Phase 3 du projet

MTH8211

Ulrich Baron-Fournier
Polytechnique Montréal
ulrich.baron-fournier@polymtl.ca

Remise 3 — Rapport final

1) Description de la problématique (révisée)

Le projet s'intéresse à la résolution de grands systèmes linéaires fortement rectangulaires, c'est-à-dire des problèmes aux moindres carrés linéaires où la matrice A est de taille $m \times n$ avec un écart extrême entre m et n (beaucoup plus d'équations que d'inconnues si $m \gg n$, cas surdéterminé, ou l'inverse $m \ll n$, cas sous-déterminé). Dans de telles situations, on cherche typiquement à calculer la solution de norme minimale du problème $\min_x \frac{1}{2} \parallel Ax - b \parallel_2^2$ c'est-à-dire la solution de plus petite longueur qui satisfait au mieux $Ax \approx b$. Ces problèmes apparaissent dans de nombreuses applications scientifiques et d'ingénierie, et la demande en solveurs plus rapides et précis s'accroît avec la taille des données et des modèles. Les approches classiques pour les moindres carrés (telles que la résolution des équations normales $A^T Ax = A^T b$ ou les factorisations QR/SVD) deviennent coûteuses ou peu pratiques lorsque m et n sont très déséquilibrés. En effet, m et n sont très déséquilibrés, on parle implicitement de valeurs extrêmement grands pour m comparativement à un valeur normale ou n (ou le contraire dans le cas sous-déterminé). Les matrices qui ont des dimensions très grandes peuvent plus facilement souffrir d'un mauvais conditionnement. Les méthodes itératives comme LSQR réduisent les besoins en mémoire, mais leur vitesse de convergence dépend fortement du conditionnement de A. Dans le cas de systèmes extrêmement rectangulaires, il est important de préconditionner le problème pour en améliorer le conditionnement avant d'appliquer un solveur itératif. C'est dans ce contexte qu'intervient LSRN, une méthode introduisant un élément aléatoire dans le processus de résolution.

Principe de l'algorithme LSRN

Dans l'algorithme **LSRN**, on considère d'abord une matrice rectangulaire de grande taille : $A \in \mathbb{R}^{m \times n}, \quad m \gg n$. On génère ensuite une matrice gaussienne aléatoire : $G \in \mathbb{R}^{s \times m}, \quad s \approx \gamma n$ où γ est un facteur de suréchantillonnage (typiquement entre 2 et 4).
Le produit matriciel $\tilde{A} = GA \in \mathbb{R}^{s \times n}$ correspond à une projection de A dans un espace de dimension beaucoup plus réduite, puisque la taille de $\tilde{A}$ est $(\gamma n) \times n$, indépendante de m. La décomposition en valeurs singulières (SVD) est alors effectuée sur $\tilde{A}$, et non sur A. Dans une approche classique, la SVD de A coûte :$O(mn^2)$ ce qui devient prohibitif lorsque m est très grand. Avec LSRN, la SVD est réalisée sur $\tilde{A}$ de taille bien plus petite, pour un coût : $O(\gamma n^3)$ qui dépend uniquement de n, et non plus de m. Ainsi, lorsque m est extrêmement grand mais que n reste

modéré (cas typique en régression surdéterminée avec un très grand nombre d'observations), LSRN permet un gain de temps considérable tout en fournissant un préconditionnement efficace pour les méthodes itératives comme LSQR.

En somme, l'objectif est de résoudre rapidement et avec précision des systèmes très rectangulaires de grande taille – un défi que des méthodes aléatoires parallèles récentes comme LSRN cherchent à relever. Dans le cadre de ce projet, nous avons pour ambition de lire et comprendre en détail l'algorithme LSRN (décrit par Meng et al. en 2014), puis d'en fournir une implémentation efficace en Julia et d'évaluer ses performances sur des problèmes de grande dimension. Le présent rapport (Phase 2) fait suite à une phase 1 où la problématique et les objectifs ont été définis, et il se concentre sur l'implémentation de LSRN (version séquentielle et parallèle) ainsi que sur les résultats préliminaires obtenus. Le code développé est disponible dans le dépôt GitHub du projet (https://github.com/Ulrizpascuit/Projet_MTH8211.git).

2) Difficultés rencontrées et comment elles ont été surmontées

Choix initial du solveur

Dans l'article, pluisieurs solveurs ont été utilisés et comparé entre eux. Afin de simplifier le tout, la fonction LSQR de krylov a été choisi pour pouvoir faire la la distinction entre l'utilisation ou non du préconditionneur LSRN.

Parallélisme

L'utilisation de `Threads.@threads` sur la multiplication $\tilde{A} = GA$ n'apportait pas de gain notable, possiblement parce que BLAS était déjà multi-threadé. Nous avons donc contrôlé explicitement le nombre de threads via `BLAS.set_num_threads(k)` pour comparer proprement : BLAS mono-thread vs. BLAS multi-thread par défaut.

Cas sous-déterminé

LSRN a été adapté pour $m < n$ en appliquant la projection aléatoire sur A^T. Les mêmes résultats que pour le cas sur-déterminé ont été produit. Dans la section analyse des résultats, on confirmera le bon fonctionnement de LSRN sur des problèmes sous-déterminés.

Génération de matrices tests

Pour couvrir un large éventail de $(m,n,\\kappa)$, nous avons écrit des générateurs (denses) contrôlant le conditionnement. L'objectif de trouver une bibliothèque de problèmes creux (ou bien générer de manière aléatoire des matrices creuses) a été mis de côtés en raison des contraintes de temps.

3) Ce qui a été accompli vs. objectifs initiaux

- Implantation séquentielle LSRN (sur- et sous-déterminé) validée.
- Intégration avec LSQR
- Générateurs de matrices fortement rectangulaires mal conditionnées pour tests systématiques.
- Benchmarks sur tailles allant jusqu'à $10^6 \times 10^3$ (denses) mettant en évidence une réduction drastique des itérations par rapport à LSQR seul lorsque $\kappa(A)$ est élevé.

- Analyse de l'effet du parallélisme BLAS sur les étapes coûteuses (projection et produits matrice-vecteur).
- Rédaction du présent rapport final avec figures extraites automatiquement du notebook fourni.

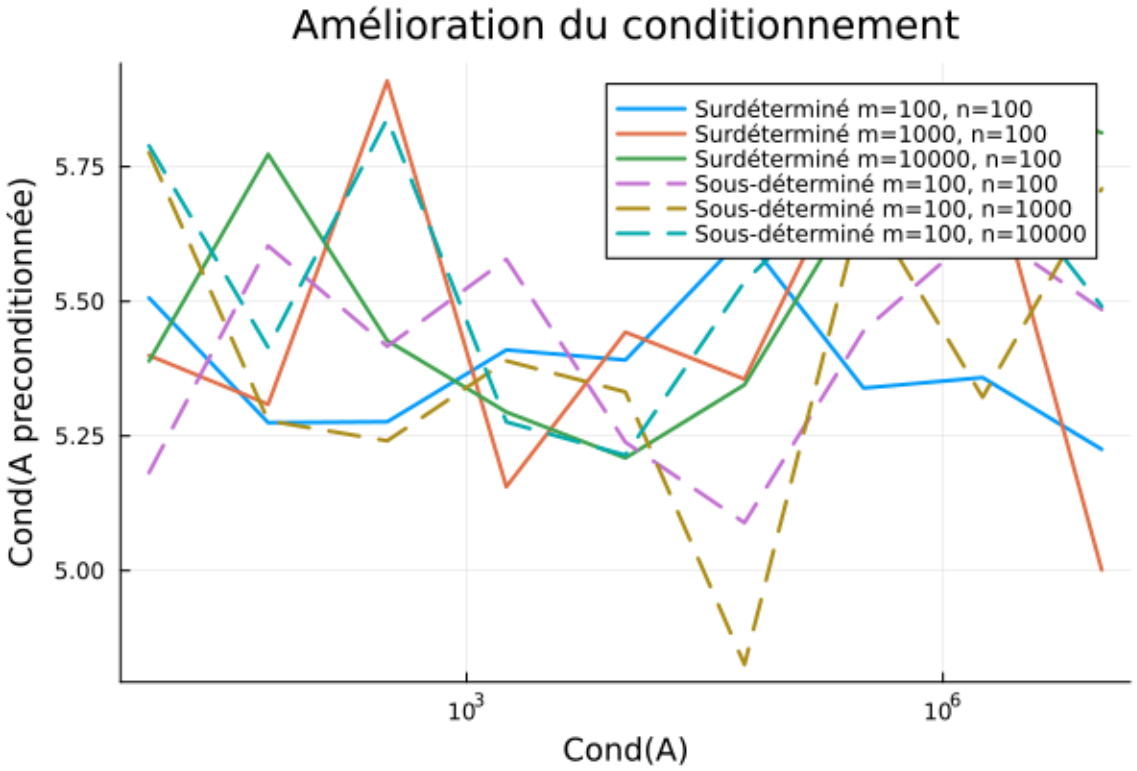
4) Analyse des résultats

4.1 Résumés quantitatifs (itérations/temps)

Les expériences confirment que, pour des matrices très rectangulaires et mal conditionnées, le préconditionneur LSRN ramène le conditionnement relativement proche de 1, ce qui se traduit par un nombre d'itérations quasi indépendant de $\kappa(A)$ pour LSQR préconditionné.

4.2 Figures et graphiques extraits du notebook

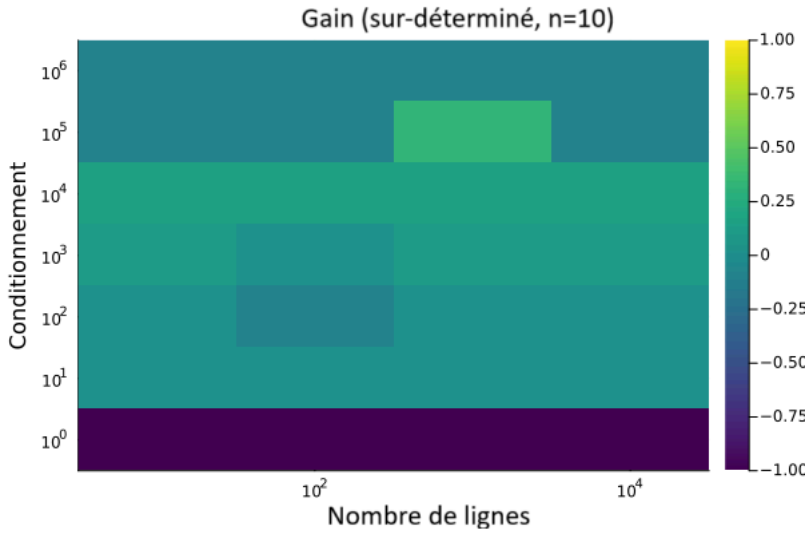
Premièrement, afin de montrer le fonctionnement du préconditionneur de LSRN, on trace le conditionnement du système préconditionné en fonction du conditionnement du système initial pour différentes dimensions de problème.

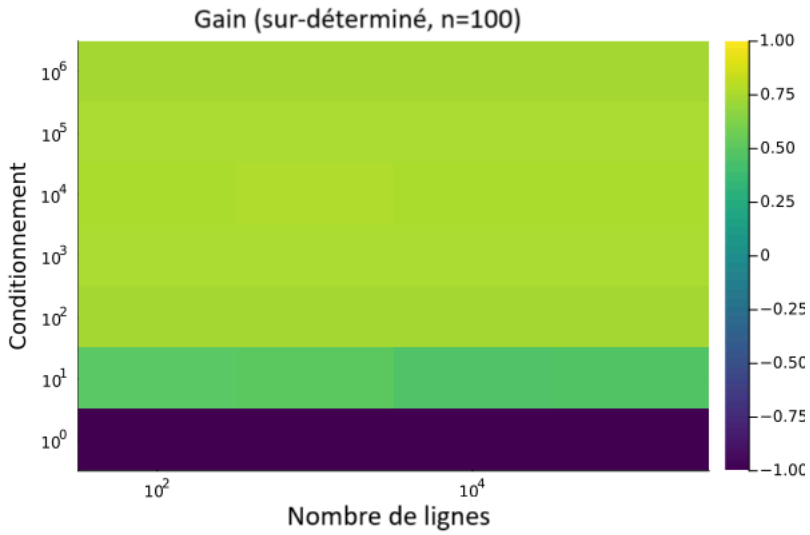


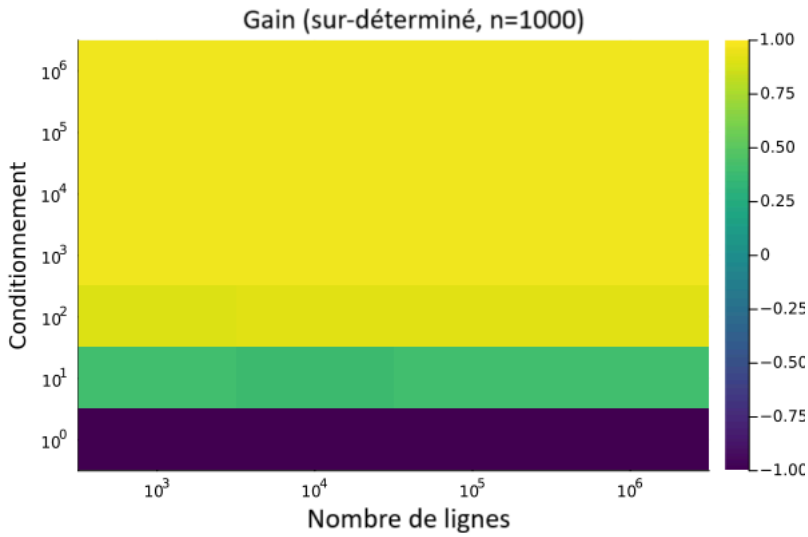
On voit ici que pour n'importe que conditionnement initial (Cond(A) = 1 à Cond(A) = 10000000), le conditionnement du système preéconditionné par LSRN est toujours entre environ 4.75 et 6. C'est le cas pour différentes dimension de matrices sur-déterminées et sous-déterminées.

Cela étant dit, on peut maintenant vérifier que cet amélioration du conditionnement a un impact direct sur le nombre d'itérations de LSQR sur un problème. Afin de quantifier l'amélioration en itération pour produire des graphique, on introduit une variable gain qui est définie comme: Gain $= max(1 - \frac{\text{nb Iter LSRN\_LSQR}}{\text{nb Iter LSQR}}$,-1) afin d'afin d'avoir une quantité normalisée offrant la même information essentielle servant à comparer les deux algorithmes. De cette maninère, on aura un

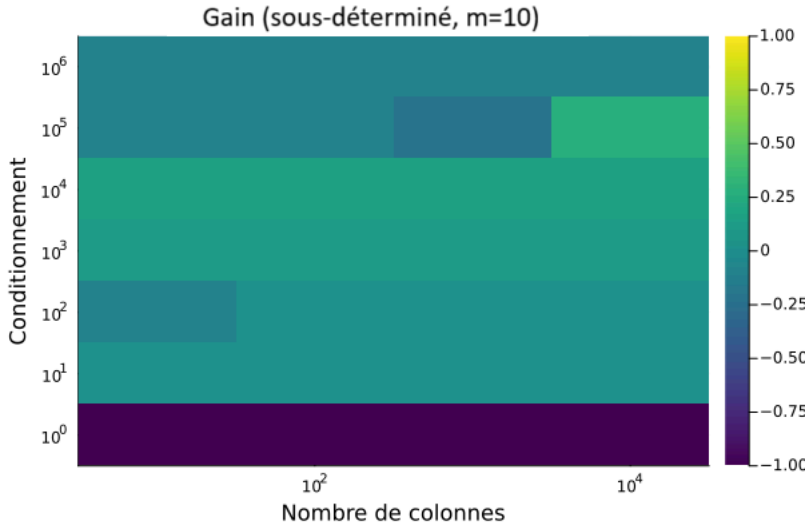
gain=-1 si l'algorithme LSQR seul est supérieur à LSRN_LSQR et un gain=1 dans le cas contraire. Les figures suivantes présentent le gain en itération obtenu par LSRN en fonction du nombre de lignes ou de colonnes (système fortement rectangulaire ou non) et du conditionnement du système. Dans le titre de chaque figure, on voit si on se trouve dans le cas sur ou sous-déterminé. De plus, puisque on regarde le comportement du gain en fonction des dimensions, plusieurs tailles ont été testées. En effet, pour le cas sur déterminé par exemple, trois graphiques sont présentés avec des n fixés différents.

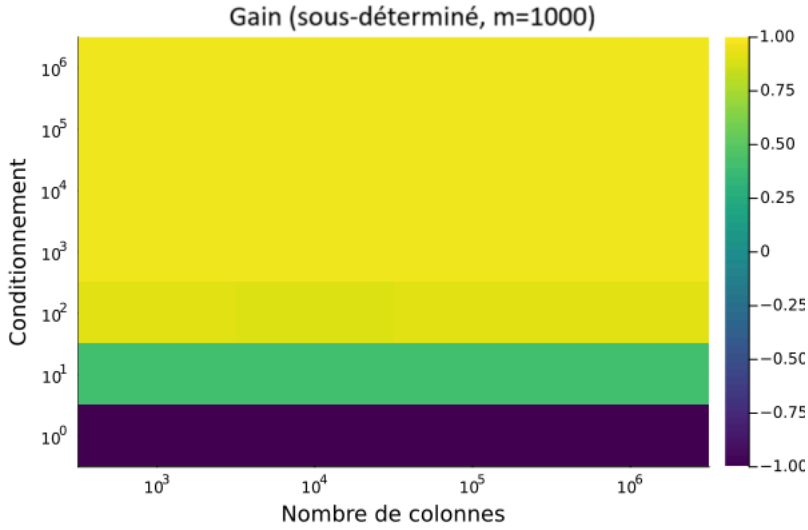




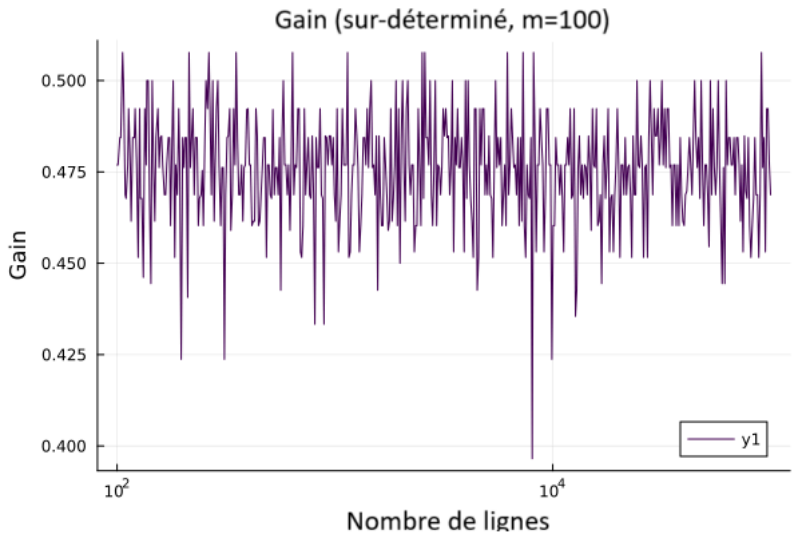


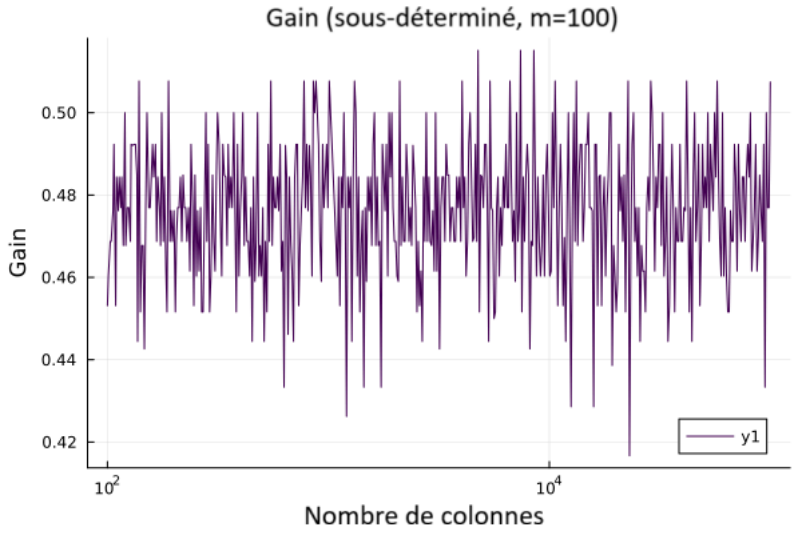
Pour les systèmes sous-déterminés, on a:



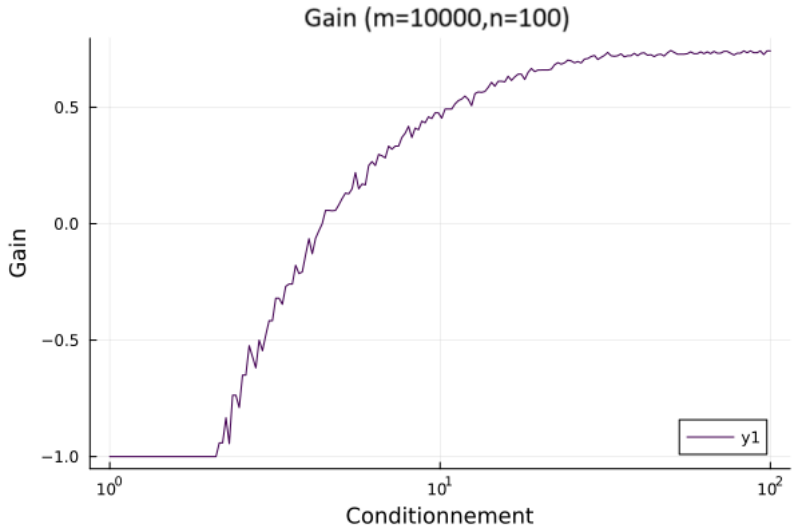


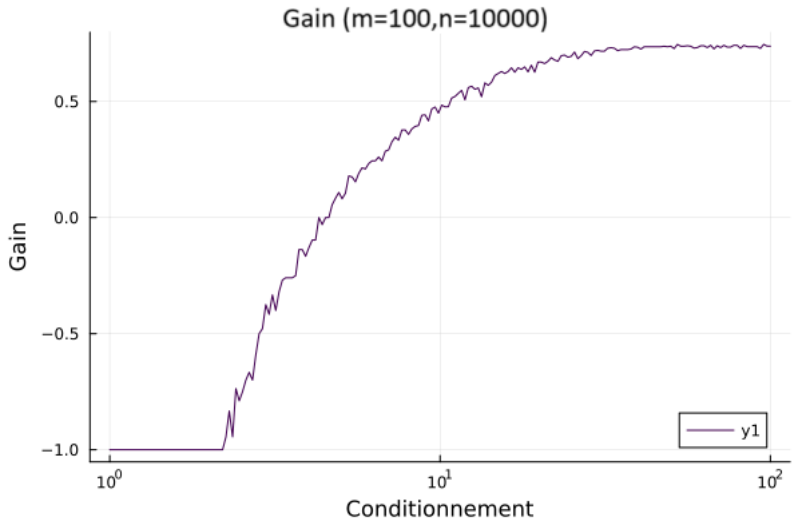
Premièrement, on trouve pratiquement le même comportement pour les systèmes sur-déterminés et les systèmes sous-déterminés. Afin de s'en assurer, on peut tracer le gain en fonction des dimensions pour un conditionnement fixé.





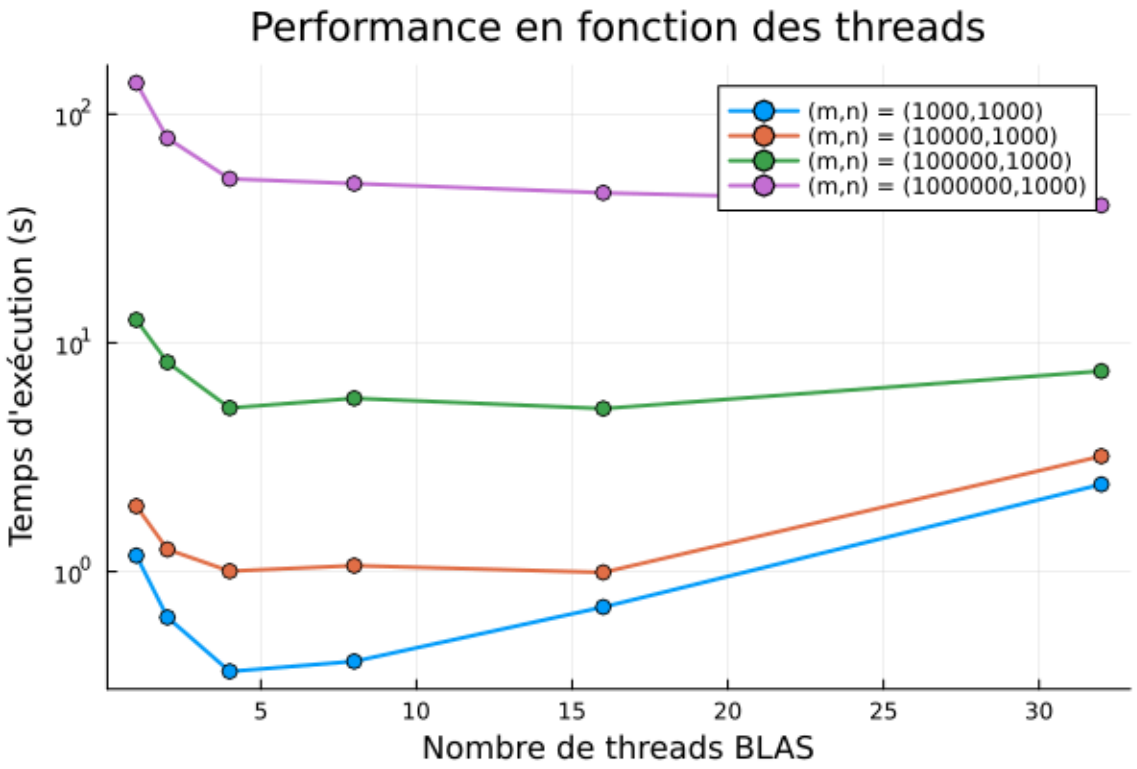
Deuxièmement, on voit que le gain ne dépend pas des dimensions de la matrice, mais bien de la taille et du conditionnement. En ce qui concerne le conditionnement, on peut voir que plus il devient élevé, plus il est préférable d'utiliser LSRN, car il apporte un gain en terme d'itération (à partir d'un conditionnement d'environ 70-80, LSRN peut devenir intéressant). Cependant, la taille montre qu'il faut que le système soit assez gros pour que LSRN soit rentable. En effet, peu importe le conditionnement, lorsque la matrice n'a que 10 colonnes ou 10 lignes, LSRN n'apporte aucun gain (même qu'il est préférable de ne pas appliquer LSRN) peu importe la taile de l'autre dimension. C'est seulement à partir de taile plus grande que 100 que LSRN devient intéressant. Pour m=1000, on voit vraiment un énorme gain avec l'ajout de LSRN. le comportement du gain en fonction du conditionnement plus en détail, on trace un graphique du gain en fonction du conditionnement pour voir la relation de manière plus détaillée.

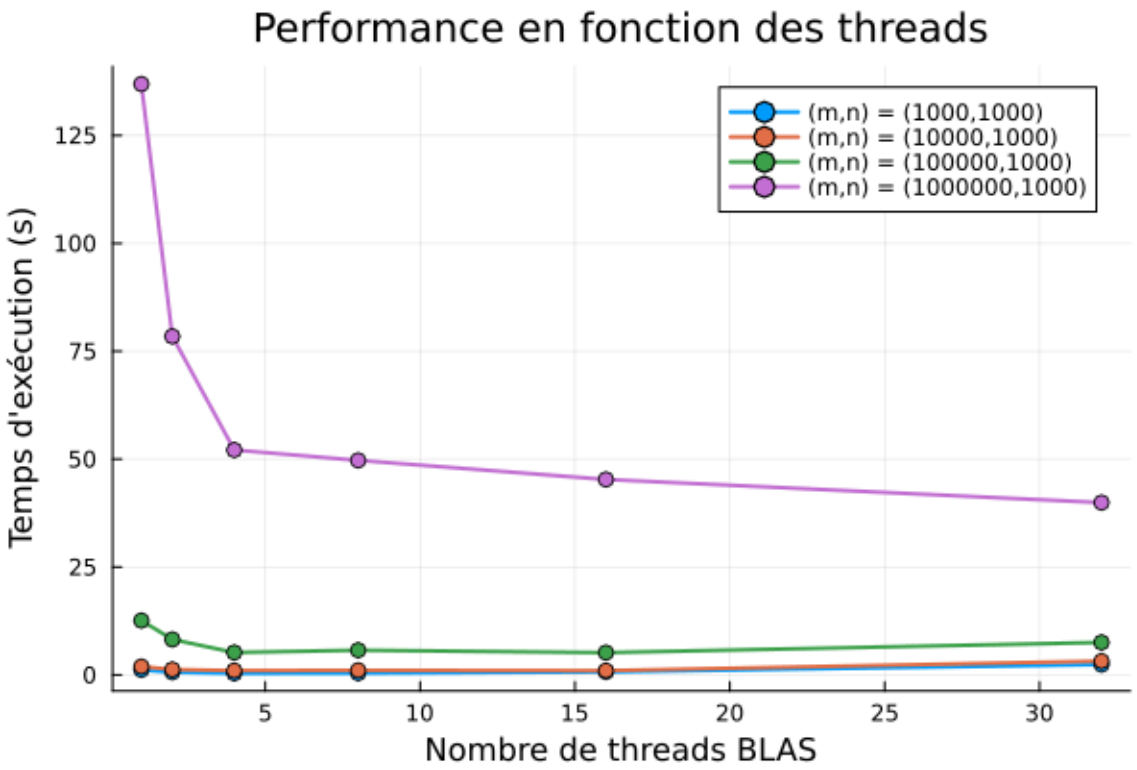




On voit que sous un seuil de conditionnement, on a un gain qui tend vers −1. Cela veut dire que LSRN nuit énormément à la résolution. Au dessus de ce seuil, on retrouve une relation de saturation. On voit qu'ici, il y a une saturation à un gain d'environ 0.7. Cela est du au fait que les courbes ont été tracées avec des dimension 1000x100 (des dimensions presque limites lorsqu'on souhaite utiliser LSRN dans des conditions optimales). Si on avait tracer cette même courbe sur des problèmes de dimensions 100000x1000, on aurait vu une saturation plus proche de 1.

On peut aussi vérifier la parallélisation de LSRN en regardant le temps de résolution de LSRN_LSQR en fonction du nombre de thread alloués à BLAS pour les calculs effectués.Afin de bien discute, deux graphiques seront produits: un avec l'axe y en échelle logarithmique, et l'autre non. On obtient les graphiques suivant:


Performance en fonction des threads
Temps d'exécution (s)
10^2
10^1
10^0
5
10
15
20
25
30
Nombre de threads BLAS
(m,n) = (1000,1000)
(m,n) = (10000,1000)
(m,n) = (100000,1000)
(m,n) = (1000000,1000)


Performance en fonction des threads
Temps d'exécution (s)
125
100
75
50
25
0
5
10
15
20
25
30
Nombre de threads BLAS
(m,n) = (1000,1000)
(m,n) = (10000,1000)
(m,n) = (100000,1000)
(m,n) = (1000000,1000)

On voit sur la première figure (échelle log) que toutes les courbes semblent initialement diminuer et après un certains nombre de threads, les temps de calcul recommence à augmenter. En regardant le graphique en échelle normale, on voit que pour des système de taille assez grande, le temps de calcul diminue toujours en augmentant le nombre de threads. Cela est un comportement typique lorsqu'on effectue des calcul en parralèle. En effet, cela montre que lorsqu'un problème est trop petit, alors le coût associée au parallélisme dépasse la gain qu'il peut apporter. Cependant à partir d'une certaine taille de problème, un nombre élevé de thread pourra faire gagner du temps de calcul. Cela étant dit, le parallélisme est bel et bien utile lors de l'utilisation de LSRN à condition de travailler sur des problèmes assez gros. Ce n'est pas vraiment un enjeux, car LSRN est intéressant pour des systèmes de grande taille.

5) Impact et recommandations

- **Quand utiliser LSRN** : problèmes fortement rectangulaires ou non(si l'algorithme de résolution est LSQR) avec grand $\kappa(A)$, où la projection aléatoire reste abordable. Dans ces régimes, LSRN offre des gains sur le nombre d'itérations et une plus grande robustesse aux mauvais conditionnements.
- **Quand s'en abstenir** : matrices bien conditionnées (conditionnement d'environ 70 et moins selon l'analyse et le type de problèmes utilisés); le coût du préconditionnement peut ne pas se rentabiliser.
- **Parallélisme** : fixer `BLAS.set_num_threads(k)` avant les expériences permet des comparaisons reproductibles. Sur machines multi-cœurs, laisser BLAS gérer le parallélisme est souvent suffisant; le parallélisme manuel n'apporte pas toujours un supplément puisque BLAS est déjà optimisé pour utiliser le parallélisme.

6) Perspective personnelle

Ce projet m'a permis de passer de la théorie (préconditionnement aléatoire et garanties de concentration) à une implantation concrète en Julia, avec une attention à la reproductibilité. LSRN s'avère une démonstration convaincante de la puissance des méthodes aléatoires en algèbre linéaire numérique moderne. Les pistes de prolongement incluent : matrices creuses réelles, LSMR en complément de LSQR, et une intégration avec des opérateurs linéaires implicites (sans matérialiser A afin de réduire l'allocation en mémoire et donc le temps de calcul).

7) Références

- X. Meng, M. A. Saunders, M. W. Mahoney (2014). *LSRN: A Parallel Iterative Solver for Strongly Over- or Under-Determined Systems*. SIAM Journal on Scientific Computing.


```
# phase3_code.jl
# %%
using Pkg
Pkg.activate("Phase3_projet")
Pkg.add(["LinearAlgebra", "SparseArrays", "Krylov", "BenchmarkTools",
"SuiteSparseMatrixCollection", "MatrixMarket"])
using LinearAlgebra, SparseArrays, Krylov, BenchmarkTools,
SuiteSparseMatrixCollection, MatrixMarket, Random, Plots
```

```
# %%
function badly_conditioned_rectangular_matrix(m, n, kappa)
    U, _ = qr(randn(m, n))
    V, _ = qr(randn(n, n))
    s = range(1.0, 1.0/kappa, length=n)
    S = Diagonal(s)
    A = U * S * V'
    return Matrix(A)
end

function badly_conditioned_underdetermined_matrix(m, n, kappa)
    U, _ = qr(randn(n, m))
    V, _ = qr(randn(m, m))
    s = range(1.0, 1.0 / kappa, length=m)
    S = Diagonal(s)
    A_tall = U * S * V'
    return Matrix(A_tall')[1:m, 1:n]
end

function lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
    m, n = size(A)
    s = ceil(Int, gamma * n)
    G = randn(s, m)
    Ã = G * A
    Ũ, Σ̃, Ṽ = svd(Ã; full=false)
    r = sum(Σ̃ .> 1e-12)
    Σinv = Diagonal(1.0 ./ Σ̃[1:r])
    V_r = Ṽ[:,1:r]
    N = V_r * Σinv
    AN = A * N
    ŷ, histo = lsqr(AN, b; atol=tol, btol=tol, itmax=itmax, history=true)

    x̂ = N * ŷ
    return x̂, histo
end

function lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10, itmax=2000)
    m, n = size(A)
    s = ceil(Int, gamma * m)
    G = randn(n, s)
    Ã = A * G
    Ũ, Σ̃, Ṽ = svd(Ã; full=false)
    r = sum(Σ̃ .> 1e-12)
    U_r = Ũ[:,1:r]
    Σ_r = Σ̃[1:r]
    Σinv = Diagonal(1.0 ./ Σ_r)
    M = U_r * Σinv
```

```
    Mt = M'
    At_pre = Mt * A
    bt_pre = Mt * b

    x̂, histo = lsqr(At_pre, bt_pre; atol=tol, btol=tol, itmax=itmax,
history=true)

    return x̂, histo
end

# %% [markdown]
# # Conditionnement après préconditionnement

# %%
function precondition_over(A; γ=2.0)
    m, n = size(A)
    s = ceil(Int, γ * n)
    G = randn(s, m)
    Ã = G * A
    Ũ, Σ̃, Ṽ = svd(Ã; full=false)
    r = sum(Σ̃ .> 1e-12)
    Σinv = Diagonal(1.0 ./ Σ̃[1:r])
    V_r = Ṽ[:, 1:r]
    N = V_r * Σinv
    AN = A * N
    return AN
end

function precondition_under(A; γ=2.0)
    m, n = size(A)
    s = ceil(Int, γ * m)
    G = randn(n, s)
    Ã = A * G
    Ũ, Σ̃, Ṽ = svd(Ã; full=false)
    r = sum(Σ̃ .> 1e-12)
    U_r = Ũ[:, 1:r]
    Σinv = Diagonal(1.0 ./ Σ̃[1:r])
    M = U_r * Σinv
    Mt = M'
    At_pre = Mt * A
    return At_pre
end

# %%
kappas = 10 .^ range(1, 7, length=9)
dims = [(100, 100), (1000, 100), (10000, 100)]
dims_under = [(100, 100), (100, 1000), (100, 10000)]
```

```
results = Dict()

for (m,n) in dims
    ys = Float64[]
    for κ in kappas
        A = badly_conditioned_rectangular_matrix(m, n, κ)
        AN = precondition_over(A)
        push!(ys, cond(AN))
    end
    results[("over", m, n)] = ys
end

for (m,n) in dims_under
    ys = Float64[]
    for κ in kappas
        A = badly_conditioned_underdetermined_matrix(m, n, κ)
        AN = precondition_under(A)
        push!(ys, cond(AN))
    end
    results[("under", m, n)] = ys
end

plt = plot(xscale=:log10,
    xlabel="Cond(A)", ylabel="Cond(A preconditionnée)",
    title="Amélioration du conditionnement")

for (m,n) in dims
    plot!(plt, kappas, results[("over", m, n)],
        lw=2, label="Surdéterminé m=$m, n=$n")
end

for (m,n) in dims_under
    plot!(plt, kappas, results[("under", m, n)],
        lw=2, linestyle=:dash, label="Sous-déterminé m=$m, n=$n")
end
savefig(plt, "Fig/cond_improvement.png")
savefig(plt, "Fig/cond_improvement.pdf")

# %% [markdown]
# # Système sur-déterminé

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio m/n allant de 1 à 1000 (avec m0=n0=1000)

# %%
ms = Int.(round.(vec(10 .^ range(3, 6, length=4))))
n = 1000
```

```
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ms))
for (j, m) in enumerate(ms)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("m           : $(m)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ms, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Ratio m/n",
    ylabel="Conditionnement",
    color=:viridis,
    title="Gain",
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_1000.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_1000.pdf")

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio m/n allant de 1 à 1000 (avec m0=n0=100)

# %%
ms = Int.(round.(vec(10 .^ range(2, 5, length=4))))
n = 100
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ms))
for (j, m) in enumerate(ms)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("m           : $(m)")
```

```
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ms, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Ratio m/n",
    ylabel="Conditionnement",
    color=:viridis,
    title="Gain",
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_100.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_100.pdf")

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio m/n allant de 1 à 1000 (avec m0=n0=10)

# %%
ms = Int.(round.(vec(10 .^ range(1, 4, length=4))))
n = 10
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ms))
for (j, m) in enumerate(ms)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("m           : $(m)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ms, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Ratio m/n",
    ylabel="Conditionnement",
    color=:viridis,
    title="Gain",
```

```
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_10.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_surdet_10.pdf")

# %% [markdown]
# ### Gain vs ratio m/n

# %%
ms = Int.(round.(vec(10 .^ range(2, 5, length=500))))
n = 100
kappa_vals = 10
gain = zeros(length(kappa_vals),length(ms))
for (j, m) in enumerate(ms)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("m           : $(m)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =plot(ms, vec(gain);
    xscale=:log10,
    xlabel="Ratio m/n",
    ylabel="Gain",
    color=:viridis,
    title="Gain en fonction du ratio m/n",
)
savefig(plt,"Fig/Gain_VS_ratio_surdet_100.png")
savefig(plt,"Fig/Gain_VS_ratio_surdet_100.pdf")

# %% [markdown]
# ### Gain vs conditionnement

# %%
ms = 10000
n = 100
kappa_vals = vec(10 .^ range(0, 2, length=200))
gain = zeros(length(kappa_vals),length(ms))
for (j, m) in enumerate(ms)
    for (i, kappa) in enumerate(kappa_vals)
```

```
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("m           : $(m)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =plot(kappa_vals, vec(gain);
    xscale=:log10,
    xlabel="Conditionnement",
    ylabel="Gain",
    color=:viridis,
    title="Gain en fonction du conditionnement",
)
savefig(plt,"Fig/Gain_VS_condi_surdet_100-10000.png")
savefig(plt,"Fig/Gain_VS_condi_surdet_100-10000.pdf")


# %% [markdown]
# # Système sous-déterminé

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio n/m allant de 1 à 1000 (avec m0=n0=10)

# %%
m = 10
ns = Int.(round.(vec(10 .^ range(1, 4, length=4))))
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ns))
for (j, n) in enumerate(ns)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_underdetermined_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10,
itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("n           : $(n)")
        println("Kappa           : $(kappa)")
```

```
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ns, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Nombre de colonnes",
    ylabel="Conditionnement",
    color=:viridis,
    title="Niter/Niter_LSRN",
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_10.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_10.pdf")

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio n/m allant de 1 à 1000 (avec m0=n0=100)

# %%
m = 100
ns = Int.(round.(vec(10 .^ range(2, 5, length=4))))
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ns))
for (j, n) in enumerate(ns)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_underdetermined_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10,
itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("n           : $(n)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ns, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Nombre de colonnes",
    ylabel="Conditionnement",
    color=:viridis,
    title="Niter/Niter_LSRN",
```

```
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_100.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_100.pdf")

# %% [markdown]
# ### ColorMap de Gain en focntion du conditionnement allant de 1 à 1000000 et
ratio n/m allant de 1 à 1000 (avec m0=n0=1000)

# %%
m = 1000
ns = Int.(round.(vec(10 .^ range(3, 6, length=4))))
kappa_vals = vec(10 .^ range(0, 6, length=7))
gain = zeros(length(kappa_vals),length(ns))
for (j, n) in enumerate(ns)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_underdetermined_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10,
itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("n           : $(n)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =heatmap(ns, kappa_vals, gain;
    xscale=:log10,
    yscale=:log10,
    xlabel="Nombre de colonnes",
    ylabel="Conditionnement",
    color=:viridis,
    title="Niter/Niter_LSRN",
    clim=(-1, 1)
)
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_1000.png")
savefig(plt,"Fig/Gain_VS_Condi-ratio_sousdet_1000.pdf")

# %% [markdown]
# ### Gain vs ratio m/n

# %%
m = 100
ns = Int.(round.(vec(10 .^ range(2, 5, length=500))))
```

```
kappa_vals = 10
gain = zeros(length(kappa_vals),length(ns))
for (j, n) in enumerate(ns)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_underdetermined_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10,
itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("n           : $(n)")
        println("Kappa           : $(kappa)")
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =plot(ns, vec(gain);
    xscale=:log10,
    xlabel="Ratio n/m",
    ylabel="Gain",
    color=:viridis,
    title="Gain en fonction du ratio n/m",
)
savefig(plt,"Fig/Gain_VS_ratio_sousdet_100.png")
savefig(plt,"Fig/Gain_VS_ratio_sousdet_100.pdf")

# %% [markdown]
# ### Gain vs conditionnement

# %%
m = 100
ns = 100000
kappa_vals = vec(10 .^ range(0, 2, length=200))
gain = zeros(length(kappa_vals),length(ns))
for (j, n) in enumerate(ns)
    for (i, kappa) in enumerate(kappa_vals)
        A = badly_conditioned_underdetermined_matrix(m, n, kappa)
        x = randn(n)
        b = A * x
        res1, hist1 = lsqr(A, b; atol=1e-10, btol=1e-10, itmax=2000,
history=true)
        x_prec, hist2 = lsrn_lsqr_underdetermined(A, b; gamma=2.0, tol=1e-10,
itmax=2000)
        rationiter = hist2.niter/hist1.niter
        println("n           : $(n)")
        println("Kappa           : $(kappa)")
```

```
        println("  ")
        gain[i,j] = max(1 - rationiter,-1)
    end
end
plt =plot(kappa_vals, vec(gain);
    xscale=:log10,
    xlabel="Conditionnement",
    ylabel="Gain",
    color=:viridis,
    title="Gain en conditionnement",
)
savefig(plt,"Fig/Gain_VS_condi_sousdet_100-10000.png")
savefig(plt,"Fig/Gain_VS_condi_sousdet_100-10000.pdf")

# %% [markdown]
# # Parallélisme

# %%
ms = Int.([1000 10000 100000 1000000])
n = 1000
kappa = 70
threads_list = [1, 2, 4, 8, 16, 32]
times = zeros(length(threads_list),length(ms))
for (j,m) in enumerate(ms)
    for (i, t) in enumerate(threads_list)
        BLAS.set_num_threads(t)
        println("Nombre de threads : $t")
        A = badly_conditioned_rectangular_matrix(m, n, kappa)
        x_true = randn(n)
        b = A * x_true

        lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)

        t_exec = @elapsed lsrn_lsqr(A, b; gamma=2.0, tol=1e-10, itmax=2000)
        times[i,j] = t_exec
        println("Temps d'exécution : $(round(t_exec, digits=4)) sec")
    end
end
plt = plot(
    threads_list, times[:,1],
    marker=:o, lw=2,
    label="(m,n) = (1000,1000)",
    xlabel="Nombre de threads BLAS",
    ylabel="Temps d'exécution (s)",
    title="Performance en fonction des threads",
    legend=true
)
for i=2:length(ms)
```

```
    plot!(plt, threads_list, times[:,i],marker=:o,
        lw=2, label="(m,n) = ($(ms[i]),1000)")
end
savefig(plt,"Fig/Time_VS_Threads.png")
savefig(plt,"Fig/Time_VS_Threads.pdf")

# %%
plt = plot(
    threads_list, times[:,1],
    marker=:o, lw=2,
    label="(m,n) = (1000,1000)",
    xlabel="Nombre de threads BLAS",
    ylabel="Temps d'exécution (s)",
    title="Performance en fonction des threads",
    legend=true
)
for i=2:length(ms)
    plot!(plt, threads_list, times[:,i],marker=:o,
        lw=2, label="(m,n) = ($(ms[i]),1000)")
end
savefig(plt,"Fig/Time_VS_Threads.png")
savefig(plt,"Fig/Time_VS_Threads.pdf")
```

“““