

# Rapport de projet - phase 3

## MTH8408

Oihan Cordelier, Oussama Mouhtal

### Lien GitHub

Ce projet est accessible sur le dépôt GitHub au lien suivant : [https://github.com/oihanc/mth8408\\_projet](https://github.com/oihanc/mth8408_projet).

### Description de 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 vérifier si L-BFGS avec une mémoire supérieure à 1 aiderait 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. Similairement, une autre approche pour résoudre les problèmes des modèles quadratiques est d'utiliser DIOM. Il s'agit d'une approche similaire à CG, cependant en apportant des corrections pour l'orthogonalisation.

### Description des difficultés rencontrées et comment les surmonter

- Efficacité du code : puisque l'objectif de ce projet était de faire une analyse comparative entre les différentes approches, il était nécessaire d'avoir du code efficace, flexible et compatible avec les autres librairies Julia d'optimisation existantes. Pour y parvenir, nous avons d'abord écrit du code « brouillon » pour valider notre compréhension, puis nous nous sommes inspirés des librairies existantes pour écrire des fonctions compatibles avec les outils tels que SolverCore et SolverBenchmark. Finalement, nous avons tenté une autre approche, où nous avons surchargé les fonctions de JSOSolvers et Krylov afin de potentiellement faire des ajouts à ces librairies. De plus, nous avons pris note des retours de la phase 1 et 2, particulièrement pour rendre le code compatible avec le type Float128 et réduire le nombre d'allocations en mémoire.
- Compatibilité entre les librairies : comme mentionné précédemment, pour la phase 3, nous nous sommes essayés à rendre nos codes compatibles avec les librairies existantes. À titre d'exemple, nous avons surchargé les fonctions de JSOSolvers et Krylov pour pouvoir utiliser le solveur trunk (de JSOSolvers) avec notre implémentation L-BFGS avec région de confiance et DIOM avec région de confiance. Cela nous a permis de tester une approche plus similaire à celle utilisée dans Bourhis et al., 2019.

### Description de ce qui a été accompli

Par rapport au cahier du GERAD (Bourhis et al., 2019) :

- Analyse comparative sur des problèmes quadratiques sur base de temps ainsi que la valeur du modèle quadratique pour CG et L-BFGS avec région de confiance.
- Utilisation d'un autre sous-solveur DIOM, modifié pour être compatible avec la contrainte de rayon de confiance.
- Des tests numériques sur des problèmes provenant de la librairie OptimizationProblems.

## Par rapport à la phase 2 :

- Implémentation de DIOM avec contrainte de région de confiance
  - Implémentaiton de L-BFGS revisitée. Une structure workspace est d'abord initialisée avant de lancer le solveur. Cela permet de réduire l'allocation en mémoire.
  - Surcharge de la fonction trunk de JSOSolvers. En plus de CG, il est possible d'utiliser L-BFGS ainsi que DIOM pour résoudre le sous-problème d'optimisation.
  - Tests numériques avec une précision numérique plus élevée en utilisant le type Float128.
- 

## Analyse détaillée - Résolution du sous problème quadratique

### Contexte et notations

On considère le système linéaire  $Ax = b$ .

### Méthode des gradients conjugués (CG)

Hypothèse :  $A$  **HPD** (hermitienne définie positive).

Notations :  $x_k$  itéré,  $r_k = b - Ax_k$  résidu,  $p_k$  direction de recherche.

Mises à jour (Saad, *Iterative Methods for Sparse Linear Systems*, chap. 6) :

$$\begin{aligned}\alpha_k &= \frac{r_k^* r_k}{p_k^* A p_k}, & x_{k+1} &= x_k + \alpha_k p_k, & r_{k+1} &= r_k - \alpha_k A p_k, \\ \beta_{k+1} &= \frac{r_{k+1}^* r_{k+1}}{r_k^* r_k}, & p_{k+1} &= r_{k+1} + \beta_{k+1} p_k.\end{aligned}$$

Les directions  $\{p_k\}$  sont  $A$ -conjuguées :  $p_i^* A p_j = 0$  pour  $i \neq j$ .

### DIOM / IOM (orthogonalisation incomplète)

**IOM(m)** construit une base  $V_k = [v_1, \dots, v_k]$  de  $\mathcal{K}_k(A, r_0)$  via une orthogonalisation **à fenêtre** de taille  $m$  (on orthogonalise  $v_{j+1}$  contre au plus  $m$  prédécesseurs).

**DIOM** est la variante « directe » qui met à jour  $x_k$  en se basant sur la projection  $AV_k = V_k H_k + h_{k+1,k} v_{k+1}$  (Saad, *Iterative Methods for Sparse Linear Systems*, chap. 6).

Avec la factorisation  $H_k = L_k U_k$ , les **directions DIOM** se construisent par

$$p_k^{\text{diom}} = u_{kk}^{-1} \left( v_k - \sum_{i=k-m+1}^{k-1} u_{ik} p_i^{\text{diom}} \right),$$

où  $m$  est la taille de la mémoire et  $u_{i,k}$  les entrées de  $U_k$  (pour  $i \leq 0$ , fixe  $u_{ik} p_i^{\text{diom}} = 0$ ).

La mise à jour des itérées est donnée par :  $x_k = x_{k-1} + \zeta_k p_k^{\text{diom}}$

**Cas HPD.** Lorsque  $(A)$  est HPD et que  $m = 2$ , IOM(2) reproduit **Lanczos** ; DIOM(2) coïncide alors, en arithmétique exacte, avec **FOM** (Galerkin) et donc avec **CG** (mêmes itérés) (Saad, *Iterative Methods for Sparse Linear Systems*, chap. 6).

### Lien DIOM(2) – CG (arithmétique exacte)

Hypothèses : (A) HPD, (m=2).

- **Égalité des itérés** (Saad, *Iterative Methods for Sparse Linear Systems*, chap. 6):

$$x_k^{\text{cg}} = x_k^{\text{diom}} \quad k = 0, 1, \dots, n.$$

- Directions proportionnelles (Saad, *Iterative Methods for Sparse Linear Systems*, chap. 6):

$$\alpha_k = 1/u_{k+1,k+1} \quad \text{et} \quad p_k^{\text{cg}} = \zeta_{k+1} u_{k+1,k+1} p_{k+1}^{\text{diom}}, \quad k = 0, 1, \dots, n \quad (*)$$

Intuition : DIOM et CG imposent la même condition de Galerkin dans l'espace de Krylov engendré par Lanczos ; seules les **normalisations** des directions diffèrent.

---

## L'algorithme diom avec région de confiance

Notre implémentation de diom avec région de confiance est dans le fichier `diom_tr.jl`. Dans cette section, nous expliquerons l'implémentation de diom avec région de confiance en détails et on fera des tests l'implémentation est inspirée de l'implémentation de la région de confiance dans CG.

### Région de confiance avec DIOM

- Dans l'algorithme diom, les itérés sont générés selon

$$x_k = x_{k-1} + \zeta_k p_k^{\text{diom}}.$$

Utiliser  $p_k^{\text{diom}}$  pour chercher une racine  $\sigma$  de  $\|x_{k-1} + \sigma p_k^{\text{diom}}\|^2 = \Delta^2$  est problématique, puisqu'il faut comparer  $\sigma$  à  $\zeta_k$  qui n'a pas potentiellement un signe constant (La problématique dans ce cas qu'elle racine faut choisir la plus grande ou la plus petite ou peut être alterné suivant le signe de  $\zeta_k$ ?). Il est donc préférable pour le moment de passer par la direction de cg, calculée à partir de la relation (\*), ce qui nécessite de stocker un vecteur supplémentaire en mémoire.

- Il faut aussi calculer la norme de cette nouvelle direction (nommée dans l'algorithme pcg), ce qui correspond à une opération additionnelle. Lorsque  $m = 2$  et que la matrice est hermitienne, on peut mettre à jour  $\|p_k^{\text{diom}}\|$  comme dans CG. En revanche, dans le cas général, puisque

$$p_k^{\text{diom}} = \frac{1}{u_{kk}} \left( v_k - \sum_{i=k-m+1}^{k-1} u_{ik} p_i^{\text{diom}} \right),$$

et que les vecteurs  $p_i$  ne sont pas orthogonaux entre eux, la mise à jour de  $\|p_k^{\text{diom}}\|$  nécessite de connaître les produits  $p_i^* p_j$ . Autrement dit, il faudrait disposer de toutes les entrées de la matrice

$$P_k^* P_k = U_k^{-*} U_k^{-1} !!!$$

### Test de la région de confiance dans diom

Les deux premiers tests sont inspirés des tests de cg dans `krylov`.

Le premier test consiste à vérifier que la solution est de norme nulle même si `radius > 0`.

Le deuxième test permet de s'assurer que, lorsque la courbure est négative, la variable `indefinite` est bien mise à jour.

Enfin, le troisième test repose sur la remarque que la façon dont la norme du résidu est calculée ne prend pas en compte que le coefficient  $u_{k,k}$  est écrasé par  $1/\sigma$ .

```
using Test
```

```
function zero_rhs(n :: Int=10; FC=Float64)
    A = rand(FC, n, n)
    b = zeros(FC, n)
    return A, b
end
```

```

for FC in (Float64, ComplexF64)
    @testset "Data Type: $FC" begin
        # Test radius > 0 and b^T A b = 0
        A, b = zero_rhs(FC=FC)
        solver = DiomTRWorkspace(A, b)
        diom!(solver, A, b, radius = 10 * real(one(FC)))
        x, stats = solver.x, solver.stats
        @test stats.status == "x is a zero-residual solution"
        @test norm(x) == zero(FC)
        @test stats.niter == 0

        # Test radius > 0 and p A p < 0
        A = FC[
            10.0 0.0 0.0 0.0;
            0.0 8.0 0.0 0.0;
            0.0 0.0 5.0 0.0;
            0.0 0.0 0.0 -1.0
        ]
        b = FC[1.0, 1.0, 1.0, 0.1]
        solver = DiomTRWorkspace(A, b)
        diom!(solver, A, b; radius = 10 * real(one(FC)))
        x, stats = solver.x, solver.stats
        @test stats.indefinite == true

        # Test residus finale
        A = FC[
            10.0 0.0 0.0 0.0;
            0.0 8.0 0.0 0.0;
            0.0 0.0 5.0 0.0;
            0.0 0.0 0.0 -1.0
        ]
        b = FC[1.0, 1.0, 1.0, 0.1]
        solver = DiomTRWorkspace(A, b)
        diom!(solver, A, b; radius = 10 * real(one(FC)), history=true)
        x, stats = solver.x, solver.stats
        r_alg = stats.residuals[end]
        r_true = norm(b - A * x)
        rtol = sqrt(eps(real(one(FC))))
        @test !isapprox(r_true, r_alg; rtol = rtol)      # Les deux résidu se diffèrent à tolérance machine
    end
end

```

```

Test Summary:      | Pass  Total  Time
Data Type: Float64 |      5      5  0.2s
Test Summary:      | Pass  Total  Time
Data Type: ComplexF64 |      5      5  0.2s

```

## Proposition d'une méthode pour calculer $\|r_k\|$

Supposons qu'à une itération  $k$ , la normalisation de la direction dans diom est faite de la façon suivante:

$$\bar{p}_k^{\text{diom}} = \sigma \left( v_k - \sum_{i=k-m+1}^{k-1} u_{ik} p_i^{\text{diom}} \right).$$

où  $\sigma$  est racine de  $\|x_{k-1} + \sigma p_k^{\text{diom}}\|^2 = \Delta^2$ . Cette notation intermédiaire  $\bar{p}_k^{\text{diom}}$  permet de réutiliser la vraie direction  $p_k^{\text{diom}} = \frac{1}{u_{kk}} \left( v_k - \sum_{i=k-m+1}^{k-1} u_{ik} p_i^{\text{diom}} \right)$  par la suite.

L'itéré est donné par

$$x_{k+1} = x_k + \zeta_k \bar{p}_k^{\text{diom}}.$$

Le résidu  $r_k$  se calcule alors de la façon suivante :

$$r_k = r_{k-1} - \zeta_k A \bar{p}_k^{\text{diom}} = r_{k-1} - \zeta_k u_{k,k} \sigma A p_k^{\text{diom}}.$$

La matrice concaténant les directions de diom satisfait par définition :

$$P_k = V_k U_k^{-1}.$$

En multipliant par  $A$ , on obtient :

$$A P_k = A V_k U_k^{-1}.$$

Or, par la relation de récurrence d'Arnoldi :

$$A V_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^\top.$$

Ainsi :

$$A P_k = (V_k H_k + h_{k+1,k} v_{k+1} e_k^\top) U_k^{-1} = (V_k L_k U_k + h_{k+1,k} v_{k+1} e_k^\top) U_k^{-1} = V_k L_k + h_{k+1,k} v_{k+1} e_k^\top U_k^{-1}.$$

En particulier,

$$A p_k^{\text{diom}} = A P_k e_k = (V_k L_k + h_{k+1,k} v_{k+1} e_k^\top U_k^{-1}) e_k = v_k + \frac{h_{k+1,k}}{u_{kk}} v_{k+1}.$$

## Expression finale du résidu

On en déduit :

$$r_k = r_{k-1} - \zeta_k \sigma u_{k,k} \left( v_k + \frac{h_{k+1,k}}{u_{kk}} v_{k+1} \right).$$

En utilisant la relation (Dans le livre de Saad Chapitre 6)

$$r_{k-1} = -\zeta_{k-1} \frac{h_{k,k-1}}{u_{k-1,k-1}} v_k,$$

on obtient:

$$r_k = \left( -\zeta_{k-1} \frac{h_{k,k-1}}{u_{k-1,k-1}} - \zeta_k \sigma u_{k,k} \right) v_k - \zeta_k \sigma h_{k+1,k} v_{k+1}.$$

finalement utilisant l'orthogonalité partielle des  $v_i$ , on obtient finalement :

$$\|r_k\| = \sqrt{\left| \zeta_{k-1} \frac{h_{k,k-1}}{u_{k-1,k-1}} + \zeta_k \sigma u_{k,k} \right|^2 + \left| \zeta_k \sigma h_{k+1,k} \right|^2}.$$

## Résultats expérimentaux pour des problèmes quadratiques

L-BFGS et DIOM sont comparés avec CG sur 3 problèmes quadratiques construit à partir des matrices `494_bus`, `1138_bus` et `bcsstk16`. Le script ci-dessous permet de comparer CG, L-BFGS et DIOM sur des problèmes mals conditionnés. Les paramètres de mémoire sont variés pour illustrer leur influence.

```

using Pkg
Pkg.activate("projet_env")
# 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")

# Pkg.add("SuiteSparseMatrixCollection")
# Pkg.add("MatrixMarket")

# Pkg.add("SparseArrays")

using LinearAlgebra, NLPModels, ADNLPModels, Printf, LinearOperators, Krylov
using OptimizationProblems, OptimizationProblems.ADNLPProblems, JSOSolvers, SolverTools, SolverCore, So
using JLD2, Plots

using SuiteSparseMatrixCollection, MatrixMarket

using Profile

using SparseArrays

include("TrunkSolverUpdate.jl")

"""
    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")
    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

# compute quadratic objective
function compute_obj(x, A, g)
    return dot(g, x) + 0.5 * dot(x, A*x)
end

function elapsed_time_comparison(A, b, name, listmem, atol, rtol; compute_obj_flag=false, itmax=10*length(b))
    n = length(b)
    x0 = zeros(n)
    println("INIT obj= ", dot(b, x0) + 0.5*dot(x0, A*x0))

    plt = plot(layout = (4, 2), size=(1200, 1200))

    elapsed_time = Float64[0.0]
    cg_obj = Vector{Float64}()
    start_time = time()

    obj = Float64[compute_obj(x0, A, b)]

    function timing(workspace)
        push!(elapsed_time, time() - start_time)
        if compute_obj_flag
            push!(obj, compute_obj(workspace.x, A, b))
        end

        return false
    end

    (xcg, statscg) = cg(A, -b; atol=atol, rtol=rtol, callback=timing, history=true, itmax=itmax)
    println("elapsed time: ", time() - start_time)
    println("CG obj= ", dot(b, xcg) + 0.5*dot(xcg, A*xcg))

    gr()
    # plot for LBFGS comparison
    plot!(plt[1, 1], statscg.residuals, label="cg", title="CG vs L-BFGS", lw=1, yaxis=:log, linestyle = :solid)
    plot!(plt[3, 1], elapsed_time*1.0e3, statscg.residuals, label="||r|| cg ", lw=1, yaxis=:log, linestyle = :solid)

    # plot for DIOM comparison
    plot!(plt[1, 2], statscg.residuals, label="cg", title="CG vs DIOM", lw=1, xlabel="Iterations", ylabel="Residuals")
    plot!(plt[3, 2], elapsed_time*1.0e3, statscg.residuals, label="||r|| cg", lw=1, xlabel="Elapsed time (ms)", ylabel="Elapsed time (ms)")

    if compute_obj_flag
        # objective plotted against iterations
        plot!(plt[2, 1], obj, label="cg", lw=1, xlabel="Iterations", ylabel="Objective Value", legend = :top)
        plot!(plt[2, 2], obj, label="cg", lw=1, xlabel="Iterations", ylabel="Objective Value", legend = :top)

        # objective plotted against elapsed time
        plot!(plt[4, 1], elapsed_time*1.0e3, obj, label="cg", lw=1, xlabel="Elapsed time (ms)", ylabel="Objective Value", legend = :top)
    end
end

```

```

    plot!(plt[4, 2], elapsed_time*1.0e3, obj, label="cg", lw=1, xlabel="Elapsed time (ms)", ylabel="Obj")
end

# ----- L-BFGS -----
for mem in listmem

    # mem_percent = round(mem/length(b)*100, digits=3)

    lbfgs_elapsed_time = Vector{Float64}()
    lbfgs_residual = Vector{Float64}()
    lbfgs_obj = Vector{Float64}()
    start_time = time()

    function lbfgs_logger(ws)
        push!(lbfgs_elapsed_time, time() - start_time)
        push!(lbfgs_residual, norm(ws.gk))
        push!(lbfgs_obj, compute_obj(ws.pk, A, b))
    end

    start_time = time()

    (xlbfgs, statslbfgs) = TrunkSolverUpdate.lbfgs_tr(A, b; mem=mem, atol=atol, rtol=rtol, callback=lbfgs_logger)
    # println("elapsed time: ", time() - start_time)
    # println("LBFGS obj= ", dot(b, xlbfgs) + 0.5*dot(xlbfgs, A*xlbfgs))

    plot!(plt[1, 1], lbfgs_residual, label="lbfgs $(m = mem)", lw=1, linestyle = :dash)
    plot!(plt[3, 1], lbfgs_elapsed_time*1.0e3, lbfgs_residual, label="lbfgs $(m = mem)", lw=1, linestyle = :dash)

    if compute_obj_flag
        plot!(plt[2, 1], lbfgs_obj, label="lbfgs $(m = mem)", lw=1, legend = :topright)
        plot!(plt[4, 1], lbfgs_elapsed_time*1.0e3, lbfgs_obj, label="lbfgs $(m = mem)", lw=1, legend = :topright)
    end
end

# ----- DIOM -----
for mem in listmem

    # mem_percent = round(mem/length(b)*100)

    obj = Float64[compute_obj(x0, A, b)]

    elapsed_time = Float64[0.0]

    start_time = time()

    (xdiom, statsdiom) = diom(A, -b; memory = mem, atol=atol, rtol=rtol, callback=timing, history=true)

    plot!(plt[1, 2], statsdiom.residuals, label="diom $(m = mem)", lw=1, linestyle = :dash)
    plot!(plt[3, 2], elapsed_time*1.0e3, statsdiom.residuals, label="diom $(m = mem)", lw=1, linestyle = :dash)

    if compute_obj_flag
        plot!(plt[2, 2], obj, label="diom $(m = mem)", lw=1, legend = :topright)
        plot!(plt[4, 2], elapsed_time*1.0e3, obj, label="diom $(m = mem)", lw=1, legend = :topright)
    end
end

```

```

    end

    end

    savefig("subsolver_comparison/cg_lbfgs_diom_comparison_${(name)}.png")
end

T = Float64

matrices = ["494_bus", "1138_bus", "bcsstk16"]

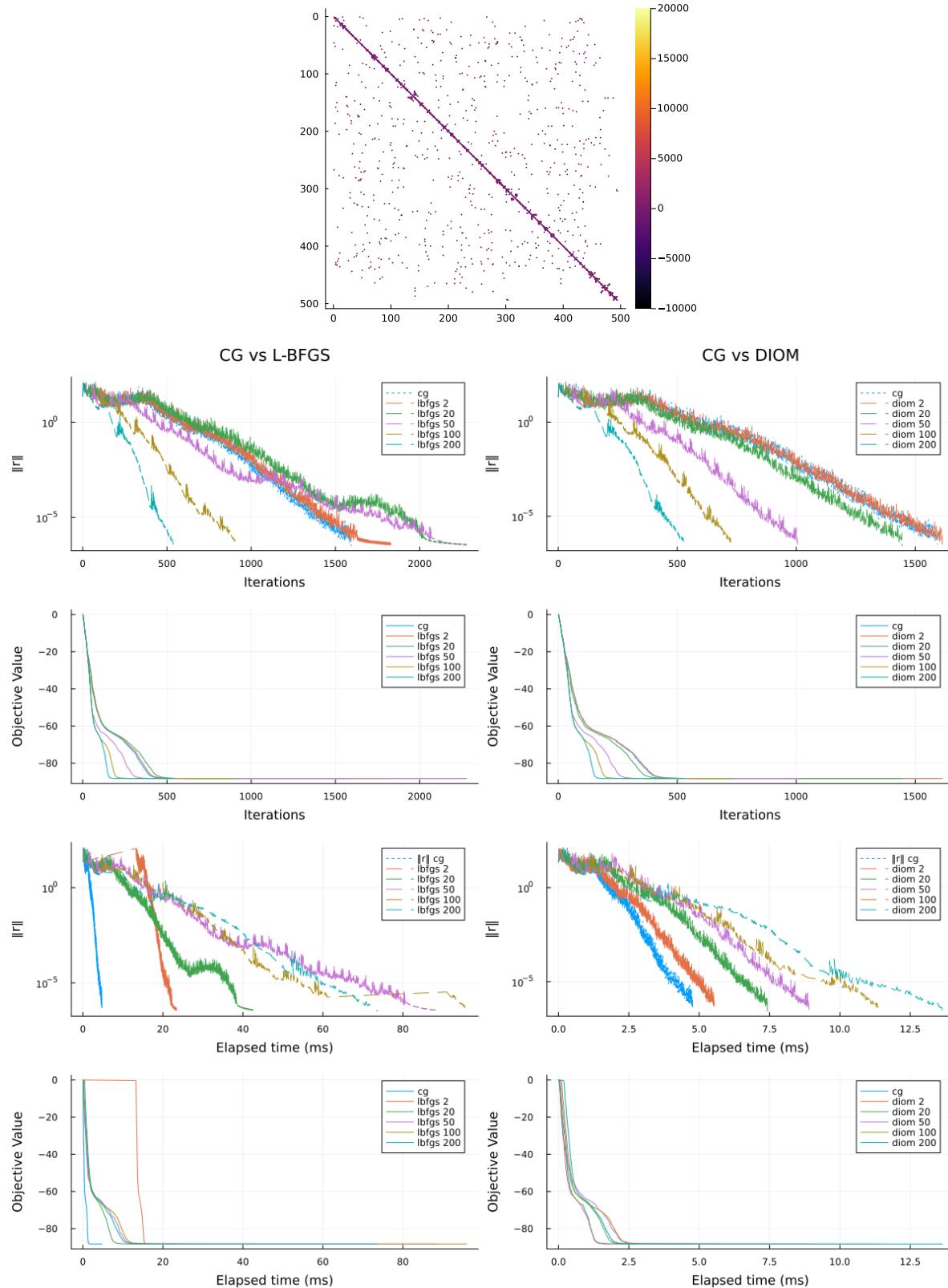
for matrix in matrices
    # display matrix
    A = get_mm(matrix)
    display(spy(A))

    # benchmark matrix on CG, L-BFGS and DIOM
    n,n = size(A)
    b = randn(eltype(A), n)
    atol = √eps(T)
    rtol = √eps(T)
    p=4
    listmem = memory(n, p)
    # println("listmem= ", listmem)
    listmem = [2, 20, 50, 100, 200]
    elapsed_time_comparison(A, b, matrix, listmem, atol, rtol, compute_obj_flag=true, itmax=5*n)
end

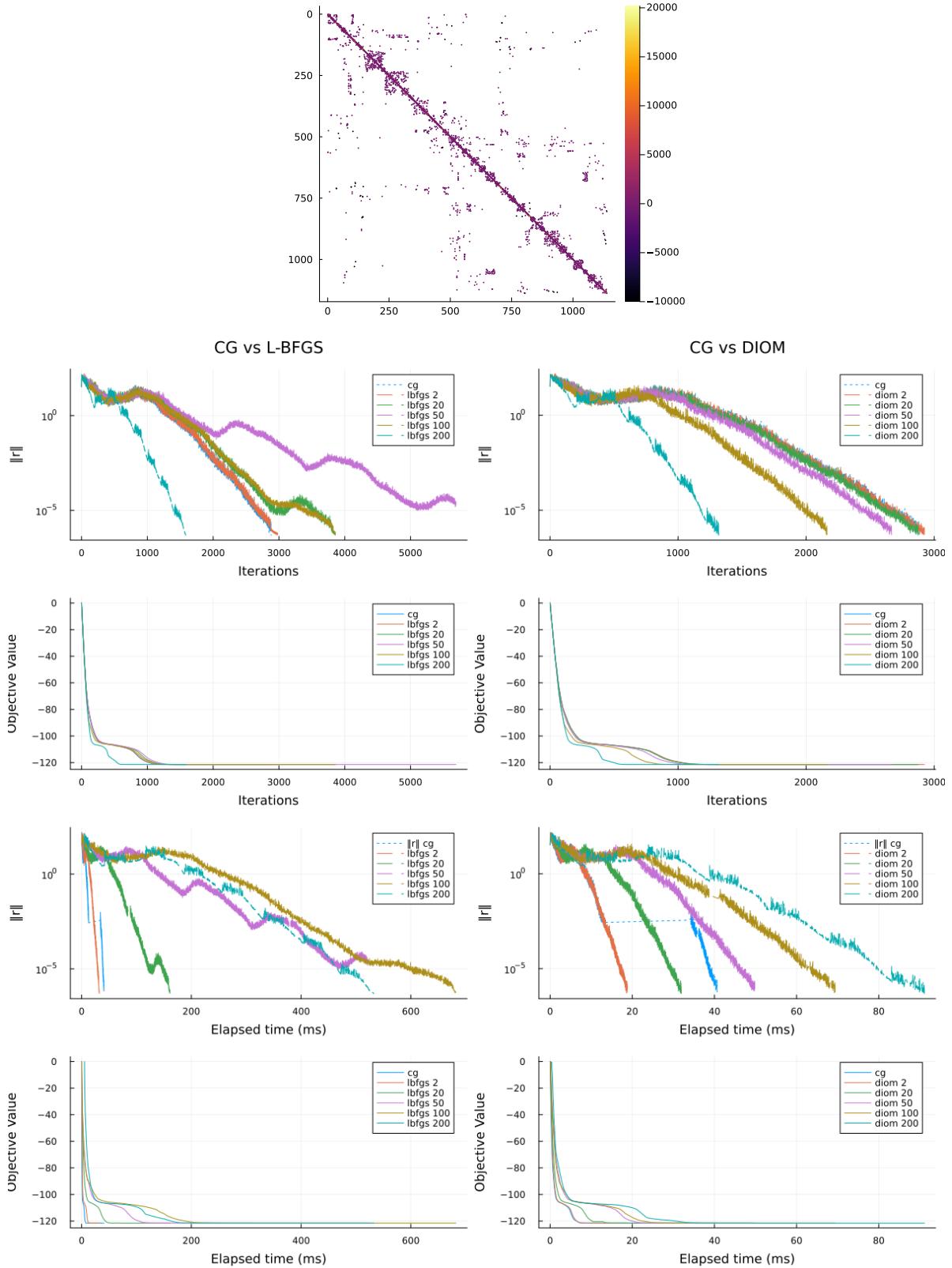
```

La norme du résidu et la valeur de l'objectif sont traçées par rapport au nombre d'itérations et le temps écoulé. En augmentant la mémoire, L-BFGS et DIOM converge en moins d'itération, cependant chaque itération devient plus couteuse. De plus, DIOM semble être plus efficace que L-BFGS.

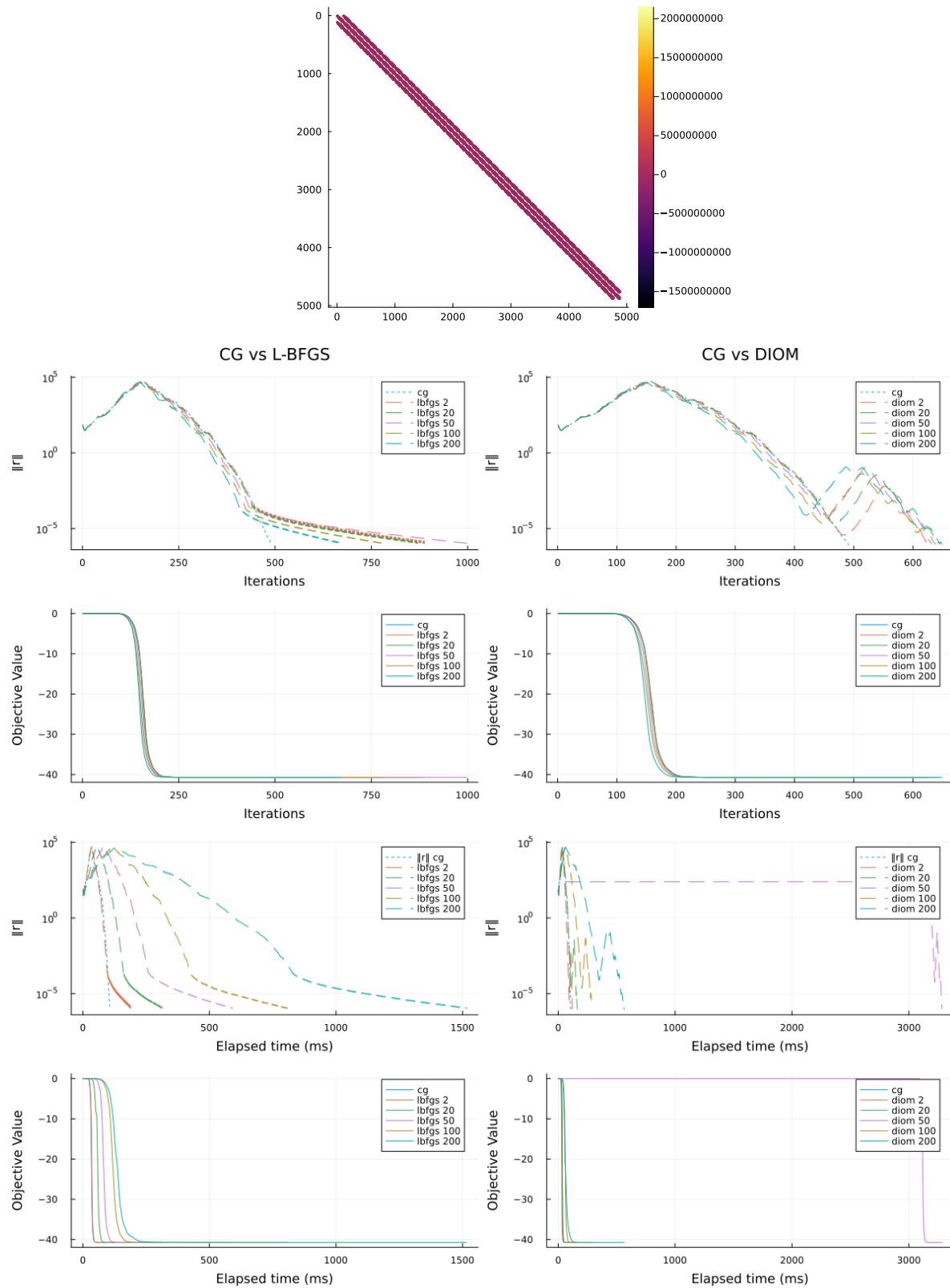
## Matrice : 494\_bus



## Matrice : 1138\_bus



## Matrice : bcsstk16



## Analyse détaillée - Optimisation sous région de confiance

L'objectif derrière cette analyse est d'évaluer la pertinence d'utiliser L-BFGS et DIOM pour résoudre les sous-problèmes des modèles quadratiques des méthodes d'optimisation quadratiques. Les approches ont d'abord été testé sur les 5 fonctions tests initialement proposées dans le cahier du GERAD (Bourhis et al., 2019).

50 problèmes appartenant à OptimizationProblems et dont les dimensions sont variables ont été sélectionnés. De plus, les analyses ont été faites en faisant varier le paramètre de mémoire de L-BFGS et DIOM et faisant varier la précision de l'arithmétique flottant (Float32, Float64 et Float128). À noter que pour certains problèmes, la dimension du problème est contrainte par un multiple. Dans ces cas-là, la dimension des problèmes est minimalement réduite pour respecter la contrainte. La tolérance absolue et relative sont toujours fixées à la racine carrée de l'epsilon machine du type du nombre à virgule flottante.

Le script ci-dessous permet d'exécuter les tests pour différents solveurs, ainsi que différents type de *Float*. Les résultats sont enregistrés dans un fichier .csv pour de futurs analyses.

```
using Pkg
Pkg.activate("projet_env")

# Pkg.add("DataFrames")
# Pkg.add("CSV")

using Random

using DataFrames, CSV

using OptimizationProblems, OptimizationProblems.ADNLPProblems
using NLPModels, ADNLPModels

using Quadmath
using JSOSolvers

using LinearAlgebra

include("TrunkSolverUpdate.jl")
include("TRSolverModule.jl")

DEBUG = false    # if set to true -> only 5 test problems

nvars = 500

result_file = "benchmark_n500_float32.csv"

meta = OptimizationProblems.meta

if DEBUG
    problem_list = ["fletcher", "nondquar", "woods", "broydn7d", "sparsine"]
else
    problem_list = meta[(meta.variable_nvar.==true).&(meta.ncon.==0).&.!meta.has_bounds.&(meta.minimize
end

if !isfile(result_file)
    CSV.write(result_file, DataFrame(
        problem=String[],
```

```

    nvar=Int16[] ,
    precision=Float64[] ,
    solver=String[] ,
    status=String[] ,
    objective=Float64[] ,
    dual feas=Float64[] ,
    iter=Int16[] ,
    elapsed_time=Float64[] ,
    neval_obj=Int16[] ,
    neval_grad=Int16[] ,
    neval_hprod=Int16[] ,
)
)
end

problem_list = problem_list[1:50]

precisions = [Float32, Float64, Float128]

max_time = 30.0

solvers = Dict(
:trunk_cg => (nlp; kwargs...) -> trunk(nlp; kwargs...),

:trunk_lbfsgs_2 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=2, kwargs...),
:trunk_lbfsgs_5 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=5, kwargs...),
:trunk_lbfsgs_10 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=10, kwargs...),
:trunk_lbfsgs_20 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=20, kwargs...),
:trunk_lbfsgs_50 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=50, kwargs...),
:trunk_lbfsgs_100 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:lbfsgs, mem=100, kwargs...),

:trunk_diom_2 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=2, kwargs...),
:trunk_diom_5 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=5, kwargs...),
:trunk_diom_10 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=10, kwargs...),
:trunk_diom_20 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=20, kwargs...),
:trunk_diom_50 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=50, kwargs...),
:trunk_diom_100 => (nlp; kwargs...) -> TrunkSolverUpdate.trunk(nlp; subsolver=:diom, mem=100, kwargs...),

# :trsolver_lbfsgs_2 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:lbfsgs, mem=2, kwargs...),
# :trsolver_lbfsgs_50 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:lbfsgs, mem=50, kwargs...),
# :trsolver_lbfsgs_100 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:lbfsgs, mem=100, kwargs...),
# :trsolver_lbfsgs_500 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:lbfsgs, mem=500, kwargs...),

# :trsolver_diom_2 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:diom, mem=2, kwargs...),
# :trsolver_diom_5 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:diom, mem=5, kwargs...),
# :trsolver_diom_50 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:diom, mem=50, kwargs...),
# :trsolver_diom_100 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:diom, mem=100, kwargs...),
# :trsolver_diom_500 => (nlp; fixed_sub_rtol=true, kwargs...) -> TRSolverModule.trsolver(nlp; subsolver=:diom, mem=500, kwargs...),
)

# df = CSV.read("benchmark_n100_float128.csv", DataFrame)
# problem_list = unique(df.problem)

```

```

# df = CSV.read(result_file, DataFrame)
# problems_in_file = unique(df.problem)

# println(length(problem_list))
# problem_list = filter(p -> !(p in problems_in_file), problem_list)
# println(length(problem_list))

# problems = (OptimizationProblems.ADNLPPProblems.eval(Meta.parse(problem))(n=nvars, type=Float32) for p in problem_list)

for float_type in precisions
    println("===== Float type= ", float_type, " =====")

    problems = (OptimizationProblems.ADNLPPProblems.eval(Meta.parse(problem))(n=nvars, type=float_type) for p in problem_list)

    for nlp in problems

        x0 = copy(nlp.meta.x0)

        shuffled_solvers = shuffle(collect(solvers))

        for (sol_name, sol) in shuffled_solvers

            if norm(x0 .- nlp.meta.x0) > √(eps(float_type))
                throw("NLPMModel.meta.x0 was changed. Review solver implementation!")
            end

            reset!(nlp)

            println("solver= ", sol_name)

            try
                stats = sol(nlp; max_time=max_time, verbose=0)

                row = DataFrame(
                    problem = [nlp.meta.name],
                    nvar = [nlp.meta.nvar],
                    precision = [float_type],
                    solver = [String(sol_name)],
                    status = [string(stats.status)],
                    objective = [stats.objective],
                    dual feas = [stats.dual_feas],
                    iter = [stats.iter],
                    elapsed_time = [stats.elapsed_time],
                    neval_obj = [neval_obj(nlp)],
                    neval_grad = [neval_grad(nlp)],
                    neval_hprod = [neval_hprod(nlp)],
                )
            )

                CSV.write(result_file, row; append=true)

            catch e
                println("For problem= ", nlp.meta.name)
            end
        end
    end
end

```

```

        if isa(e, MethodError)
            println("caught a MethodError: $e")
        elseif isa(e, ArgumentError)
            println("caught an ArgumentError: $e")
        else
            println("caught an unexpected error: $e")
        end
    end

    end
end
end

```

À partir du fichier .csv, il est possible de tracer des profiles de performances. Un problème est dit résolu si son statut de convergence est :first\_order. La portion de problèmes résolus est comparé contre le nombre d'itération de la boucle extérieure, le temps requis ainsi que le nombre d'opération matricielle sur la hessienne. Le script utilisé pour tracer les profiles de performance est le suivant :

```

using Pkg
Pkg.activate("projet_env")

using DataFrames, CSV, Plots

"""
    performance_profiles(filename::String;
                          nvar::Union{Nothing, Int}=nothing,
                          precision::Union{Nothing, String}=nothing,
                          solvers::Union{Nothing, Vector{String}}=nothing)

create performance profiles for a given CSV results file.

# arguments
- `filename::String`: path to CSV file with benchmark results
- `nvar::Int`: filter problems by the number of variables
- `precision::String`: filter by the precision string (e.g. "Float64", "Float128")
"""

function performance_profiles(filename::String;
                               nvar::Union{Nothing, Int}=nothing,
                               precision::Union{Nothing, String}=nothing,
                               solvers::Union{Nothing, Vector{String}}=nothing)

    df = CSV.read(filename, DataFrame)

    if nvar !== nothing && :nvar in Symbol.(names(df))
        df = subset(df, :nvar => ByRow==(nvar)))
    end

    if precision !== nothing && :precision in Symbol.(names(df))
        df = subset(df, :precision => ByRow==(precision)))
    end

    if solvers !== nothing && :solver in Symbol.(names(df))
        df = subset(df, :solver => ByRow(in(solvers))))
    end

```

```

# Collect solvers and problems
solvers = unique(df.solver)
problems = unique(df.problem)

metrics = [:iter, :elapsed_time, :neval_hprod]
titles = ["Iterations", "Elapsed time (s)", "Neval hprod"]

plt = plot(layout=(1,3), size=(1500, 500), legend=:bottomright, framestyle=:box, margin=10Plots.mm)

for (j, metric) in enumerate(metrics)

    println("metric= ", metric)

    # ratios per solver
    ratios = Dict(s => Float64[] for s in solvers)

    for prob in problems
        sub = df[df.problem .== prob, :]

        # select only converged runs
        solved = subset(sub, :status => ByRow==(("first_order")))

        # if all solvers failed to solve a problem
        if nrow(solved) == 0 || !(metric in Symbol.(names(solved)))
            for s in solvers
                push!(ratios[s], Inf)
            end
            continue
        end

        best = minimum(solved[:, metric])

        for s in solvers
            row = filter(:solver => ==(s), solved)
            if nrow(row) == 0
                push!(ratios[s], Inf)
            else
                push!(ratios[s], row[1, metric] / best)
            end
        end
    end

    # extend each line to the maximum ratio
    allvals = vcat(values(ratios)...)

    finite_vals = filter(isfinite, allvals)
    ratio_max = 1.05 * maximum(finite_vals)

    for (solver, rvals) in ratios
        finite_rvals = sort(filter(isfinite, rvals))
        portion = (1:length(finite_rvals)) ./ length(problems)

        # extend the step horizontally to ratio_max
        if !isempty(finite_rvals)

```

```

        xvals = vcat(finite_rvals, ratio_max)
        yvals = vcat(portion, portion[end])
    else
        # all failures, plot a horizontal line at 0
        xvals = [1.0, ratio_max]
        yvals = [0.0, 0.0]
    end

    plot!(plt[j], xvals, yvals,
          linetype=:steppost, label=solver,
          xlabel=titles[j], xaxis=:log, ylabel="Portion solved")
end
end

display(plt)
return plt
end

# solvers = [
#     "trunk_cg",
#     "trunk_lbfsgs_2",
#     "trunk_lbfsgs_50",
#     "trunk_lbfsgs_100",
# ]
# solvers = [
#     "trunk_cg",
#     "trunk_diom_2",
#     "trunk_diom_5",
#     "trunk_diom_50",
#     "trunk_diom_100",
# ]
solvers = [
    "trunk_cg",
    "trunk_lbfsgs_100",
    "trunk_diom_100",
]
]

performance_profiles("benchmark_n500_float64.csv"; nvar=500, precision="Float64", solvers=solvers)

```

## Résultats

$N = 500 / \text{Float32}$

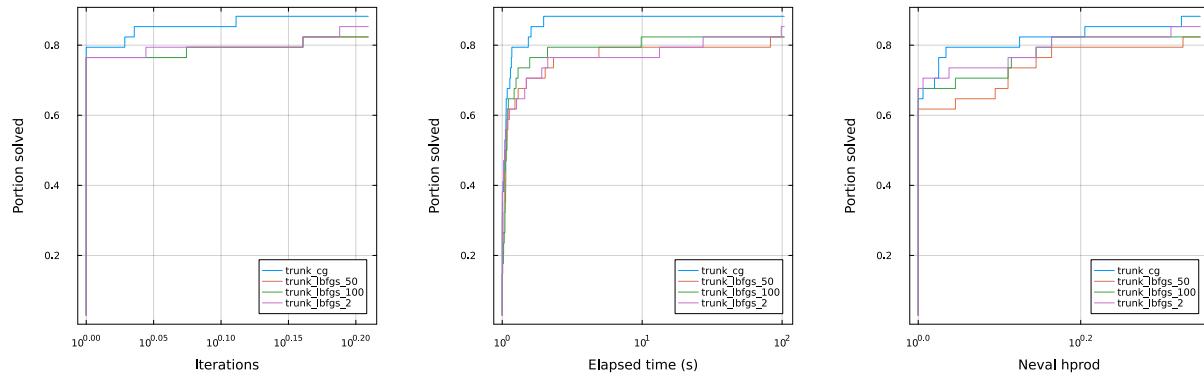


Figure 1: Comparaison entre CG et L-BFGS

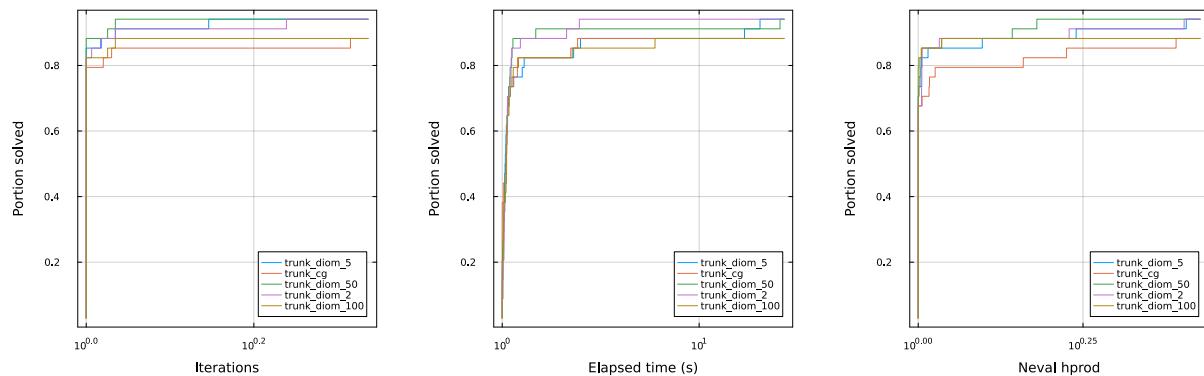


Figure 2: Comparaison entre CG et DIOM

**N = 500 / Float64**

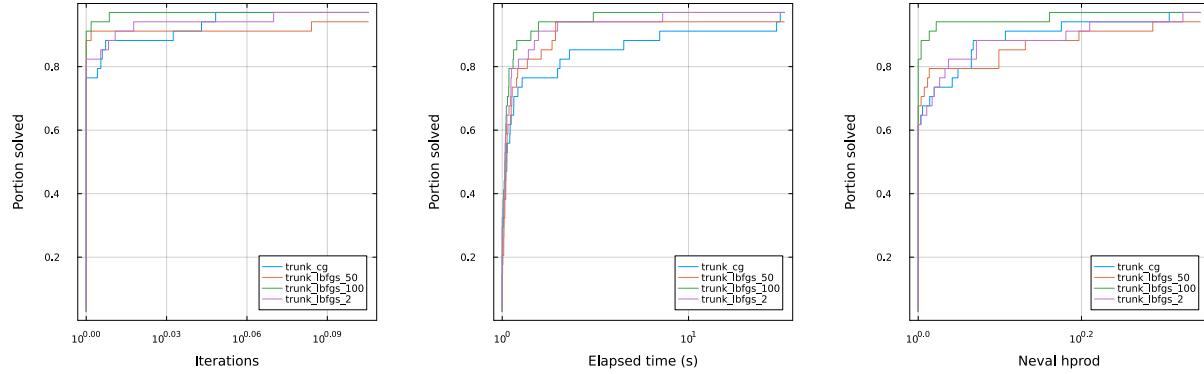


Figure 3: Comparaison entre CG et L-BFGS

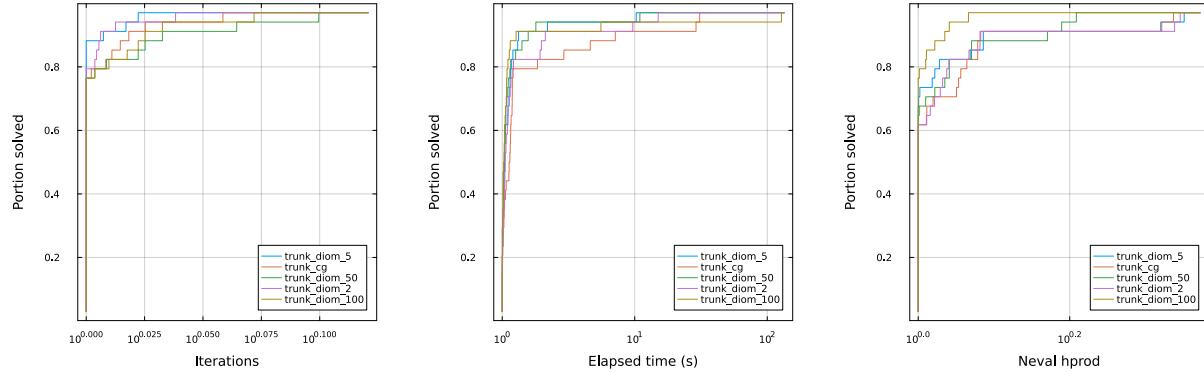


Figure 4: Comparaison entre CG et DIOM

Les résultats sont particulièrement bons pour les problèmes de dimension  $n = 500$  et en utilisant les `Float64`. La figure suivante compare la meilleure configuration de L-BFGS avec DIOM tout en gardant CG comme référence.

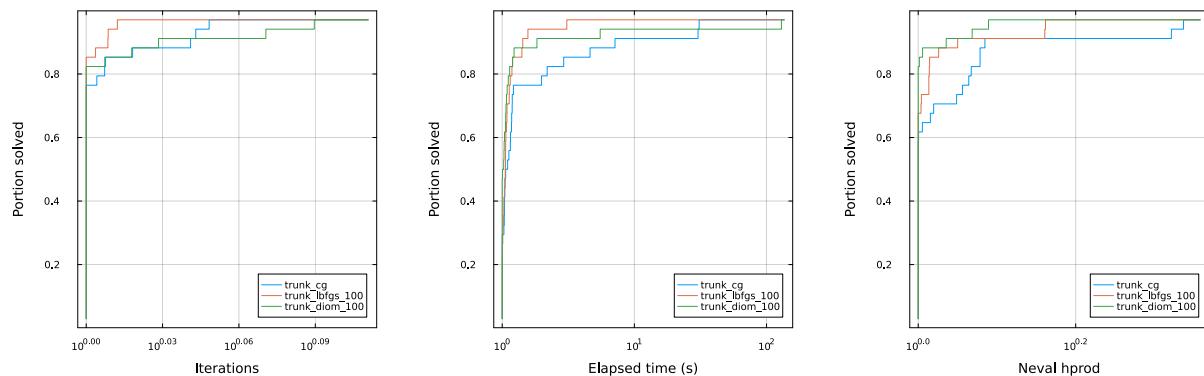


Figure 5: Comparaison entre CG, L-BFGS et DIOM, avec  $n = 500$  et `Float64`

**N = 100 / Float128**

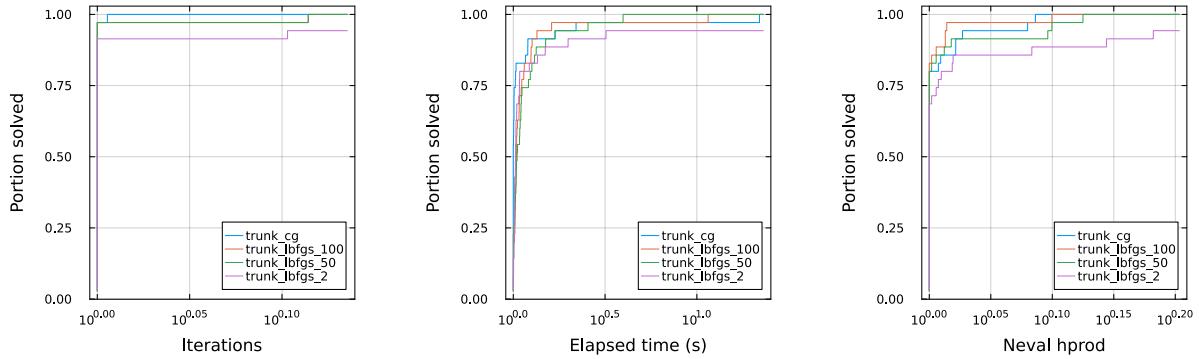


Figure 6: Comparaison entre CG et L-BFGS

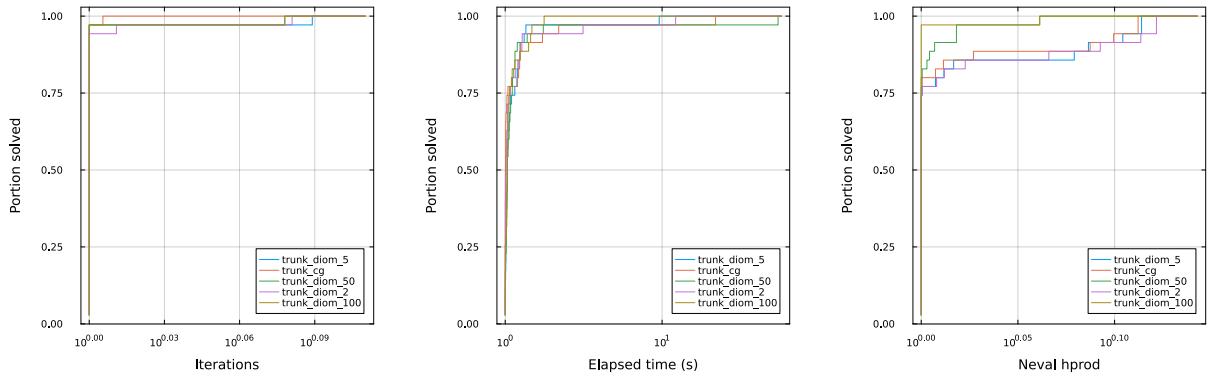


Figure 7: Comparaison entre CG et DIOM

## 5 problèmes du cahier du GERAD (Bourhis et al., 2019)

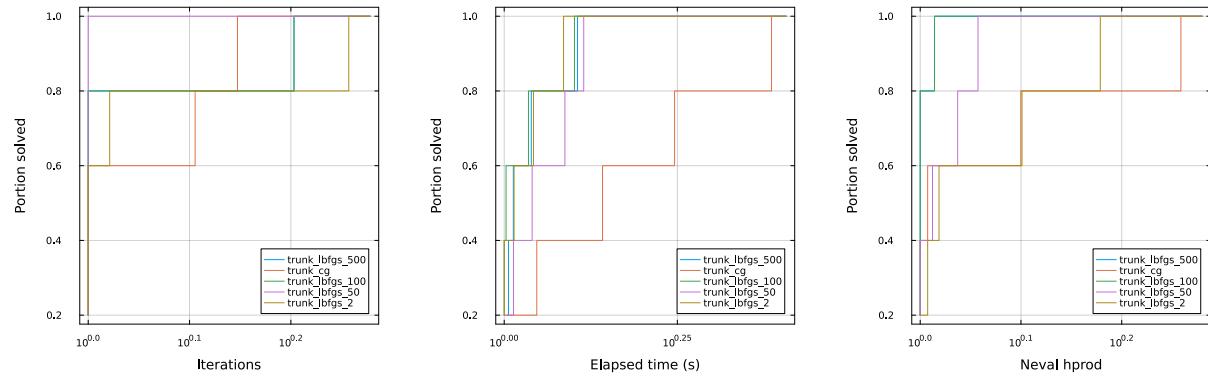


Figure 8: Comparaison entre CG et L-BFGS, avec N = 100 et Float128

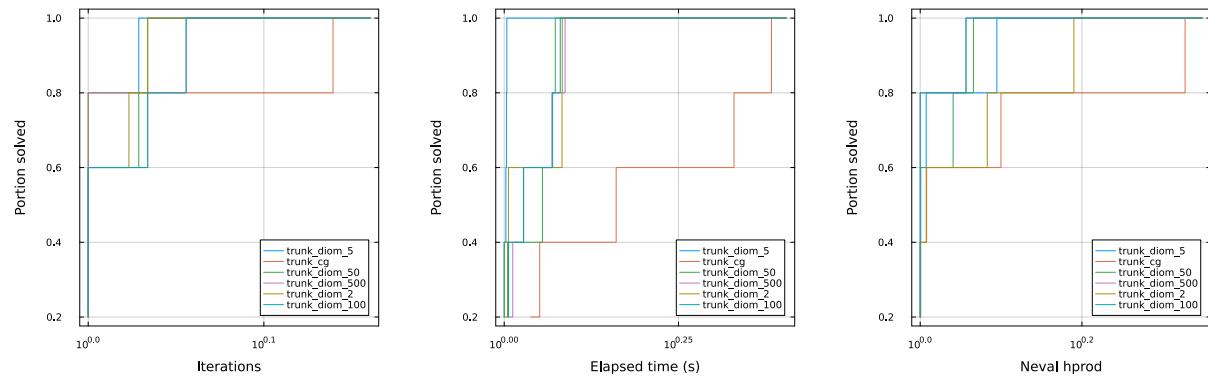


Figure 9: Comparaison entre CG et DIOM, avec N = 100 et Float128

## Notre perspective sur le projet et interprétation

Les tests de la première partie démontrent qu'augmenter le paramètre de mémoire permet de réduire le nombre d'itérations requit pour résoudre le sous-problème quadratique. Les sous-solveurs quadratiques font un produit matriciel avec la hessienne par itération. Donc si le nombre d'itération est réduit, le nombre de produit matriciel devrait aussi l'être. Réduire le nombre d'opération matriciel devient particulièrement intéressant lorsque celui-ci est coûteux, par exemple lorsque la précision est élevée (Float128).

Concernant les résultats de la deuxième partie, nous constater les éléments suivants : pour les problèmes qui se résoud en quelques itérations, le coût d'utiliser L-BFGS ou DIOM n'est pas justifié. Cependant, pour les problèmes qui prend de nombreuses itérations, il est possible de constater que L-BFGS et DIOM permettent de réduire le nombre de produit avec la hessienne, ce qui pourrait justifier la réduction de temps pour certaines des instances. À titre d'exemple, les 5 problèmes tests proposés dans le cahier du GERAD (Bourhis et al., 2019) font de nombreux produits avec les hessiennes, ce qui expliquerait qu'il y ait une réduction de temps. Le tableau ci-dessous liste les statistiques sur le problème *nondquar*. Il est possible de constater que le nombre de produits avec la hessienne est considérablement réduit lorsque la mémoire est augmentée.

```
df = CSV.read("benchmark_original.csv", DataFrame)
df = subset(df, :precision => ByRow==(("Float128")))
df = subset(df, :problem => ByRow==(("nondquar")))
df = subset(df, :solver => ByRow(x -> occursin("trunk", x)))
df = df[:, [:problem, :solver, :neval_obj, :neval_grad, :neval_hprod, :elapsed_time]]
df
```

	problem	solver	neval_obj	neval_grad	neval_hprod	elapsed_time
	String15	String31	Int64	Int64	Int64	Float64
1	nondquar	trunk_cg	164	72	4216	4.1566
2	nondquar	trunk_diom_50	158	77	3679	4.31811
3	nondquar	trunk_diom_5	181	77	4167	3.80609
4	nondquar	trunk_diom_2	169	76	5189	4.6066
5	nondquar	trunk_lbfsgs_100	162	82	3350	4.50562
6	nondquar	trunk_lbfsgs_50	125	53	3823	5.41446
7	nondquar	trunk_lbfsgs_2	229	93	5056	4.58095
8	nondquar	trunk_diom_100	162	82	3350	4.05666

Le projet a été très formateur : il nous a permis de nous familiariser avec les méthodes de région de confiance, en plus de pousser notre compréhension du développement d'outils dans **Julia**.

## Référence

- Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd edition, SIAM, 2003.
- Y. Saad, *Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear systems*, SIAM journal on scientific and statistical computing, 5(1), pp. 203–228, 1984. “ ”

## Annexe

### DIOM avec région de confiance

```
# An implementation of DIOM for the solution of the square linear system Ax = b.
#
# This method is described in
#
# Y. Saad, Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear
# SIAM journal on scientific and statistical computing, 5(1), pp. 203--228, 1984.
#
# Alexis Montoison, <alexis.montoison@polymtl.ca>
# Montreal, September 2018.

export diom, diom!, DiomTRWorkspace


mutable struct DiomTRWorkspace{T,FC,S} <: KrylovWorkspace{T,FC,S}
    m      :: Int
    n      :: Int
    Δx    :: S
    x     :: S
    t     :: S
    z     :: S
    w     :: S
    P     :: Vector{S}
    V     :: Vector{S}
    L     :: Vector{FC}
    H     :: Vector{FC}
    warm_start :: Bool
    stats   :: SimpleStats{T}
end

function DiomTRWorkspace(m::Integer, n::Integer, S::Type; memory::Integer = 20)
    memory = min(m, memory)
    FC   = eltype(S)
    T   = real(FC)
    Δx = S(undefined, 0)
    x   = S(undefined, n)
    t   = S(undefined, n)
    z   = S(undefined, 0)
    w   = S(undefined, 0)
    P   = S[S(undefined, n) for i = 1 : memory-1]
    V   = S[S(undefined, n) for i = 1 : memory]
    L   = Vector{FC}(undefined, memory-1)
    H   = Vector{FC}(undefined, memory)
    S = isconcretetype(S) ? S : typeof(x)
    stats = Krylov.SimpleStats(0, false, false, false, T[], T[], T[], 0.0, "unknown")
    workspace = DiomTRWorkspace{T,FC,S}(m, n, Δx, x, t, z, w, P, V, L, H, false, stats)
    return workspace
end

function DiomTRWorkspace(A, b; memory::Integer = 20)
    m, n = size(A)
```

```

S = ktypeof(b)
DiomTRWorkspace(m, n, S; memory=memory)
end

function DiomTRWorkspace(x; memory::Integer = 20)
    nvar = length(x)
    S = ktypeof(x)
    DiomTRWorkspace(nvar, nvar, S; memory=memory)
end

"""

(x, stats) = diom(A, b::AbstractVector{FC};
                     memory::Int=20, M=I, N=I, ldiv::Bool=false,
                     reorthogonalization::Bool=false, atol::T=√eps(T),
                     rtol::T=√eps(T), itmax::Int=0,
                     timemax::Float64=Inf, verbose::Int=0, history::Bool=false,
                     callback=workspace->false, iostream::IO=kstdout)

`T` is an `AbstractFloat` such as `Float32`, `Float64` or `BigFloat`.
`FC` is `T` or `Complex{T}`.

(x, stats) = diom(A, b, x0::AbstractVector; kwargs...)

```

DIOM can be warm-started from an initial guess `x0` where `kwargs` are the same keyword arguments as above.

Solve the consistent linear system  $Ax = b$  of size  $n$  using DIOM.

DIOM only orthogonalizes the new vectors of the Krylov basis against the `memory` most recent vectors. If CG is well defined on `Ax = b` and `memory = 2`, DIOM is theoretically equivalent to CG. If `k` `memory` where `k` is the number of iterations, DIOM is theoretically equivalent to FOM. Otherwise, DIOM interpolates between CG and FOM and is similar to CG with partial reorthogonalization.

An advantage of DIOM is that non-Hermitian or Hermitian indefinite or both non-Hermitian and indefinite systems of linear equations can be handled by this single algorithm.

#### #### Interface

To easily switch between Krylov methods, use the generic interface `[`krylov_solve`]`(@ref) with `method`.

For an in-place variant that reuses memory across solves, see `[`diom!`]`(@ref).

#### #### Input arguments

- \* `A`: a linear operator that models a matrix of dimension `n`;
- \* `b`: a vector of length `n`.

#### #### Optional argument

- \* `x0`: a vector of length `n` that represents an initial guess of the solution `x`.

#### #### Keyword arguments

```

* `memory`: the number of most recent vectors of the Krylov basis against which to orthogonalize a new vector;
* `M`: linear operator that models a nonsingular matrix of size `n` used for left preconditioning;
* `N`: linear operator that models a nonsingular matrix of size `n` used for right preconditioning;
* `ldiv`: define whether the preconditioners use `ldiv!` or `mul!`;
* `reorthogonalization`: reorthogonalize the new vectors of the Krylov basis against the `memory` most recent vectors;
* `atol`: absolute stopping tolerance based on the residual norm;
* `rtol`: relative stopping tolerance based on the residual norm;
* `itmax`: the maximum number of iterations. If `itmax=0`, the default number of iterations is set to `1000`;
* `timemax`: the time limit in seconds;
* `verbose`: additional details can be displayed if verbose mode is enabled (verbose > 0). Information is collected in the `stats` argument;
* `history`: collect additional statistics on the run such as residual norms, or A-residual norms;
* `callback`: function or functor called as `callback(workspace)` that returns `true` if the Krylov method should stop;
* `iostream`: stream to which output is logged.

#### Output arguments

* `x`: a dense vector of length `n`;
* `stats`: statistics collected on the run in a [`SimpleStats`](@ref) structure.

#### Reference

* Y. Saad, [*Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear systems*]

```

```

function diom end

"""
workspace = diom!(workspace::DiomWorkspace, A, b; kwargs...)
workspace = diom!(workspace::DiomWorkspace, A, b, x0; kwargs...)

```

In these calls, `kwargs` are keyword arguments of [`diom`](@ref).  
The keyword argument `memory` is the only exception.  
It is only supported by `diom` and is required to create a `DiomWorkspace`.  
It cannot be changed later.

See [`DiomWorkspace`](@ref) for instructions on how to create the `workspace`.

For a more generic interface, you can use [`krylov\_workspace`](@ref) with `method = :diom` to allocate a workspace and [`krylov\_solve!`](@ref) to run the Krylov method in-place.

```

"""
function diom! end

def_args_diom = (:(
    A,
    b::AbstractVector{FC}
),)

def_optargs_diom = (:(
    x0::AbstractVector,
))

def_kwargs_diom = (:(
    M = I,
    N = I,
    ldiv::Bool = false,
    radius::T = zero(T),
    reorthogonalization::Bool = false,
    atol::T = √eps(T),
    rtol::T = √eps(T),
))

```

```

    :(; itmax::Int = 0                      ),
    :(; timemax::Float64 = Inf              ),
    :(; verbose::Int = 0                   ),
    :(; history::Bool = false            ),
    :(; callback = workspace -> false   ),
    :(; iostream::IO = kstdout           ))
)

def_kwarg_diom = extract_parameters.(def_kwarg_diom)

args_diom = (:A, :b)
optargs_diom = (:x0,)
kwarg_diom = (:M, :N, :ldiv, :radius, :reorthogonalization, :atol, :rtol, :itmax, :timemax, :verbose,
              :history, :callback, :iostream)

@eval begin
    function diom!(workspace :: Union{DiomWorkspace{T,FC,S}, DiomTRWorkspace{T, FC, S}}, $(def_args_diom))
        # Timer
        start_time = time_ns()
        timemax_ns = 1e9 * timemax

        m, n = size(A)
        (m == workspace.m && n == workspace.n) || error("(workspace.m, workspace.n) = $(workspace.m), $(workspace.n) != $(n) || error("System must be square"))
        length(b) == m || error("Inconsistent problem size")
        (verbose > 0) && @printf(iostream, "DIOM: system of size %d\n", n)

        # Check M = I and N = I
        MisI = (M === I)
        NisI = (N === I)
        # Check M = N
        MisN = (M === N)
        # Check type consistency
        eltype(A) == FC || @warn "eltype(A) $FC. This could lead to errors or additional allocations in operations"
        ktypeof(b) == S || error("ktypeof(b) must be equal to $S")

        # Set up workspace.
        allocate_if(!MisI, workspace, :w, S, workspace.x) # The length of w is n
        allocate_if(!NisI, workspace, :z, S, workspace.x) # The length of z is n

        Δx, x, t, P, V = workspace.Δx, workspace.x, workspace.t, workspace.P, workspace.V
        L, H, stats = workspace.L, workspace.H, workspace.stats
        warm_start = workspace.warm_start
        rNorms = stats.residuals
        reset!(stats)
        w = MisI ? t : workspace.w
        r = MisI ? t : workspace.w

        # Initial solution x and residual r .
        kfill!(x, zero(FC)) # x
        if warm_start
            mul!(t, A, Δx)
            kaxpby!(n, one(FC), b, -one(FC), t)
        else

```

```

    kcopy!(n, t, b)  # t ← b
end
MisI || mulorldiv!(r, M, t, ldiv)  # M(b - Ax)
rNorm = knorm(n, r)                #   = ||r ||
history && push!(rNorms, rNorm)
if rNorm == 0
    stats.niter = 0
    stats.solved, stats.inconsistent = true, false
    stats.timer = start_time |> ktimer
    stats.status = "x is a zero-residual solution"
    warm_start && kaxpy!(n, one(FC), Δx, x)
    workspace.warm_start = false
    return workspace
end

iter = 0
itmax == 0 && (itmax = 2*n)

= atol + rtol * rNorm
(verbose > 0) && @printf(iostream, "%5s %7s %5s\n", "k", "||r||", "timer")
kdisplay(iter, verbose) && @printf(iostream, "%5d %7.1e %.2fs\n", iter, rNorm, start_time |> ktim

mem = length(V)  # Memory
for i = 1 : mem
    kfill!(V[i], zero(FC))  # Orthogonal basis of K (MAN, Mr).
end
for i = 1 : mem-1
    kfill!(P[i], zero(FC))  # Directions P = NV (U)-1.
end
pcg = P[1]
pcgnorm2 = zero(FC)
kfill!(H, zero(FC))  # Last column of the band hessenberg matrix H = LU .
# Each column has at most mem + 1 nonzero elements.
# h. is stored as H[k-i+1], i k. h . is not stored in H.
# k-i+1 represents the indice of the diagonal where h. is located.
# In addition of that, the last column of U is stored in H.
kfill!(L, zero(FC))  # Last mem-1 pivots of L .

# Initial and v .
= rNorm
kdivcopy!(n, V[1], r, rNorm)  # v = r / ||r||

# Stopping criterion.
solved = rNorm
on_boundary = false
tired = iter < itmax
status = "unknown"
user_requested_exit = false
overtimed = false
while !(solved || tired || user_requested_exit || overtimed)
    # Update iteration index.
    iter = iter + 1

```

```

# Set position in circulars stacks.
pos = mod(iter-1, mem) + 1      # Position corresponding to v in the circular stack V.
next_pos = mod(iter, mem) + 1    # Position corresponding to v in the circular stack V.

# Incomplete Arnoldi procedure.
z = NisI ? V[pos] : workspace.z
NisI || mulorldiv!(z, N, V[pos], ldiv) # Nv , forms p
mul!(t, A, z)                         # ANv
MisI || mulorldiv!(w, M, t, ldiv)       # MANv , forms v
for i = max(1, iter-mem+1) : iter
    ipos = mod(i-1, mem) + 1 # Position corresponding to v in the circular stack V.
    diag = iter - i + 1
    H[diag] = kdot(n, w, V[ipos])    # h . = MANv , v
    kaxpy!(n, -H[diag], V[ipos], w) # w ← w - h . v
end
# Partial reorthogonalization of the Krylov basis.
if reorthogonalization
    for i = max(1, iter-mem+1) : iter
        ipos = mod(i-1, mem) + 1
        diag = iter - i + 1
        Htmp = kdot(n, w, V[ipos])
        H[diag] += Htmp
        kaxpy!(n, -Htmp, V[ipos], w)
    end
end

# Compute h . and v .
Haux = knorm(n, w) # h . = ||v ||
if Haux == 0 # h . = 0 "lucky breakdown"
    kdivcopy!(n, V[next_pos], w, Haux) # v = w / h .
end

# Update the LU factorization of H .
# Compute the last column of U .
if iter == 2
    # u . ← h .           if iter == mem
    # u . ← h . . if iter == mem + 1
    for i = max(2, iter-mem+2) : iter
        lpos = mod(i-1, mem-1) + 1 # Position corresponding to l . in the circular stack L.
        diag = iter - i + 1
        next_diag = diag + 1
        # u . ← h . - l . * u .
        H[diag] = H[diag] - L[lpos] * H[next_diag]
        if i == iter
            # Compute the last component of z = (L)⁻¹e .
            #     = -l . *
            #     = - L[lpos] *
        end
    end
end

# Compute next pivot l . = h . / u .
next_lpos = mod(iter, mem-1) + 1

```

```

L[next_lpos] = Haux / H[1]

ppos = mod(iter-1, mem-1) + 1 # Position corresponding to p in the circular stack P.

# Compute the direction p , the last column of P = NV (U ) ^ .
# u . p + ... + u . p = Nv if k < mem
# u . p + ... + u . p = Nv if k >= mem + 1
for i = max(1,iter-mem+1) : iter-1
    ipos = mod(i-1, mem-1) + 1 # Position corresponding to p in the circular stack P.
    diag = iter - i + 1
    if ipos == ppos
        # p ← -u . * p
        kscal!(n, -H[diag], P[ppos])
    else
        # p ← p - u . * p
        kaxpy!(n, -H[diag], P[ipos], P[ppos])
    end
end

# p ← p + Nv
kaxpy!(n, one(FC), z, P[ppos])

# Compute step size to boundary if applicable.
if radius == 0
    = 1/H[1]
elseif NisI && MisI
    pcg = .* P[ppos]
    pcgnorm2 = knorm(n, pcg)^2
    = maximum(to_boundary(n, x, pcg, z, radius, dNorm2= pcgnorm2))
elseif MisN
    = maximum(to_boundary(n, x, pcg, z, radius, M=M, ldiv=!ldiv))
else
    error("Must use split preconditioning")
end

# Move along p from x to the boundary if either
# the next step leads outside the trust region or
# we have nonpositive curvature.
if (radius > 0) && ((1/real(H[1])) < 0) || (1/real(H[1]) > ))
    if 1/real(H[1]) < 0
        stats.indefinite = true
    end
    H[1] = 1/
    on_boundary = true
end

# p = p / u .

kdiv!(n, P[ppos], H[1])

# Update solution x .
# x = x + * p
kaxpy!(n, , P[ppos], x)

```

```

# Compute residual norm.
# || M(b - Ax) || = h . * | / u . |
rNorm = Haux * abs( / H[1])
# New way to compute the residual norm.
# ||M(b - Ax)|| = h . * | / u . |
# rNorm1 = sqrt(rNorm1^2 + ()^2*((c/H[1])^2+ (Haux/H[1])^2) - 2*((-1)^(iter+1)) *(c/H[1])* * rN
history && push!(rNorms, rNorm)
#(verbose > 0) && println("iter: $iter, rNorm: $rNorm, rNorm1: $rNorm1")
# Stopping conditions that do not depend on user input.
# This is to guard against tolerances that are unreasonably small.
resid_decrease_mach = (rNorm + one(T) one(T))

# Update stopping criterion.
user_requested_exit = callback(workspace) :: Bool
resid_decrease_lim = rNorm
solved = resid_decrease_lim || resid_decrease_mach || on_boundary
tired = iter itmax
timer = time_ns() - start_time
overtimed = timer > timemax_ns
kdisplay(iter, verbose) && @printf(iostream, "%5d %7.1e %.2fs\n", iter, rNorm, start_time |> ktm
end
(verbose > 0) && @printf(iostream, "\n")

# Termination status
on_boundary && (status = "on trust-region boundary")
tired && (status = "maximum number of iterations exceeded")
solved && !on_boundary && (status = "solution good enough given atol and rtol")
user_requested_exit && (status = "user-requested exit")
overtimed && (status = "time limit exceeded")

# Update x
warm_start && kaxpy!(n, one(FC), Δx, x)
workspace.warm_start = false

# Update stats
stats.niter = iter
stats.solved = solved
stats.inconsistent = false
stats.timer = start_time |> ktimer
stats.status = status
return workspace
end
end

```

## LBFGS avec région de confiance + *wrapper* autour de `trunk` de JSOSolvers

```

using Pkg
Pkg.activate("projet_env")

"""
The following script is meant to leverage the trust region implementation `trunk` in JSOSolver and
make it compatible with a L-BFGS and DIOM trust region quadratic solver
"""

# TODO: add documentation (doc-string)

module TrunkSolverUpdate

using LinearAlgebra
using LinearOperators
using Krylov
using JSOSolvers
using SolverTools
using NLPMODELS
using ADNLPMODELS

Base.include(Krylov, "diom_tr.jl")

mutable struct LBFGSSStats
    niter::Int
    residuals::Vector{Float64}
    elapsed_time::Vector{Float64}
end

mutable struct LBFGSWorkspace{T, FC <: T, S <: AbstractVector{T}} <: KrylovWorkspace{T, FC, S}
    n::Int      # dimension
    m::Int
    mem::Int     # memory
    Hk::AbstractLinearOperator # Inverse LBFGS Operator
    x0::AbstractVector{T}    # starting point
    pk::AbstractVector{T}    # current point
    gk::AbstractVector{T}    # gradient
    dk::AbstractVector{T}    # search direction
    bk::AbstractVector{T}
    sk::AbstractVector{T}    # step update
    yk::AbstractVector{T}    # gradient update
    x::AbstractVector{T}     # solution vector

    stats::Union{Nothing, LBFGSSStats}
end

function LBFGSWorkspace{T,FC,S}(n::Int, mem::Int; scaling::Bool=true) where {T, FC<:T, S<:AbstractV
    Hk = InverseLBFGSOperator(T, n, mem=mem; scaling=scaling)
    x0 = zeros(T, n)
    pk = zeros(T, n)
    gk = zeros(T, n)

```

```

        dk = zeros(T, n)
        bk = zeros(T, n)
        sk = zeros(T, n)
        yk = zeros(T, n)
        x = zeros(T, n)
        stats = nothing      # Review
        return new{T,FC,S}(n, n, mem, Hk, x0, pk, gk, dk, bk, sk, yk, x, stats)
    end
end

function lbfgs_tr end

function lbfgs_tr! end

function lbfgs_tr(
    A::Union{AbstractLinearOperator, Any},
    b::AbstractVector{T};
    mem::Int=2,
    radius::T=zero(T),
    atol::T=√eps(T),
    rtol::T=√eps(T),
    itmax::Int=0,
    verbose::Int=0,
    callback=workspace -> nothing,
) where {T}

    workspace = LBFGSWorkspace{eltype(b), eltype(b), AbstractVector{eltype(b)}}(length(b), mem)
    return lbfgs_tr!(workspace, A, b; radius, atol, rtol, itmax, verbose, callback)
end

function lbfgs_tr!(
    ws :: LBFGSWorkspace{T, FC, S},
    A,
    b::AbstractVector{FC},
    radius::T=zero(T),
    atol::T=√eps(T),
    rtol::T=√eps(T),
    itmax::Int=0,
    verbose::Int=0,
    callback=workspace -> false) where {T<:AbstractFloat, FC<:T, S<:AbstractVector{FC}}


    reset!(ws.Hk)

    copyto!(ws.gk, b)
    gnorm0 = gnormk = norm(ws.gk)

    tolerance = atol + rtol*gnorm0

    if itmax == 0
        itmax = 2*ws.n
    end

```

```

fill!(ws.pk, zero(T))
alphak = zero(T)

callback(ws)

k = 1
done = false

while !done
    mul!(ws.dk, -ws.Hk, ws.gk) # compute search direction
    mul!(ws.bk, A, ws.dk)      # compute curvature

    dkbk = dot(ws.dk, ws.bk)

    # handle negative curvature
    if radius > 0.0 && dkbk <= 0
        alphak = -sign(dot(ws.gk, ws.dk))*2*radius/norm(ws.dk)
    else
        alphak = -dot(ws.gk, ws.dk)/dkbk    # compute search step
    end

    ws.sk .= alphak .* ws.dk      # scale search direction
    ws.pk .+= ws.sk              # update current point

    if radius > 0 && norm(ws.pk) >= radius
        ws.pk .-= ws.sk
        pksk = dot(ws.pk, ws.sk)
        sksk = dot(ws.sk, ws.sk)

        tau = (-pksk + sqrt(pksk^2 + sksk*(radius^2 - dot(ws.pk, ws.pk))))/sksk

        return ws.pk .+ tau .* ws.sk, nothing # TODO: add SimpleStats
    end

    ws.yk .= alphak .* ws.bk      # compute gradient update
    ws.gk .+= ws.yk              # update gradient

    gnormk = norm(ws.gk)
    k += 1

    if gnormk <= tolerance
        done = true
    elseif k >= itmax
        done = true
    end

    # update the inverse Hessian approximation
    if !done
        push!(ws.Hk, ws.sk, ws.yk)
    end

    callback(ws)
end

```

```

    return ws.pk, nothing # TODO: add SimpleStats
end

function Krylov.krylov_solve!(
    ws::LBFGSWorkspace{T},
    A::LinearOperator{T},
    b::AbstractVector{T};
    atol::T = √eps(T),
    rtol::T = √eps(T),
    radius::T = zero(T),
    itmax::Int = 0,
    timemax::Float64 = Inf,
    verbose::Int = 0,
    kwargs...
) where {T}

    x, stats = lbfgs_tr!(ws, A, -1 .* b, radius=radius, atol=atol, rtol=rtol, itmax=itmax, verbose=verbose)

    ws.x = x
    ws.stats = stats
end

function lbfgs_workspace(::Val{:lbfgs}, n::Int, V::Type{<:AbstractVector}; mem=5, scaling=true)
    return LBFGSWorkspace{eltype(V), eltype(V), V}(n, mem; scaling=scaling)
end

function Krylov.diom(A::LinearOperator, b::AbstractVector{T}; memory::Int = 20, radius::T=zero(T), atol::T=√eps(T))
    workspace = DiomTRWorkspace(A, b; memory=memory)
    diom!(workspace, A, b, radius=radius, atol=atol, rtol=rtol, itmax=itmax, verbose=verbose)
    return workspace.x, workspace
end

function Krylov.krylov_solve!(
    ws::DiomTRWorkspace{T},
    A::LinearOperator{T},
    b::AbstractVector{T};
    atol::T = √eps(T),
    rtol::T = √eps(T),
    radius::T = zero(T),
    itmax::Int = 0,
    timemax::Float64 = Inf,    # TODO
    verbose::Int = 0,
    kwargs...
) where {T}

    diom!(ws, A, b, radius=radius, atol=atol, rtol=rtol, itmax=itmax, verbose=verbose)
end

function TrunkSolver(
    nlp::AbstractNLPModel{T, V};

```

```

bk_max::Int = get(JSOSolvers.TRUNK_bk_max, nlp),
monotone::Bool = get(JSOSolvers.TRUNK_monotone, nlp),
nm_itmax::Int = get(JSOSolvers.TRUNK_nm_itmax, nlp),
subsolver::Symbol = :cg,
mem::Int=5,
scaling::Bool=true,
) where {T, V <: AbstractVector{T}}
params = TRUNKParameterSet(nlp; bk_max = bk_max, monotone = monotone, nm_itmax = nm_itmax)
nvar = nlp.meta.nvar
x = V(undef, nvar)
xt = V(undef, nvar)
gx = V(undef, nvar)
gt = V(undef, nvar)
gn = isa(nlp, JSOSolvers.QuasiNewtonModel) ? V(undef, nvar) : V(undef, 0)
Hs = V(undef, nvar)

krylov_subsolver =
    if subsolver == :cg
        krylov_workspace(Val(subsolver), nvar, nvar, V)
    elseif subsolver == :lbfgs
        lbfgs_workspace(Val(subsolver), nvar, V; mem=mem, scaling=scaling)
    elseif subsolver == :diom
        DiomTRWorkspace(nlp.meta.x0; memory=mem)
    else
        throw("Not a valid subsolver choice.")
    end

Sub = typeof(krylov_subsolver)
H = hess_op!(nlp, x, Hs)
Op = typeof(H)
tr = TrustRegion(gt, one(T))
return JSOSolvers.TrunkSolver{T, V, Sub, Op}(x, xt, gx, gt, gn, Hs, krylov_subsolver, H, tr, params)
end

function trunk(
    nlp::AbstractNLPModel;
    x::V = nlp.meta.x0,
    bk_max::Int = get(JSOSolvers.TRUNK_bk_max, nlp),
    monotone::Bool = get(JSOSolvers.TRUNK_monotone, nlp),
    nm_itmax::Int = get(JSOSolvers.TRUNK_nm_itmax, nlp),
    subsolver::Symbol = :cg,
    mem::Int=5,
    scaling::Bool=true,
    kwargs...,
) where {V}
    solver = TrunkSolver(
        nlp;
        bk_max = bk_max,
        monotone = monotone,
        nm_itmax = nm_itmax,
        subsolver = subsolver,
        mem = mem,
        scaling = scaling,

```

```

)
return solve!(solver, nlp; x = x, kwargs...)
end

end # module

# f(x) = x[1]^2 * x[2]^2
# x0 = ones(2)
# nlp = ADNLPModel(f, x0)

# A = hess_op(nlp, x0)
# b = grad(nlp, x0)

# # # lbfsgs_tr(A, b)

# solver = TrunkSolver(nlp, subsolver=:lbfsgs, mem=2, scaling=false)
# stats = solve!(solver, nlp)
# print(stats)

# # solver = TrunkSolver(nlp, subsolver=:lbfsgs, mem=10, scaling=true)
# # stats = solve!(solver, nlp)
# # print(stats)

```

## Notre implémentation de l'algorithme de région de confiance

```
using Pkg
Pkg.activate("projet_env")

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

module TRSolverModule

using LinearAlgebra, Logging, Printf
using Krylov, LinearOperators, NLPMODELS, ADNLPMODELS, SolverTools, SolverCore

# include("subsolvers.jl")
include("TrunkSolverUpdate.jl")

import SolverCore.solve!
export solve!

mutable struct TRSolver{
    T,
    V <: AbstractVector{T},
    Sub <: KrylovWorkspace{T, T, V},
    Op <: AbstractLinearOperator{T},
} <: AbstractOptimizationSolver
    x::V
    g::V
    H::Op
    subsolver_ws::Sub
end

function TRSolver(
    nlp::AbstractNLPMODEL{T, V};
    subsolver::Symbol = :cg,
    mem::Int=2,
    scaling::Bool=true,
) where {T, V <: AbstractVector{T}}
    nvar = nlp.meta.nvar
    x = copy(nlp.meta.x0)      # V(undefined, nvar)
    g = grad(nlp, x)          # V(undefined, nvar)
    H = hess_op(nlp, x)       #, Hs)
    Op = typeof(H)

    subsolver_ws =
        if subsolver == :cg
            krylov_workspace(Val(subsolver), nvar, nvar, V)
        elseif subsolver == :lbfsgs
            TrunkSolverUpdate.lbfsgs_workspace(Val(subsolver), nvar, V; mem=mem, scaling=scaling)
        elseif subsolver == :diom
            TrunkSolverUpdate.DiOMTRWorkspace(nlp.meta.x0; memory=mem)
        else
            throw("Invalid subsolver.")
        end
    Sub = typeof(subsolver_ws)
```

```

    return TRSolver{T, V, Sub, Op}(x, g, H, subsolver_ws)
end

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

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

"""
trsolver(nlp; kwargs...)

A trust-region solver for unconstrained optimization leveraging L-BFGS and DIOM to solve the quadratic
solver.

function trsolver(
    nlp::AbstractNLPModel;
    x::V = nlp.meta.x0,
    subsolver::Symbol = :cg,
    mem::Int = 2,                      # memory parameter
    scaling::Bool = true,               # scaling parameter
    kwargs...,
) where {V}
    solver = TRSolver(nlp; subsolver, mem=mem, scaling=scaling)
    stats = GenericExecutionStats(nlp)
    return solve!(solver, nlp, stats; x = x, kwargs...)
end

function solve!(
    solver::TRSolver{T, V},
    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)),
    fixed_sub_rtol = false,           # use fixed relative tolerance for the quadratic solver
    verbose::Int = 0,    # TODO: to be implemented
    mu = 0.25,
    eta = 0.01,
    delta_max = zero(T),
    max_iter = 10*nlp.meta.nvar,
    max_time = 30.0,
) where {T, V <: AbstractVector{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 the algorithm

SolverCore.reset!(stats)

start_time = time()
elapsed_time = 0.0
set_time!(stats, 0.0)

f = obj(nlp, solver.x)

solver.H = hess_op(nlp, solver.x)

delta = one(T)

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

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

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

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

    krylov_solve!(solver.subsolver_ws, solver.H, -1 .* solver.g, atol=atol, rtol=sub_rtol, radius=0)
    p .= solver.subsolver_ws.x

    b .= solver.H * p      # using hess_op

    f_new = obj(nlp, solver.x .+ p)
    rho = (f_new - f) / (dot(p, solver.g) + (dot(p, b))) / 2

    # bad approximation
    if rho <= mu

```

```

    delta /= 4

    # very good approximation
    elseif rho > 1 - mu && delta_max > 0
        delta = min(delta/2, delta_max)
    end

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

        solver.H = hess_op(nlp, solver.x)

        if !fixed_sub_rtol
            sub_rtol = max(rtol, min(sqrt(grad_norm), T(0.1)))
        end
    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, solver.x)
stats
end
end      # module

# f(x) = x[1]^2 * x[2]^2
# x0 = ones(2)
# nlp = ADNLPModel(f, x0)

```

```
# A = hess_op(nlp, x0)
# b = grad(nlp, x0)

# # # lbfsgs_tr(A, b)

# # solver = TRSolver(nlp, subsolver=:lbfsgs, mem=2, scaling=false)
# # stats = solve!(solver, nlp)
# # print(stats)

# stats = trsolver(nlp, subsolver=:lbfsgs)
# println(stats)

# # solver = TrunkSolver(nlp, subsolver=:lbfsgs, mem=10, scaling=true)
# # stats = solve!(solver, nlp)
# # print(stats)
```