TME n° 5

Ryan Lahfa

22 décembre 2019

Table des matières

Table des figures

Exercice 5

Implémentation des algorithmes

Tout d'abord, on implèmente les fonctions, afin d'essayer d'obtenir des schémas assez stables, on utilisera un critère d'arrêt lorsque: $\|x_k - x_{k+1}\| < \varepsilon$ pour un ε fixé, sinon, si après un certain nombre d'itérations (5000 par exemple), les différences consécutives en norme ne deviennent pas assez petits (cas de non-convergence ou convergence très lente), on s'arrête avant.

De même, si à un moment, notre itération n'est plus un nombre, on s'arrête.

On implémente les fonctions à la suite en se référant au cours muni de la technique de détection de (non-)convergence expliqué au dessus.

```
function x = Jacobi(A, y, x0, tol)
    n = length(y);
    x = x0;
    xold = x;
    itmax = 5000;
    divcounter = 1;
    it = 1;
    while (divcounter <= itmax && norm(xold - x) > tol) || it == 1
        for i=1:n-1
            x(i) = (y(i) - A(i,[1:i - 1, i + 1:n]) * xold([1:i-1, i+1:n])')/A(i,i);
        end
        if norm(xold - x) < tol
            divcounter = 0;
        end
        xold = x;
        divcounter = divcounter + 1;
        it = it + 1;
    end
end

function x = GS(A, y, x0, tol)
    n = length(y);
    x = x0;

    itmax = 5000;
    divcounter = 1;
    it = 1;

    xold = x;

    while (divcounter <= itmax && norm(xold - x) > tol) || it == 1
        for i=1:n
            x(i) = (y(i) - A(i, 1:i -1)*x(1:i-1)' - A(i, i + 1:n) * x(i + 1:n)') / A(i, i);
        end
        if norm(xold - x) < tol
            divcounter = 0;
        end
        xold = x;
        divcounter = divcounter + 1;
        it = it + 1;
    end
end
```

```
function x = SOR(A, y, x0, w, tol)
    n = length(y);
    x = x0;

    itmax = 5000;
    divcounter = 1;
    it = 1;

    xold = x;

    while (divcounter <= itmax && norm(xold - x) > tol) || it > 1
        for i=1:n
            x(i) = w*(y(i) - sum(A(i, 1:i-1).*x(1:i-1)) - sum(A(i, i+1:n).*x(i + 1:n))) / A(i, i) + (1 - w)*x(i)
        end

        if norm(xold - x) < tol
            divcounter = 0;
        end
        xold = x;
        divcounter = divcounter + 1;
        it = it + 1;
    end
end

function x = GCJ(A, y, x0)
    n = length(y);
    r = y - A*(x0');
    p = r;
    x = x0;
    for k=0:n-1
        a = (p')*r / (p' * A * p);
        x = x + a*p;
        r = r - a*A*p;
        beta = - (p' *A*r) / (p' * A * p);
        p = r + beta*p;
    end
end
```

Génération des matrices

À présent, on se donne quatre types de matrice: I_n, matrices à diagonale dominante, matrices de Vandermonde de support $[[1, n]]$ (nombre de conditionnement très élevé) et symétriques définies positives, voici le code de génération:

```
% Matrices très mal conditionnés
function [A, b] = generate_ill_conditionned_matrix(n)
    A = vander(1:n); % Vandermonde([[1, n]])
    x = ones(n, 1);
```

```
    b = A*x;
end

% Diagonally dominant matrix (proven convergence for Jacobi)
function [A, b] = generate_nice_matrix_for_jacobi(n)
    x = 0.05; % part de zéros hors de la diagonale
    k = round(n*(n-1)*x);

    data = randn(n*(n-1)-k,1);
    data = [data;zeros(k,1)];
    data = data(randperm(length(data)));

    diag_index = 1:n+1:n*n;
    offd_index = setdiff(1:n*n,diag_index);
    A = zeros(n,n);
    A(offd_index) = data;
    A(diag_index) = sum(A,1);

    b = A*ones(n, 1);
end

% Identity matrices
function [A, b] = identity_matrix(n)
    A = eye(n);
    b = A*on
end

% SDP matrices
function [A, b] = generate_sdp(n)
    A = rand(n, n);
    A = (1/2)*(A + A');
    A = A + n*eye(n);
    b = A*ones(n,1);
end
```

On génère des problèmes (A, b) où $b = A1$ où 1 est le vecteur rempli de 1.

Tests de performance / efficacité

Puis, on génère des courbes de test de performance: temps d'exécution et norme de l'erreur avec ce code:

```
MAX_N = 3000;
MIN_N = 100; % < 100, on peut utiliser du pivot donc c'est pas très intéressant.
STEP = 100; % saut de la taille des matrices
w0 = 0.5; % valeur pour SOR < 1 (marche plutôt bien pour des matrices SDP)
tol = 1; % tolérance de tous les algorithmes
```

```
plot_performance_by_generation("I_n", @identity_matrix, MIN_N, MAX_N, STEP, w0, tol);
plot_performance_by_generation("Matrices à diagonale dominante", @generate_nice_matrix_for_jacobi, MIN_N, MAX_N,
plot_performance_by_generation("Matrice très mal conditionnées", @generate_ill_conditionned_matrix, MIN_N, MAX_N
plot_performance_by_generation("Matrice symétriques définies positives", @generate_sdp, MIN_N, MAX_N, STEP, w0,


function plot_performance_by_generation(test_name, g, start, max_N, step, w0, tol)
    J_ = zeros((max_N - start)/step + 1, 2);
    GS_ = zeros((max_N - start)/step + 1, 2);
    SOR_ = zeros((max_N - start)/step + 1, 2);
    GCJ_ = zeros((max_N - start)/step + 1, 2);

    R = start:step:max_N;

    fprintf("Computing all the values...\n");
    for q = R
        [A, b] = g(q);
        x0 = zeros(1,q);
        [jacobi, gs, sor, gcj] = test_algorithms(A, b, x0, w0, tol);
        j = ((q - start)/step) + 1;
        J_(j,:) = jacobi;
        GS_(j,:) = gs;
        SOR_(j,:) = sor;
        GCJ_(j,:) = gcj;
        fprintf("%d/%d computed.\n", q, max_N);
    end

    fprintf("Algorithms computed. Drawing now.\n")
    figure
    tiledlayout(2,1)

    for k = 1:2
        nexttile
        hold on
        plot(R, J_(:,k));
        plot(R, GS_(:,k));
        plot(R, SOR_(:,k));
        plot(R, GCJ_(:,k));
        lgd = legend('Jacobi', 'Gauss Seidel', 'SOR', 'Conjugate gradient');
        xlabel('Taille de la matrice');
        if k == 1
            title(lgd, test_name + " - norme de l'erreur en fonction de la taille de la matrice");
            ylabel("Norme 2 de l'erreur entre la solution et l'approximation");
        else
            title(lgd, test_name + " - temps d'exécution en fonction de la taille de la matrice");
            ylabel('Temps en seconde');
```

```
        end
    end
end



function [jacobi, gs, sor, gcj] = test_algorithms(A, b, x0, w0, tol)
    tic
    x_jacobi_approx = Jacobi(A, b, x0, tol);
    jacobi_time = toc;
    jacobi = [compute_approx_error(x_jacobi_approx) jacobi_time];

    tic
    x_gs_approx = GS(A, b, x0, tol);
    gs_time = toc;
    gs = [compute_approx_error(x_gs_approx) gs_time];

    tic
    x_sor_approx = SOR(A, b, x0, w0, tol);
    sor_time = toc;
    sor = [compute_approx_error(x_sor_approx) sor_time];

    tic
    x_gcj_approx = GCJ(A, b, x0);
    gcj_time = toc;

    gcj = [compute_approx_error(x_gcj_approx) gcj_time];
end

function e = compute_approx_error(x_approx)
    n = length(x_approx);
    e = norm(ones(n, 1) - x_approx);
end
```

Résultats & Interprétation

À présent, nous allons interpréter les résultats.

Avant-propos

Tout d'abord, rappelons que ces algorithmes ont pour but d'être plus rapide qu'un pivot classique en $O(n^3)$, ils sont efficaces sur des matrices parcimonieuses. [1] Leur convergence est démontrée dans certains cas, mais elle n'est que théorique, elle suppose que les erreurs numériques n'existent pas. Donc il est fort probable qu'on rencontre des cas d'erreurs numériques qui rendraient impossible la convergence, révélant des schémas numériquement instables ou très sensibles aux erreurs dûes à la norme IEE754.

1. dit aussi sparse.

Matrices I_n (figure 1)

Naturellement, les matrices I_n sont un très bon exemple de matrices parcimonieuses, elles n'ont que n éléments non nuls sur n^2, leur conditionnement est très faible. Ainsi, on teste dans la figure 1 tous les algorithmes sur les matrices I_n.

On observe que le gradient conjugué, Gauss-Seidel et Jacobi [2] donnent toujours le résultat exact sans problème avec un des meilleurs temps, seul SOR est plus rapide que le gradient conjugué avec une erreur qui croît en $O(\log n)$.

En termes d'efficacité néanmoins, la convergence de Jacobi et Gauss-Seidel est fonction de n, tandis que celle de SOR et du gradient conjugué ne l'est pas. D'ailleurs, il semblerait que SOR soit un petit plus rapide que le gradient conjugué en pratique, mais celui commettant des erreurs sur un exemple aussi simple que I_n le rend inapproprié dans ce cas.

On peut facilement imaginer que SOR est très dépendant de ω (notamment pour sa convergence mais aussi pour sa précision), malheureusement, effectuer une recherche d'ω par dessus ne permet pas de comparer de façon égale avec les autres algorithmes.

De plus, dans (Hackbusch 1994), on prouve que le paramètre de relaxation optimal est de la forme:

$$\omega_{\text{opt}} = 1 + \left(\frac{\rho(I_n - D^{-1}A)}{1 + \sqrt{1 - \rho(I_n - D^{-1}A)}} \right)^2$$

Où ρ est le rayon spectrale et $I_n - D^{-1}A$ est la matrice d'itération de la méthode de Jacobi.

Ainsi, son calcul requiert de calculer le rayon spectral, ce qui pourrait se faire aussi par un algorithme d'itération (puissance itérée en se basant sur la théorie de la réduction de Jordan).

Matrices à diagonale (strictement) dominante (figure 2)

Déjà, l'on peut prouver que la méthode de Jacobi, Gauss-Seidel convergent face à ce genre de matrices, cependant, remarquons que la méthode dont on génère ces matrices fait en sorte qu'elles ne soient pas très parcimonieuses, 95% d'éléments non nuls hors de la diagonale.

On s'attend donc à une performance correcte sur ces matrices.

Or, on voit que SOR est parfait sur ce problème, tandis que Gauss-Seidel a des erreurs énormes de temps en temps sur des grosses matrices: on peut soupçonner des erreurs numériques, mais reste parfait aussi la majeure partie du temps.

Aussi, Jacobi a des erreurs en $10^k, k \in \{0, 1, 2\}$ en moyenne, ce qui ne peut pas se voir sur la graphe pour des raisons d'échelles.

En revanche, ceci n'est pas visible sur le graphe mais le gradient conjugué échoue complètement sur ces matrices et donne en sortie des `NaN`, on peut commencer à conjecturer que le gradient conjugué est très sensible aux erreurs numériques, j'ai essayé de modifier l'algorithme pour voir si cela pouvait changer quelque chose (éviter des éliminations catastrophiques par exemple), mais cela ne suffisait pas.

Quant à l'efficacité, on voit que, non seulement, le gradient conjugué ne donne aucun résultat, mais en plus, il est parmi les plus lents.

2. On ne les voit pas vraiment sur le graphe, mais les résultats montrent des normes nuls.

Ensuite, Jacobi est plus lent que Gauss-Seidel bien que Gauss-Seidel est plus exact mais a des erreurs catastrophiques tandis que Jacobi a des erreurs contrôlés. Enfin, on remarque que SOR est d'une rapidité hallucinante, mais comme indiqué plus haut, le paramètre ω joue un rôle non négligeable dans la vitesse de convergence (il a été pris $\omega = 0.4$, connu pour avoir de bonnes performances sur les matrices symétriques définies positives).

Matrices très mal conditionnées (figure 3)

On remarquera que ces matrices ne sont généralement pas très parcimonieuses, mais il était intéressant de tester quand même à quel point les résultats seraient catastrophiques.

Contre toute attente, SOR parvient à avoir des erreurs à deux chiffres.

Mais toutes les autres méthodes échouent avec des `NaN` ou alors avec des erreurs en 10^{199}, ce qui est rassurant.

En efficacité, on voit aussi que SOR est extrêmement rapide à nouveau tout en donnant une réponse de bonne facture, tandis que les autres méthodes qui échouent croissent en fonction de n à l'exception du gradient conjugué qui doit certainement détecter `NaN` tôt et sortir de l'algorithme.

Matrices symétriques définies positives (figure 4)

Pour ces matrices aussi, on dispose de résultat de convergence théorique fort pour Jacobi, Gauss-Seidel, SOR ($\omega < 1$). Mais aussi, c'est les seules matrices sur lequel le gradient conjugué est supposé avoir de bon résultats en théorie.

On observe que Gauss-Seidel est plus performant que Jacobi et que SOR lui-même, mais les erreurs étant relativement faibles, ces comparaisons ne sont pas très intéressantes.

Mieux, on peut conjecturer de l'allure des courbes que l'erreur avec la méthode de SOR croît plus rapidement que l'erreur avec la méthode de Gauss-Seidel qui semble rester relativement stable. Ce qui nous laisse penser que Gauss-Seidel est non seulement plus précis mais ses erreurs sont stables même avec de grandes matrices.

Hélas, on constate que le gradient conjugué a une performance très mauvaise, en pratique, sur plusieurs exécutions, j'ai pu constater qu'il y a deux scénarios: solution parfaite ou `NaN` selon la matrice. Comme on peut le voir, en moyenne, on constate plus de `NaN`. La conjecture effectuée plus haut concernant le caractère sensible aux erreurs numériques du gradient conjugué se confirme.

En cherchant, il me paraît intéressant d'appliquer quelques opérations sur la matrice afin de stabiliser le gradient conjugué, certaines méthodes sont expliquées en détails dans (Knyazev et Lashuk 2007).

Enfin, à nouveau sur l'efficacité, on observe que le gradient conjugué est très lent tandis que tous les autres algorithmes sont très rapides. En pratique, en zoomant, on voit que SOR est plus rapide que Gauss-Seidel qui est lui-même plus rapide que Jacobi.

Ce qui est intéressant puisque SOR est le moins précis, mais Gauss-Seidel est ls plus précis et seulement le 2ème le plus lent.

Conclusion

Avec tous ces tests, on peut établir que Gauss-Seidel semble être l'algorithme le plus pratique parmi tous ceux là, il a une bonne stabilité et une vitesse acceptable.

En revanche, il semblerait qu'un SOR bien paramétré serait plus efficace en temps que Gauss-Seidel au prix de la précision.

Enfin, le gradient conjugué est intéressant mais trop sensible, il serait intéressant d'implémenter les différentes améliorations mentionnés, voire d'essayer l'algorithme du gradient biconjugué qui se généralise aux matrices non symétriques.

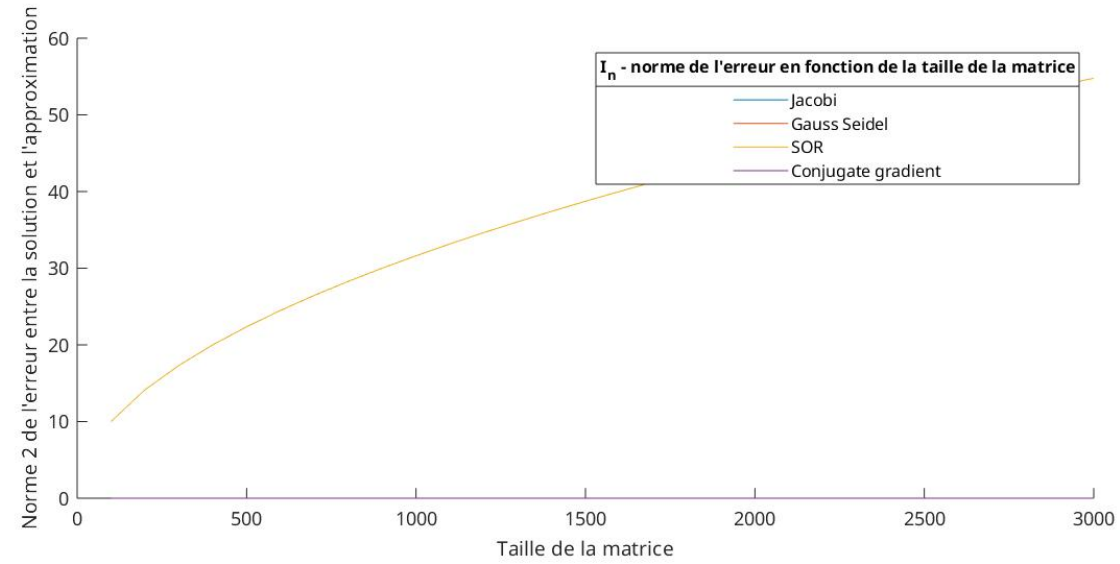
Enfin, Jacobi est un algorithme qui est toujours un petit plus lent que tous les autres avec des précisions médiocres. Ce qui fait de lui un bon étalon pour les autres algorithmes.

Il serait aussi très intéressant d'ajouter à SOR la recherche du rayon spectral pour voir à quel point cela le rend plus lent et moins performant que sa version fixe.

Références bibliographiques

Hackbusch, Wolfgang. 1994. *Iterative solution of large sparse systems of equations.* Vol. 95. Springer.

Knyazev, Andrew V, et Ilya Lashuk. 2007. « Steepest descent and conjugate gradient methods with variable preconditioning ». *SIAM Journal on Matrix Analysis and Applications* 29 (4): 1267-80.



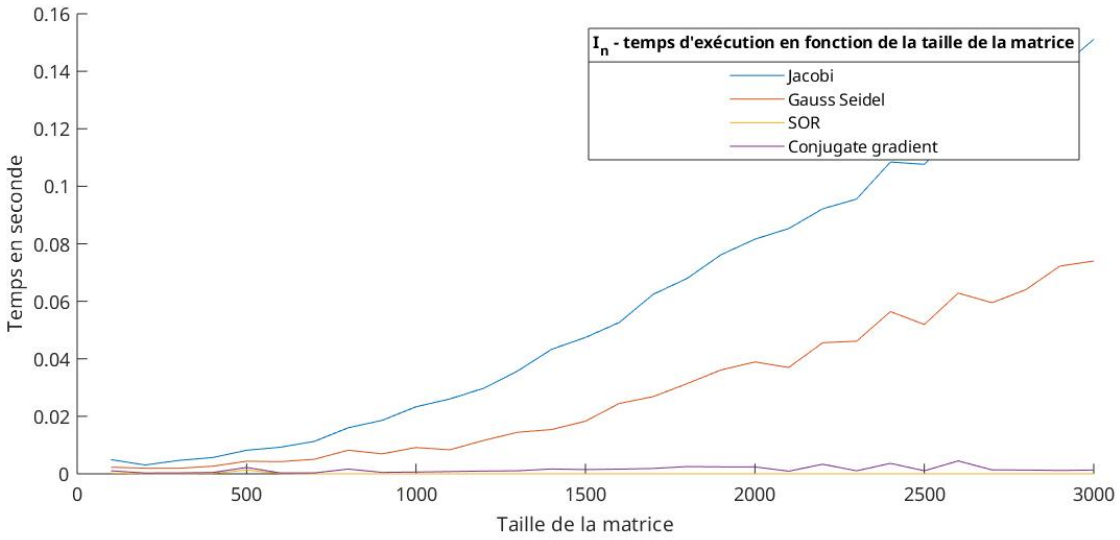
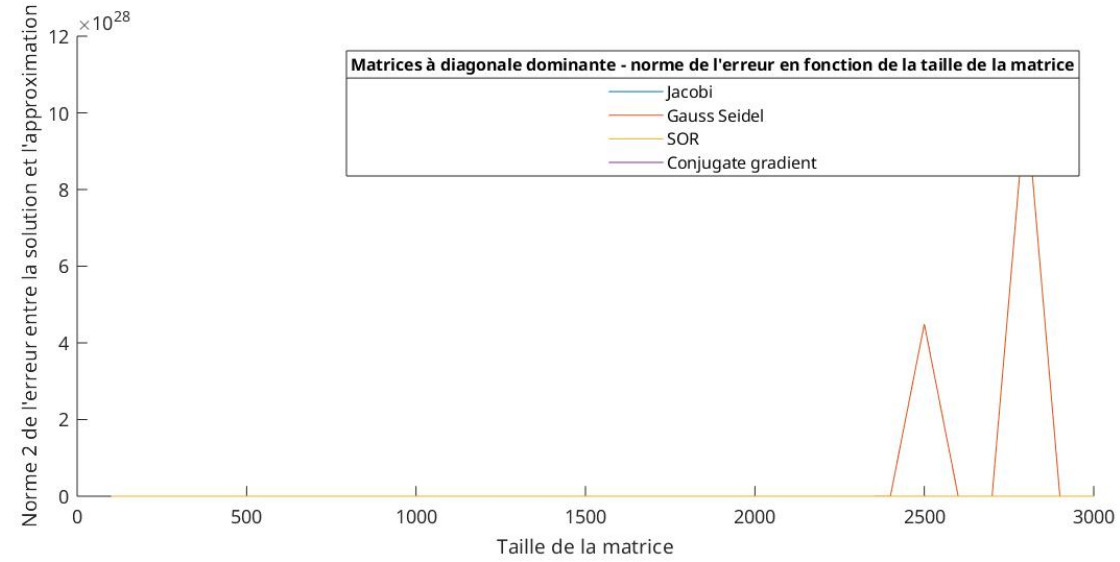


FIGURE 1 – Matrices I_n pour $n \in [[100, 3000]]$



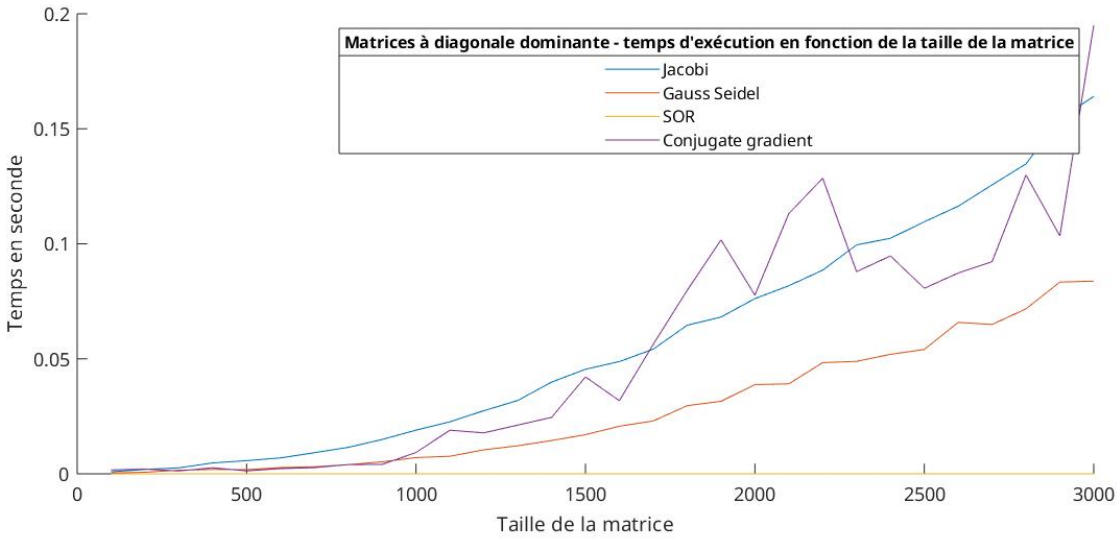
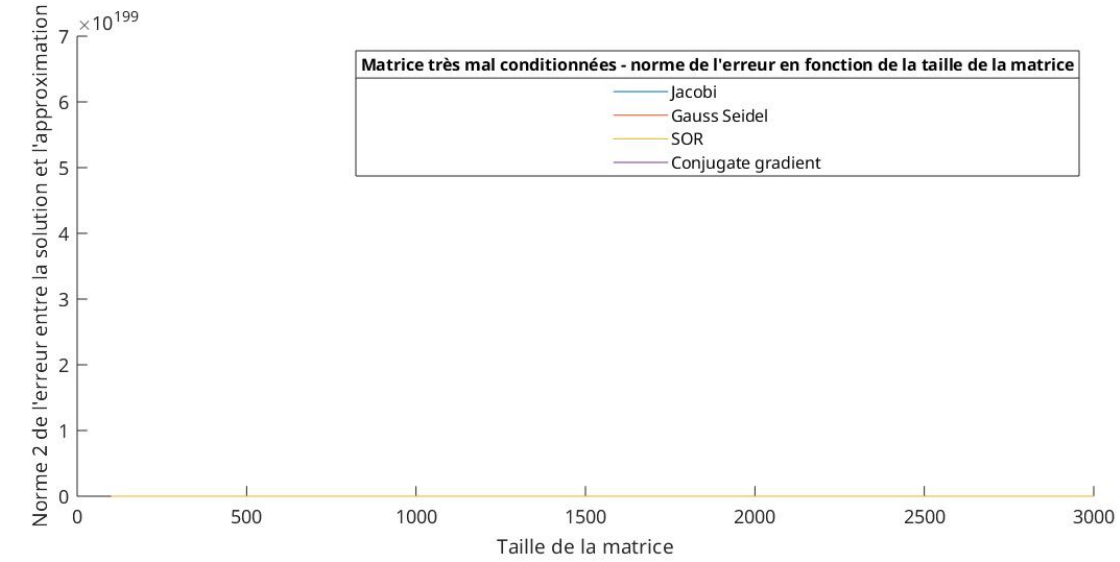


FIGURE 2 – Matrices à dominante diagonale (avec 5% de zéros hors de la diagonale en moyenne) pour $n \in [[100, 3000]]$



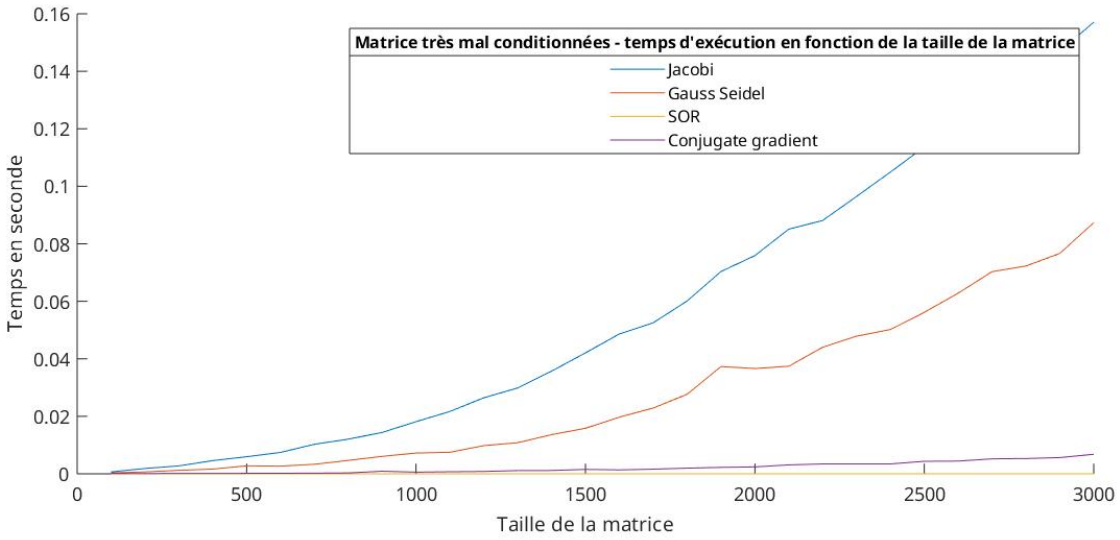
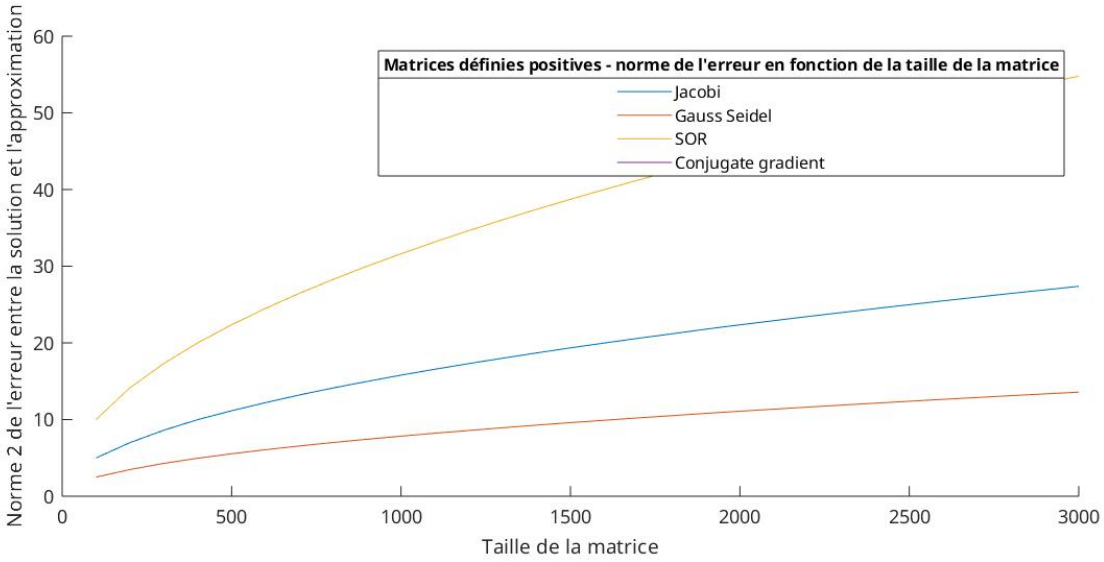


FIGURE 3 – Matrices très mal conditionnées (de Vandermonde de support $[[1, n]]$) pour $n \in [[100, 3000]]$



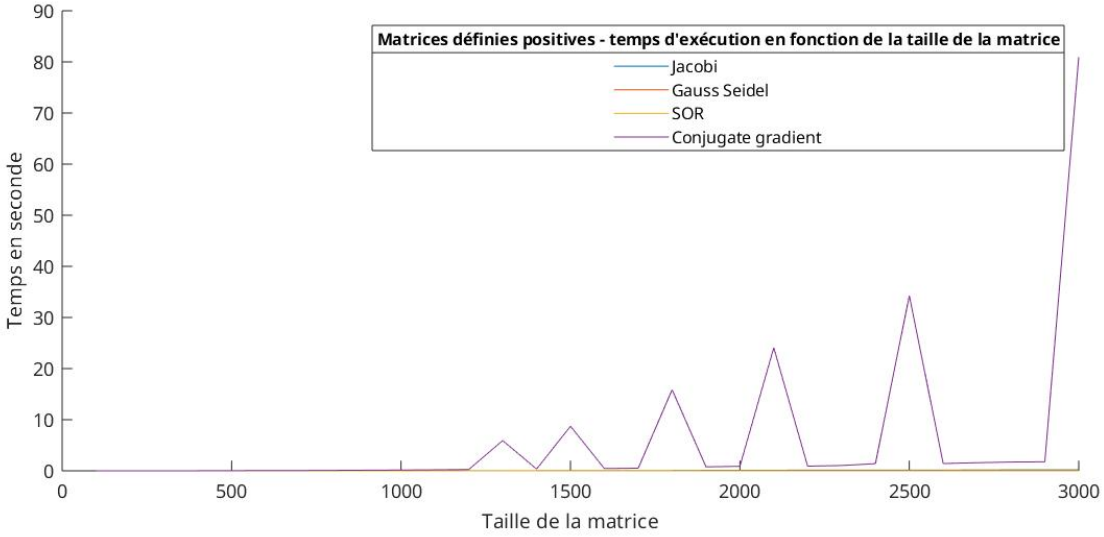


FIGURE 4 – Matrices symétriques définies positives pour $n \in [[100, 3000]]$