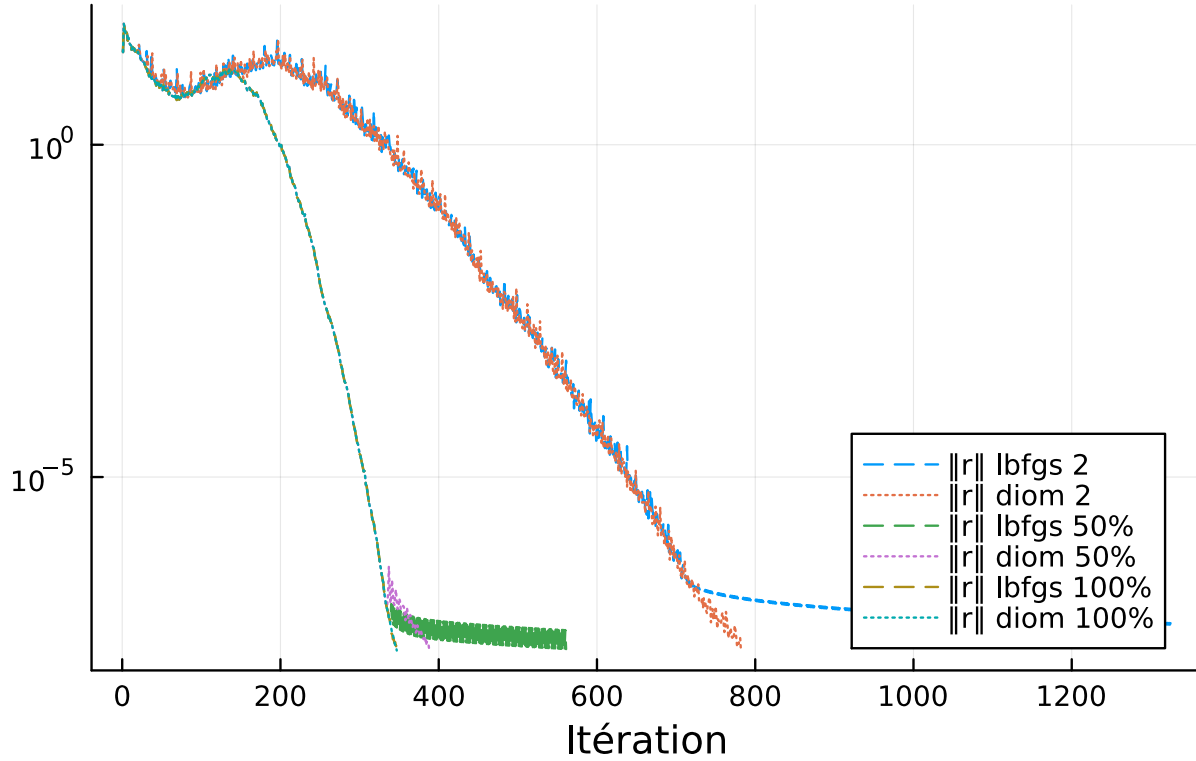
```

"/home/corde/mth8408/projet/phase_2/Diom_versus_lbfgs_1138_bus.pdf"



```
A = get_mm("662_bus")
n,n = size(A)
b = randn(eltype(A), n)
p=2
listmem = memory(n, p)
diom_lbfgs(A, b, "662_bus", listmem, atol, rtol)
```

"/home/corde/mth8408/projet/phase_2/Diom_versus_lbfgs_662_bus.pdf"



Région de confiance

L'utilisation de CG et de L-BFGS est pour résoudre le sous-problème de minimisation du modèle quadratique. Nous nous sommes donc aussi intéressés à l'implémentation de l'algorithme de région de confiance. Pour le moment, nous avons reproduit l'algorithme tel que fournie dans le cahier du GERAD (Bourhis et al., 2019), cependant le code reste fortement inspiré du code `trunk` du module `JSOSolver.jl`. Un possible objectif pour la troisième phase de ce projet serait de directement modifier le solveur `trunk`. À des fins de comparaison, nous avons appelé notre solveur `TRSolver`. Afin de valider notre implémentation, nous l'avons comparé avec le solveur `trunk` ainsi que `Ipopt`. Notre solveur permet d'utiliser comme sous-solveur CG ou bien L-BFGS. Pour ce dernier, il est possible de varier le paramètre de mémoire.

Pour la deuxième phase, nous avons eu comme objectifs de :

- Valider l'implémentation de `TRSolver` avec CG (notre implémentation) contre le solveur `trunk`. Ce sont essentiellement les mêmes algorithmes. Puisque `trunk` est déjà fortement optimisé, si `TRSolver` obtient des performances similaires, cela veut dire que notre implémentation de l'algorithme de région de confiance est efficace.
- Comparer les performances de `TRSolver` avec CG et avec L-BFGS. L'idée est d'identifier si, à ce stade, nous avons des gains en efficacité avec L-BFGS.
- Comparer les performances de `TRSolver` avec L-BFGS pour différents paramètres de mémoire. L'objectif est d'identifier comment le paramètre de mémoire impacte l'efficacité de l'algorithme.

Le code ci-dessous est celui de l'algorithme de région de confiance. Comme mentionné plus tôt, il est fortement inspiré de celui provenant de `JSOSolver`. Notamment, un effort particulier a été apporté pour le rendre compatible avec le module `SolverCore` et `SolverTools`. L'implémentation se base sur celle du cahier du GERAD (Bourhis et al., 2019). Une différence notable est la modification dynamique de la tolérance relative du sous-solveur. Celle-ci suit l'implémentation dans `JSOSolver` afin de s'assurer qu'elle ne soit jamais plus stricte que le solveur de région de confiance.

Voici les modifications les plus importantes par rapport à la phase 1 :

- Compatibilité avec les modules **SolverTools** et **SolverBenchmark** pour faciliter les comparaisons. On peut noter particulièrement l'utilisation de la structure **GenericExecutionStats**, l'ajout du temps écoulé et du statut de convergence.
- Utilisation de l'opérateur hessien du **NLPModel** au lieu de la matrice hessienne pour gagner en efficacité.
- Le réglage dynamique de la tolérance relative du sous-solveur.

```

# TODO: add missing documentation for the functions in this file

using LinearAlgebra, Logging, Printf
using Krylov, LinearOperators, NLPModels, ADNLPModels, SolverTools, SolverCore

include("subsolvers.jl")

import SolverCore.solve!
export solve!

mutable struct TRSolver{
    T,
    V <: AbstractVector{T},
    Op <: AbstractLinearOperator{T},
} <: AbstractOptimizationSolver
    x::V
    gx::V
    Hs::V
    H::Op
    subsolver::Symbol
end

function TRSolver(
    nlp::AbstractNLPModel{T, V};
    subsolver::Symbol = :cg
) where {T, V <: AbstractVector{T}}
    nvar = nlp.meta.nvar
    x = V(undef, nvar)
    gx = V(undef, nvar)
    Hs = V(undef, nvar)
    H = hess_op!(nlp, x, Hs)
    Op = typeof(H)
    subsolver = subsolver

    return TRSolver{T, V, Op}(x, gx, Hs, H, subsolver)
end

function SolverCore.reset!(Solver::TRSolver)
    solver
end

function SolverCore.reset!(Solver::TRSolver, nlp::AbstractNLPModel)
    solver
end

function trsolver(
    nlp::AbstractNLPModel;
    x::V = nlp.meta.x0,
    subsolver::Symbol = :cg,
    kwargs...,

```

```

) where {V}
  solver = TRSolver(nlp; subsolver)
  return solve!(solver, nlp; x = x, kwargs...)
end

function solve!(
  solver::TRSolver{T, V, Op},
  nlp::AbstractNLPModel{T, V},
  stats::GenericExecutionStats{T, V};
  callback = (args...) -> nothing,
  x::V = nlp.meta.x0,
  atol::T = sqrt(eps(T)),
  rtol::T = sqrt(eps(T)),
  sub_rtol = sqrt(eps(T)),
  verbose::Int = 0,
  mu = 0.25,
  eta = 0.01,
  delta_max = Inf,
  max_iter = 3*nlp.meta.nvar,
  max_time = 30.0,
  mem = 1,
  scaling = true,
) where {T, V <: AbstractVector{T}, Op <: AbstractLinearOperator{T}}
  if !(nlp.meta.minimize)
    error("TR Solver only works for minimization problem")
  end
  if !unconstrained(nlp)
    error("TR Solver should only be called for unconstrained problems.")
  end

  # start of algorithm

  SolverCore.reset!(stats)
  start_time = time()
  elapsed_time = 0.0
  set_time!(stats, 0.0)

  n = nlp.meta.nvar

  # solver.x .= x
  # x = solver.x
  # grad = solver.gx
  # H = solver.H
  subsolver = solver.subsolver

  x = nlp.meta.x0
  f = obj(nlp, x)
  g = grad(nlp, x)

  H = hess_op(nlp, x)

  delta = 1.0

```

```

p = similar(x)
b = similar(x)

grad_norm = norm(g)
tolerance = atol + grad_norm * rtol

sub_rtol = max(rtol, min(sqrt(grad_norm), T(0.1)))

iter = 0
set_iter!(stats, iter)
set_objective!(stats, f)
set_dual_residual!(stats, grad_norm)

status = :unknown
set_status!(stats, status)

done = false

while !done

    iter += 1

    # solve quadratic model subject to a trust region constraint
    if subsolver == :cg
        p, sub_stats = Krylov.cg(H, -g, radius=delta, atol=atol, rtol=sub_rtol)
    elseif subsolver == :lbfgs
        p, sub_stats = lbfgs(H, g, delta=delta,
                             atol=atol,
                             rtol=sub_rtol,
                             mem = mem,
                             itmax = 2*n,
                             scaling = scaling)
    end

    b .= H * p      # using hess_op

    f_new = obj(nlp, x + p)
    rho = (f_new - f)/(dot(p, g) + 0.5*(dot(p, b)))

    # bad approximation
    if rho <= mu
        delta *= 0.25

    # very good approximation
    elseif rho > 1 - mu
        delta = min(2.0*delta, delta_max)
    end

    # good approximation -> update current point
    if rho >= eta
        x .+= p
        f = f_new
        g .= grad(nlp, x)
    end
end

```

```

        grad_norm = norm(g)

        H = hess_op(nlp, x)

        sub_rtol = max(rtol, min(sqrt(grad_norm), T(0.1)))
    end

    if grad_norm <= tolerance
        done = true
        status = :first_order
    elseif iter >= max_iter
        done = true
        status = :max_iter
    elseif elapsed_time >= max_time
        done = true
        status = :max_time
    end

    elapsed_time = time() - start_time

    set_iter!(stats, iter)
    set_objective!(stats, f)
    set_dual_residual!(stats, grad_norm)
    set_status!(stats, status)
    set_time!(stats, elapsed_time)

    callback(nlp, solver, stats)
end

set_solution!(stats, x)
stats
end

```

Pour comparer les performances des solveurs, nous utilisons tous les problèmes non contraints de dimensions 100 dans le module `OptimizationProblems`. Les profils de performance sont faits avec le module `OptimizationBenchmark`. Ceci permet d'illustrer la portion de problème résolu par les algorithmes en fonction d'un ratio d'itération ou de temps.

Un problème est dit τ -résolu si la condition suivante est rencontrée :

$$\frac{f^i - f^0}{f^* - f^0} \geq (1 - \tau)$$

Les algorithmes suivants sont testés :

- `trunk` : région de confiance avec CG (implémentation provenant de `JSOSolver`)
- `trSolver_cg` : région de confiance avec CG
- `trsolver_lbfgs` : région de confiance avec `L_BFGS`. Le nombre associé indique la mémoire utilisée.
- `Ipopt` : optimiseur à points intérieurs

Le script ci-dessous permet de lancer les différents solveurs sur les problèmes tests. Les données sont sauvegardées et peuvent être réutilisées plus tard sans relancer les tests.

```

using Pkg
Pkg.activate("projet_env_phase2_3")
Pkg.add("ADNLPMODELS")
Pkg.add("NLPMODELS")
Pkg.add("Krylov")
Pkg.add("LinearOperators")
Pkg.add("JSOSolvers")
Pkg.add("SolverTools")
Pkg.add("SolverCore")
Pkg.add("OptimizationProblems")
Pkg.add("SolverBenchmark")
Pkg.add("NLPMODELSIpopt")
Pkg.add("JLD2")
Pkg.add("Plots")

# TODO: add CUTest

using LinearAlgebra, NLPMODELS, ADNLPMODELS, Printf, LinearOperators, Krylov
using OptimizationProblems, OptimizationProblems.ADNLPProblems, JSOSolvers, SolverTools, SolverCore, SolverBenchmark
using JLD2, Plots

include("TRSolver.jl")

DEBUG = false
EXE_PROBLEMS = false

meta = OptimizationProblems.meta
problem_list = meta[(meta.ncon.==0) && !meta.has_bounds && (meta.nvar.==100), :name]
problems = nothing

problem_to_exe = ["fletcher", "nondquar", "woods", "broydn7d", "sparsine"]

if DEBUG
    problems = (OptimizationProblems.ADNLPProblems.eval(Meta.parse(problem))() for problem in problem_to_exe)
else
    problems = (OptimizationProblems.ADNLPProblems.eval(Meta.parse(problem))() for problem in problem_list)
end

solvers = Dict{
    :ipopt => nlp -> ipopt(nlp, print_level=0),
    :trunk => nlp -> trunk(nlp, verbose=0),
    :trsolver_cg => nlp -> trsolver(nlp, subsolver=:cg, max_time=10.0),
    :trsolver_lbfgs_1 => nlp -> trsolver(nlp, subsolver=:lbfgs, max_time=10.0, mem=1),
    :trsolver_lbfgs_5 => nlp -> trsolver(nlp, subsolver=:lbfgs, max_time=10.0, mem=5),
    :trsolver_lbfgs_10 => nlp -> trsolver(nlp, subsolver=:lbfgs, max_time=10.0, mem=10),
    :trsolver_lbfgs_10_no_scaling => nlp -> trsolver(nlp, subsolver=:lbfgs, max_time=10.0, mem=5, scaling=:none)
}

if EXE_PROBLEMS
    stats = bmark_solvers(solvers, problems)
    @save "stats_opt_problems.jld2" stats
else

```

```

@load "stats_opt_problems.jld2" stats
end

# set default plot dpi
default(dpi=300)

plt_iter = performance_profile(stats, df -> df.iter, tol = 1e-5, xlabel="Iterations ratio")
savefig(plt_iter, "performance_profile_iter.png")

plt_iter = performance_profile(stats, df -> df.neval_obj, tol = 1e-5, xlabel="Objective evaluations ratio")
savefig(plt_iter, "performance_profile_neval.png")

plt_time = performance_profile(stats, df -> df.elapsed_time, tol = 1e-5, xlabel="Elapsed time ratio")
savefig(plt_time, "performance_profile_time.png")

lbfgs_stats = Dict{k => v for (k, v) in stats if occursin("lbfgs", String(k))}
plt_iter = performance_profile(lbfgs_stats, df -> df.elapsed_time, tol = 1e-5, xlabel="Elapsed time ratio")
savefig(plt_iter, "performance_profile_time_lbfgs.png")

```

Nous proposons 3 profils de performance en fonction de plusieurs critères, soit le nombre d'itérations, le nombre d'évaluations de la fonction objective et le temps écoulé. Nous proposons ces 3 critères puisque certains algorithmes peuvent prendre moins d'itérations que d'autres, cependant chaque itération est plus longue à évaluer.

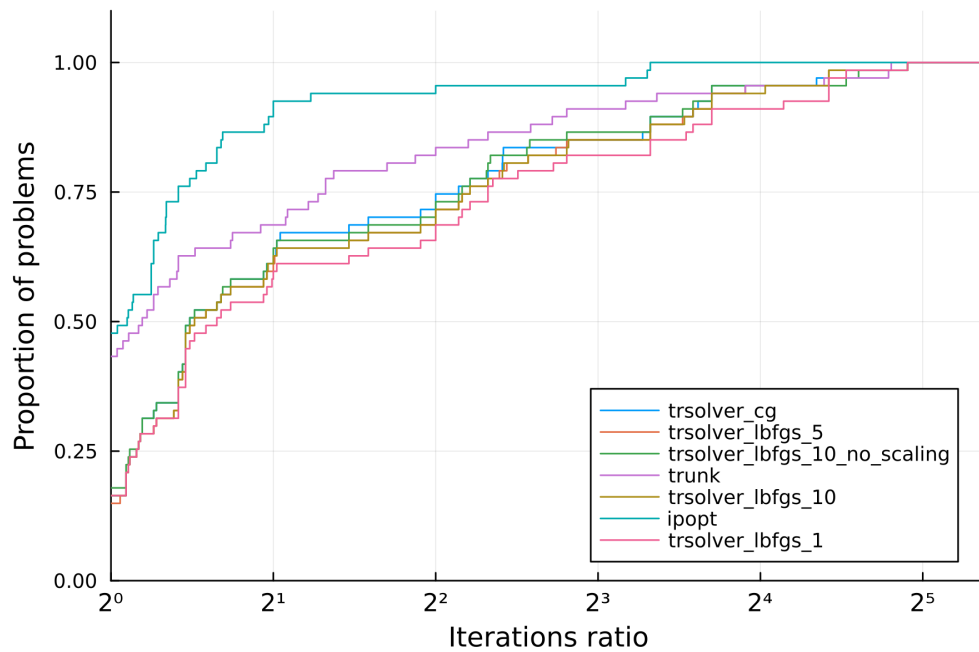


Figure 1: Profile de performance en fonction du nombre d'itérations

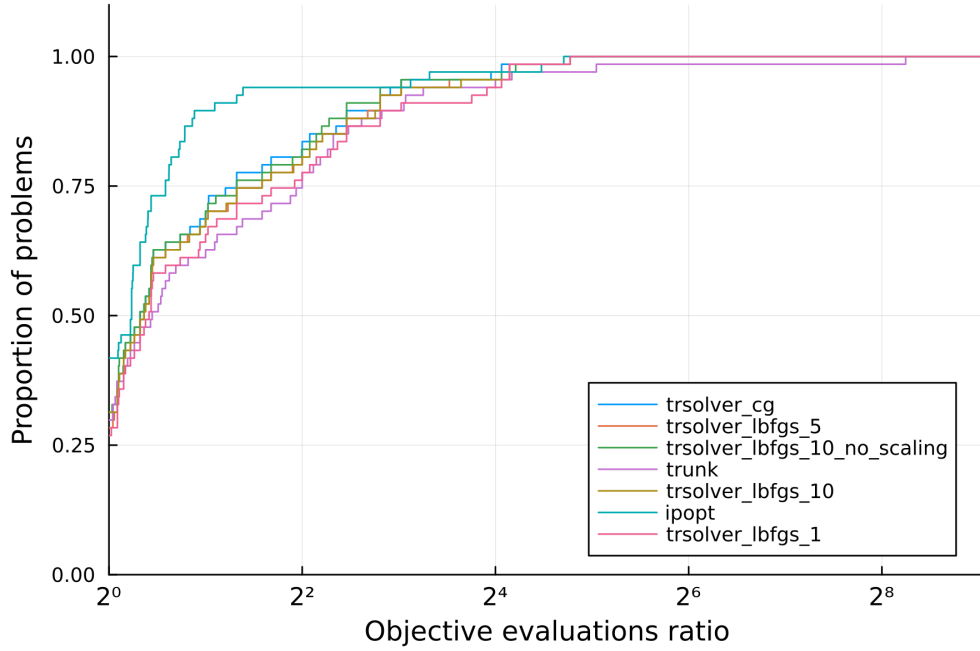


Figure 2: Profile de performance en fonction du nombre d'évaluations de la fonction objective

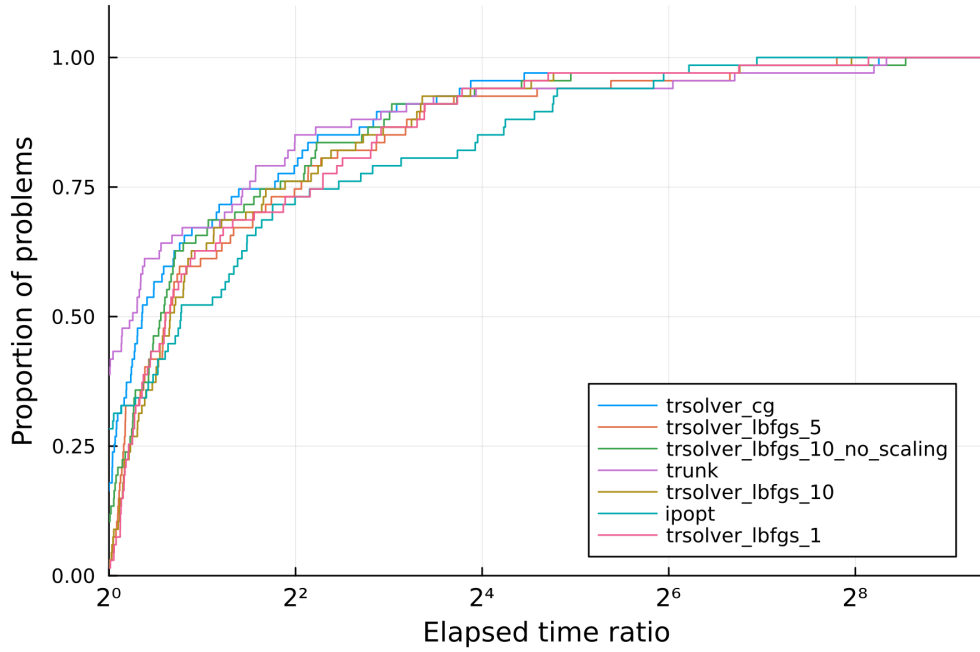


Figure 3: Profile de performance en fonction du temps écoulé

Classement des algorithmes

À la lumière des résultats obtenus, nous pouvons déjà tirer quelques conclusions sur notre implémentation de l'algorithme de région de confiance. Le profile de performance en fonction du temps illustre que :

- **trunk** est l'implémentation qui résout le plus de problèmes rapidement.

- TRSolver performe initialement moins bien, mais rapidement rapidement **trunk**.
- TRSolver avec L-BFGS performe nettement moins bien.
- Ipopt est l'algorithme le moins efficace en termes de temps.

Influence des paramètres de L-BFGS

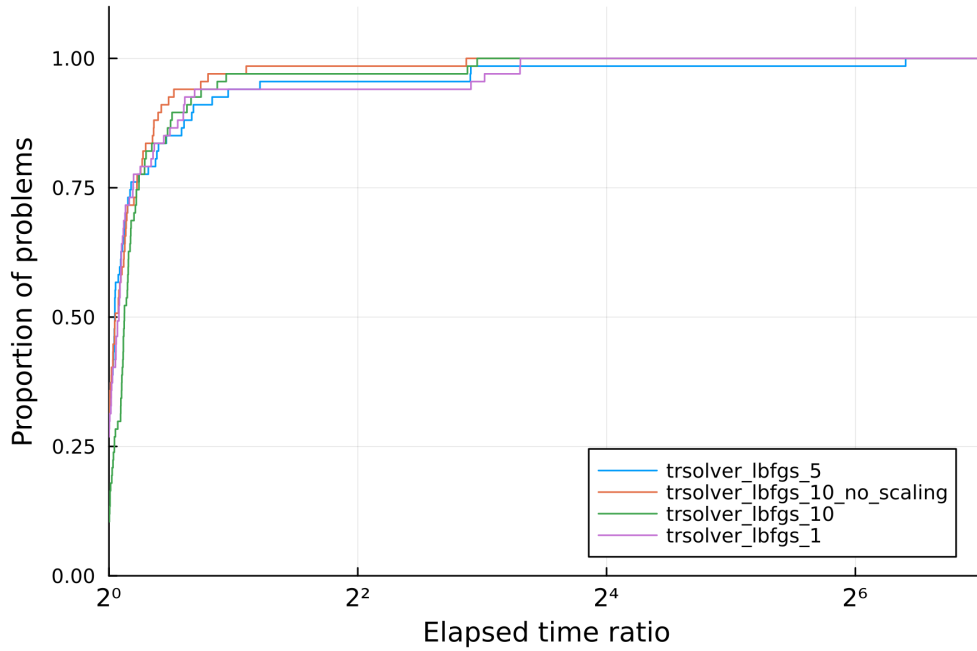


Figure 4: Profile de performance en fonction du temps écoulé pour les sous-solveurs L-BFGS

Le profile de performance ci-haut montre qu'il y a peu de gain en efficacité lorsque la mémoire est augmentée. Cependant, on peut tout de même noter que lorsque la mémoire est réglée à 10 et sans mise à l'échelle, l'optimiseur de région de confiance performe légèrement mieux qu'avec les autres paramètres de mémoire. Il reste donc à voir si avec une implémentation plus efficace de L-BFGS, nous pouvons obtenir de meilleur résultat que TRSolver avec CG. De plus, il serait intéressant de voir s'il est possible d'aller chercher des gains supplémentaires en augmentant davantage la mémoire.

Difficultés rencontrées

- Pour l'implémentation du sous-solveur L-BFGS, nous voulions initialement modifier le code de **trunk**. À la place nous avons jugé qu'il était plus facile de reproduire l'algorithme de région de confiance pour faire des tests initiaux avec L-BFGS. Son utilisation reste compatible avec **SolverCore**.

Résumé de ce qui a été accompli

- Comparaison de la résolution des sous-problèmes convexes avec CG, L-BFGS et DIOM.
- Implémentation de l'algorithme de région de confiance revue
- Comparaison des divers solveurs avec **SolverBenchmark**
- Comparaison de l'effet de la mémoire et de la mise à l'échelle avec le sous-solveur L-BFGS.

Étapes à terminer

- Implémenter dans **Krylov.jl** une version de DIOM adaptée à la résolution de sous-problèmes de région de confiance.

- Analyser la stagnation de LBFGS lorsque la norme du résidu devient très faible (inférieure à 10^{-6}).
- Finaliser une implémentation opérationnelle de la méthode LBFGS.
- Ajouter DIOM comme sous-solveur possible dans l'algorithme de région de confiance TRSolver.
- Comparer les solveurs sur des problèmes de plus grandes dimensions (au minimum 1000 variables).
- *Si les prochains résultats avec L-BFGS sont concluants, il serait intéressant de modifier la fonction `trunk` pour permettre l'utilisation de L-BFGS.*

Annexes

Code L-BFGS avec région de confiance(non optimisé, implémentation à revoir)

```

struct LBFGSStats
    niter::Int
    residuals::Vector{Float64}
    PAP::Matrix{Float64}
end

"""
    lbfgs(Bj, gk, delta; atol=1e-5, rtol=1e-5, mem = 5, max_iter = 10)

Find the minima of a quadratic model with respect to a trust region. This
implementation is based on: https://www.gerad.ca/fr/papers/G-2019-64
"""
function lbfgs(Bj, b; delta = 0.0, atol=1e-26, rtol=1e-26, mem = 2, itmax = 0, scaling = false)

    gk = copy(b)
    dim = length(gk)
    gnormk = gnorm0 = norm(gk)

    k = 1
    pk = zeros(dim)
    residuals = Float64[gnorm0]

    AP_list = Vector{Vector{Float64}}()
    P_list = Vector{Vector{Float64}}()

    if itmax == 0
        itmax = 2*dim
    end
    Hk = InverseLBFGSOperator(dim, mem = mem, scaling = scaling)

    while gnormk > atol + rtol * gnorm0 && k <= itmax

        dk = -Hk*gk
        bk = Bj*dk
        push!(AP_list, bk ./ norm(dk))
        push!(P_list, dk ./ norm(dk))

        # handle negative curvature
        if dot(dk, bk) <= 0
            alphak = -sign(dot(gk, dk))*2*delta/norm(dk)
        else
            alphak = -dot(gk, dk)/dot(dk, bk)
        end
        end
    
```

```

    sk = alphak .* dk
    pk = pk + sk

    if delta > 0.0 && norm(pk) >= delta
        pk -= sk
        # TODO: optimize implementation
        # compute eq (87) such that norm(pk + tau*sk) = delta (see reference)
        tau = (-dot(pk, sk) + sqrt(dot(pk, sk)^2 + dot(sk, sk)*(delta^2 - dot(pk, pk))))/dot(sk, sk)

        return pk + tau .* sk
    end

    yk = alphak .* bk
    gk += yk

    gnormk = norm(gk)
    k += 1

    # update the inverse Hessian approximation
    push!(Hk, sk, yk)
    push!(residuals, gnormk)
end
AP = hcat(AP_list...) # construit la matrice P de taille (dim, k)
P = hcat(P_list...)
PAP = P' * AP
return pk, LBGSSStats(k, residuals, PAP)
end

```