Rapport de projet - phase 1

MTH8408

Oihan Cordelier, Oussama Mouhtal

Problématique

Dans la cadre des algorithmes de résolution de problème d'optimisation en tenant compte de l'information de second ordre sous région de confiance, il y a toujours des problèmes quadratiques à résoudre. La méthode la plus utilisée et le gradient conjugué. Il existe des méthodes qui approxime la hessienne et sont connues sous le nom de méthodes quasi-newton. Une des plus utilisées est la méthode LBFGS. Il est connu que la méthode CG souffre en orthogonalisation. Pour cela qu'il serait intéressant d'explorer d'autres méthodes plus stables, notamment LBFGS.

Objectifs

Dans ce projet, le but sera de comparer la méthode CG et la méthode LBFGS dans le cadre de résolutions de problèmes quadratiques. Les objectifs suivants ont été identifiés :

1. Implémentation optimisée des algorithmes LBFGS pour le cas quadratique et newton inexact.
2. Compréhension théorique des résultats entre quasi newtons, notamment LBFGS et gradient conjugué
3. Comparaison numérique entre les CG et LBFGS sur les banques de problèmes Optimization Problems et CUTest et étudier la relation entre reorthogonalisations dans CG (le nombre de vecteurs à re-orthogonalisés) et la mémoire dans LBFGS
4. Validation théorique de la relation entre reorthogonalisation dans CG et la mémoire dans LBFGS (si le temps le permet)

Plan d'action

Le brouillon de l'article (Bourhis et al, 2020) ainsi que le rapport de stage (Bourhis, 2019) sont les deux références principales pour mener à bien ce projet.

À cela s'ajoutent les notes du cours MTH8408 ainsi que le manuel Numerical Optimization de (Nocedal et al., 2006) où les pseudo-codes de divers algorithmes sont présents.

Des problèmes non contraints des banques de problèmes *OptimizationProblems* et CUTEst (Gould et al., 2015) seront utilisés pour valider les implémentations des méthodes d'optimisation.

Impact attendu

Ce travail vise à améliorer la compréhension des liens entre la méthode du gradient conjugué et LBFGS, dans le contexte des sous-problèmes quadratiques en région de confiance. Une meilleure compréhension de ces liens, en particulier entre le processus de réorthogonalisation dans CG et le paramètre de mémoire dans LBFGS, pourrait permettre de mieux choisir entre les deux approches selon la nature du problème (taille, conditionnement, coût de stockage, etc.).

Résultats préliminaires

La présente section présente une implémentation préliminaire de méthode de newton inexacte avec région de confiance qui utilise le CG ou LBFGS pour résoudre le sous-problème quadratique.

```julia
using Pkg
Pkg.activate("projet_env")
Pkg.add("ADNLPModels")
Pkg.add("NLPModels")
Pkg.add("Krylov")
Pkg.add("LinearOperators")

Pkg.add("Plots")

Pkg.add("OptimizationProblems")  # collection + outils pour sélectionner les problèmes
# TODO: add CUTest

using LinearAlgebra, NLPModels , ADNLPModels, Printf, Krylov, LinearOperators

using OptimizationProblems, OptimizationProblems.ADNLPProblems

using Plots
# gr(fmt = :png)
```

Implémentation préliminaire de la méthode LBFGS pour résoudre le sous-problème quadratique sous région de confiance

Il s'agit d'une première implémentation qui requiert encore des améliorations. Cette méthode provient du rapport (Bourhis et al., 2019).

Cette implémentation propose une manière pour éviter les courbures négatives, tel que : $d_k^T A d_k < 0$, alors le pas sera calculé de tel sorte qu'il sortira de la région de confiance. Un re-dimensionnement sera alors appliqué pour rester à sur la frontière de la région de confiance. Si l'itéré courant sort du rayon de confiance, alors la direction est mise à l'échelle par un scalaire τ de tel que sorte que l'itéré sera dans la région de confiance.

```julia
"""
    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, gk, delta; atol=1e-5, rtol=1e-5, mem = 5, max_iter = 10)

    dim = length(gk)

    gnorm0 = norm(gk)

    k = 1
    pk = zeros(dim)

    # TODO: scaling should be an argument
    Hk = InverseLBFGSOperator(dim, mem = mem, scaling = true)

    # TODO: review the use of norm(gk) here
    while norm(gk) > atol + rtol * gnorm0 && k <= max_iter

        dk = -Hk*gk

        # TODO: review if there is a more efficient way of computing Bj*dk
```

```
        bk =  Bj*dk

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

        sk = alphak*dk
        pk = pk + sk

        if norm(pk) >= delta
            pk = 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
        k = k + 1

        # update the inverse Hessian approximation
        push!(Hk, sk, yk)

    end

    # TODO: return a stats like object similar to the Krylov.cg method
    return pk
end
```

Main.Notebook.lbfgs

```
# struct to track the evolution of an optimization problem. (For later analysis between
# the stutied methods)
mutable struct OptimizationResult
    sol::Vector         # argmin of f
    fun::Float64        # min of f
    fk::Vector          # evolution of fk
    gk_norm::Vector     # evolution of ||gk||
    converged::Bool     # convergence flag -> did the optimizer converge?
end

"""
    newton_inexact_trust_region(model; sub_optimizer="cg", atol=1.0e-5, rtol=1.0e-5, mem=10, max_iter_ra

Minimize a non-linear function with a trust region approach. The implementation
is based on: https://www.gerad.ca/fr/papers/G-2019-64.

The sub-problem minimization problem method can be set by the user (either cg or lbfgs).
"""
function newton_inexact_trust_region(model; sub_optimizer="cg", atol=1.0e-5, rtol=1.0e-5, mem=10, max_it
```

```
# TODO: add verbose argument

xk = model.meta.x0
n = length(xk)
fk = obj(model, xk)
gk = grad(model, xk)
gnorm = norm(gk)
gnorm0 = deepcopy(gnorm)

k = 0
deltak = 1.0
@printf "%2s  %9s  %7s\n" "k" "fk" "‖grad‖"
@printf "%2d  %9.2e  %7.1e\n" k fk gnorm

# save convergence history
fk_history = Vector{Float64}()
gk_norm_history = Vector{Float64}()

# push starting point and starting ||gk||
push!(fk_history, fk)
push!(gk_norm_history, gnorm)

converged_flag = false

max_iter = max_iter_ratio*n

while gnorm > atol + rtol * gnorm0 && k < max_iter
    Bk = hess(model,xk)
    ##############################
    #    Trust region update     #
    ##############################
    # select sub optimizer CG or L-BFGS
    if sub_optimizer == "cg"
        (sk, stats) = Krylov.cg(Bk, -gk, radius=deltak, atol=atol_inner, rtol=rtol_inner)
    elseif sub_optimizer == "lbfgs"
        sk = lbfgs(Bk, gk, deltak, atol=atol_inner, rtol=rtol_inner, mem=mem, max_iter=max_iter)
    else
        throw("Selected sub optimizer is not valid.")
    end

    ##############################
    #       Trust region         #
    ##############################
    # predicted objective reduction
    red_pred = -dot(gk, sk) - 0.5*(sk'*(Bk*sk))

    f_new = obj(model, xk .+ sk)

    # actual obj reduction
    red_act = fk - f_new

    # actual obj reduction to predicted obj red ratio
    rho = red_act / red_pred
```

```
        # if reduction is greater than tolerance, accept step size. Otherwise,
        # reject step size
        # TODO: should be function argument
        if rho >= 1e-4
            # update new point
            xk .+= sk
            fk = f_new
            gk .= grad(model, xk)
            gnorm = norm(gk)
        end

        # increase / decrease trust region radius
        # TODO: should be function argument
        if rho >= 0.99
            deltak *= 3
        elseif rho < 1e-4
            deltak /= 3
        end
        ##############################

        k += 1
        @printf "%2d  %9.2e  %7.1e\n" k fk gnorm

        push!(fk_history, fk)
        push!(gk_norm_history, gnorm)

    end

    # update convergence flag
    if gnorm <= atol + rtol * gnorm0
        converged_flag = true
    end

    # save optimization data to OptimizationResult struct
    optimize_result = OptimizationResult(
        xk,
        fk,
        fk_history,
        gk_norm_history,
        converged_flag
    )

    # return solution and optimization history
    return xk, optimize_result
end
```

`Main.Notebook.newton_inexact_trust_region`

Les méthodes implémentés sont testées sur une fonction test provennant de Optimization Problems.

```
meta = OptimizationProblems.meta
problem_list = meta[(meta.ncon.==0).&.!meta.has_bounds.&(meta.nvar.==100), :name]
problems = (OptimizationProblems.ADNLPProblems.eval(Meta.parse(problem))() for problem   problem_list)
```

```
models = []
push!(models, OptimizationProblems.ADNLPProblems.chainwoo())
push!(models, OptimizationProblems.ADNLPProblems.errinros_mod())
push!(models, OptimizationProblems.ADNLPProblems.freuroth())

# Validation of the newton inexact with CG
x_star, res_cg = newton_inexact_trust_region(models[2], sub_optimizer="cg")
```

```
 k         fk    ‖grad‖
 0   1.57e+05  9.5e+04
 1   8.43e+04  5.4e+04
 2   1.68e+04  1.6e+04
 3   3.34e+03  4.7e+03
 4   7.05e+02  1.4e+03
 5   7.05e+02  1.4e+03
 6   2.52e+02  4.3e+02
 7   7.69e+01  1.2e+02
 8   7.69e+01  1.2e+02
 9   7.69e+01  1.2e+02
10   7.69e+01  1.2e+02
11   7.69e+01  1.2e+02
12   7.69e+01  1.2e+02
13   5.33e+01  4.1e+01
14   5.33e+01  4.1e+01
15   5.33e+01  4.1e+01
16   4.66e+01  1.5e+01
17   4.66e+01  1.5e+01
18   4.35e+01  8.1e+00
19   4.35e+01  8.1e+00
20   4.13e+01  6.3e+00
21   4.11e+01  2.0e+01
22   4.05e+01  1.8e+01
23   3.95e+01  8.8e+00
24   3.93e+01  9.3e-01
```

```
([1.2210382038940586, 0.1153220318557405, 0.08206406827991462, 0.20615508578227734, 0.32076071632274955
```

```
models = []
push!(models, OptimizationProblems.ADNLPProblems.chainwoo())
push!(models, OptimizationProblems.ADNLPProblems.errinros_mod())

# Validation of the newton inexact with LBFGS
x_star, res_lbfgs = newton_inexact_trust_region(models[2], sub_optimizer="lbfgs", mem=50, atol_inner=1e
```

```
 k         fk    ‖grad‖
 0   1.57e+05  9.5e+04
 1   8.43e+04  5.4e+04
 2   1.68e+04  1.6e+04
 3   3.34e+03  4.7e+03
 4   7.05e+02  1.4e+03
 5   7.05e+02  1.4e+03
 6   2.52e+02  4.3e+02
 7   7.68e+01  1.2e+02
 8   7.68e+01  1.2e+02
```

```
 9   7.68e+01  1.2e+02
10   7.68e+01  1.2e+02
11   7.68e+01  1.2e+02
12   7.68e+01  1.2e+02
13   5.32e+01  4.1e+01
14   5.32e+01  4.1e+01
15   5.32e+01  4.1e+01
16   4.66e+01  1.5e+01
17   4.66e+01  1.5e+01
18   4.35e+01  8.0e+00
19   4.35e+01  8.0e+00
20   4.13e+01  6.2e+00
21   4.11e+01  2.0e+01
22   4.05e+01  1.8e+01
23   3.95e+01  8.7e+00
24   3.93e+01  9.2e-01
```

```
([1.2189755325603713, 0.1152329693336169, 0.08184988086843517, 0.20610994087623283, 0.32073051000156916
```