I) Limitations of the power method

Q1)

Pour les matrices de type 1:

Pour une matrice de taille 200 x 200 : eig : 0,02s et power v11: pourcentage de 4% non atteint avec 20 valeurs propres

Pour les matrices de type 2:

Pour une matrice de taille 200 x 200 : eig : 0,01s et power v11: 0,09s

Pour une matrice de taille 500 x 500: eig : 0,05s et power v11 : 1,65s

Pour une matrice de taille 1000 x 1000: eig : 0,21s et power v11 : pourcentage de 4% non atteint avec 20 valeurs propres

Pour les matrices de type 3:

Pour une matrice de taille 200 x 200 : eig : 0,02s et power v11: 0,18s

Pour une matrice de taille 500 x 500: eig : 0,05s et power v11 : pourcentage de 4% non atteint avec 20 valeurs propres

Pour les matrices de type 4:

Pour une matrice de taille 200 x 200 : eig : 0,01s et power v11: pourcentage de 4% non atteint avec 20 valeurs propres

On remarque ainsi que l'algorithme power_v11 est moins efficace que la fonction eig de matlab.

Q2)

Nouvel algorithme:

```
y = A *  v
B = transpose(v) * y
repeat
        v = y / norm(y)
        Bold = B
        y = A *  v
        B = transpose(v) * y
until |B - Bold| / |B| < eps
lambda 1 = v  and  v1 = v
```

Pour les matrices de type 2:

Pour une matrice de taille 500 x 500: power v11 : 1,65s et power v12: 0,9s

Pour une matrice de taille 700 x 700: power v11 : 3,7s et power v12: 2,2s

Pour une matrice de taille 700 x 700: power v11 : 9s et power v12: 4,8s

Notre nouvel algorithme est ainsi environ 2 fois plus rapide pour des matrices sensiblement grandes (entre 500 x 500 et 800 x 800)

Q3)
L'algorithme donné ne calcule les valeurs et vecteurs propres qu'un à un tandis qu'il est possible de calculer un nombre de valeurs propres souhaitées (utiles tandis que certaines ne le sont pas).

II) Extending the power method to compute dominant eigenspace vectors

Q4)
V converge vers une matrice telle que les vecteurs la formant convergent vers un vecteur associé à la plus grande valeur propre de notre matrice A.

Q5)
En inspectant les dimensions de H, on se rend compte que A est de dimensions n x n, V de dimensions n x m, alors Y aussi, et alors le produit de VT * A * V est de dimensions m x m donc H ne possède que m valeurs propres. Par ailleurs, ces m valeurs propres sont les m premières de A du fait du caractère orthogonal de V, et ne divergent pas (conservent même leur valeur) grâce au caractère normalisé de V.

Q6)
c.f. code

Q7)

```
%% repeat until ( PercentReached > PercentTrace or nev = m or k > MaxIter )
while (~conv & k < maxit),
        %% k <- k + 1
        k = k+1;
        %% Y <- A*V
        Y = A*Vr;
        %% orthogonalisation
        Vr = mgs(Y);

        %% Projection de Rayleigh-Ritz
        [Wr, Vr] = rayleigh_ritz_projection(A, Vr);

        %% Quels vecteurs ont convergé à cette itération
        analyse_cvg_finie = 0;
        % nombre de vecteurs ayant convergé à cette itération
        nbc_k = 0;
        % nb_c est le dernier vecteur à avoir convergé à l'itération précédente
        i = nb_c + 1;

        while(~analyse_cvg_finie),
        % tous les vecteurs de notre sous-espace ont convergé
        % on a fini (sans avoir obtenu le pourcentage)
        if(i > m)
                analyse_cvg_finie = 1;
        else
```

```
            % est-ce que le vecteur i a convergé

            % calcul de la norme du résidu
            aux = A*Vr(:,i) - Wr(i)*Vr(:,i);
            res = sqrt(aux'*aux);

            if(res >= eps*normA)
            % le vecteur i n'a pas convergé,
            % on sait que les vecteurs suivants n'auront pas convergé non plus
            % => itération finie
            analyse_cvg_finie = 1;
            else
            % le_vecteur i a convergé
            % un de plus
            nbc_k = nbc_k + 1;
            % on le stocke ainsi que sa valeur propre
            W(i) = Wr(i);

            itv(i) = k;

            % on met à jour la somme des valeurs propres
            eigsum = eigsum + W(i);

            % si cette valeur propre permet d'atteindre le pourcentage
            % on a fini
            if(eigsum >= vtrace)
            analyse_cvg_finie = 1;
            else
            % on passe au vecteur suivant
            i = i + 1;
            end
            end
        end
        end
```

III) subspace iter v2 and subspace iter v3: toward an efficient solver

Q8)
Le coût de calcul de A^p est de 2 * n^2p flops
On pourrait simplement calculer A^p puis multiplier cela par V mais ce serait une perte de temps car en effet, cela revient à faire quelque chose de la sorte: (A*A*A*A…) * V tandis que faire A*V (de dimension n*m) puis tout multiplier par A encore p - 1 fois revient à un coût de 2 * n*m^p flops, avec m < n. Cette méthode est donc plus rapide en temps de calcul.

Exemple pour p = 3:
((A*A)*A)*V -> 2 multiplications de matrices de tailles n*n et n*n et une multiplication n*n avec une n*m

A*(A*(A*V)) -> 3 multiplications de matrices de taille n*n avec une n*m

Q9)
c.f. code

Q10)
Pour les matrices de type 2, pour une matrice de taille 500 x 500:
avec p = 2: iter_v1 : 0,11s et iteer_v2: 0,07s
avec p = 3: iter_v1 : 0,11s et iteer_v2: 0,05s
avec p = 5: iter_v1 : 0,11s et iteer_v2: 0,05s
avec p = 10: iter_v1 : 0,11s et iteer_v2: 0,05s
avec p = 20: iter_v1 : 0,11s et iteer_v2: 0,03s
Mais à partir du dépassement du nombre d'itérations maximales qu'il aurait fallu pour l'algorithme iter_v1, notre algorithme est plus lent du fait d'itérations inutiles de calculs.

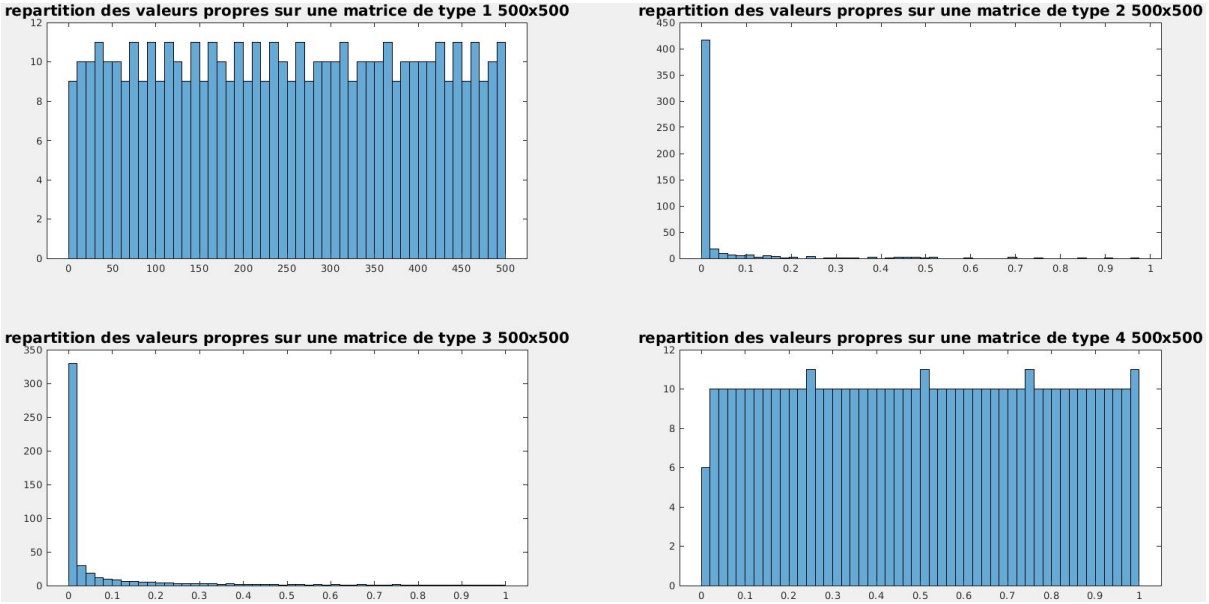
Q11)
On observe effectivement une précision très différentes pour chaque vecteur, allant par exemple de l'ordre de 10^-15 à 10^-6 pour une matrice 1000x1000 de type 2.
Cela s'explique par le fait que même si un vecteur est considéré comme ayant une précision suffisante, le programme va continuer à affiner la précisions de tous les vecteurs tant que tous ne sont pas considérés comme ayant convergé vers un résultat satisfaisant.

Q12)
La fonction subspace_iter_v3 se sert de l'observation faite à la Q12 pour améliorer l'efficacité du programme. Dès qu'un vecteur est considéré comme ayant convergé, il n'est plus modifié lors des itérations qui suivent. Donc la précision de calcul restera proche du seuil d'acceptation.

Q13)
c.f. code

IV) TO DO: Numerical experiments

Q14)
Ce qui différencie la création des 4 types de matrices est la génération de leurs valeurs propres.
En effet, les matrices de premier type ont des valeurs propres telles que la ième valeur propre vaut i.
Les matrices de deuxième type quant à elles ont des valeurs propres tirées aléatoirement de manière uniformément répartie sur l'intervalle [0; 1/cond].
Les matrices de troisième type ont pour valeur propre la valeur cond élevée à une puissance qui est déterminée selon l'indice de la valeur propre.
Enfin, les matrices de type 4 ont des valeurs propres linéaires selon l'équation données dans le début du code.



Q15)
test de subspace_iter_v3 avec p = 5
Pour les matrices de type 1:
Pour une matrice de taille 200 x 200 :convergence non atteinte

Pour les matrices de type 2:
Pour une matrice de taille 200 x 200 : 0.03s
Pour une matrice de taille 500 x 500: 0,04s
Pour une matrice de taille 800 x 800: 0,3s

Pour les matrices de type 3:
Pour une matrice de taille 200 x 200 : 0.02s
Pour une matrice de taille 500 x 500: convergence non atteinte

Pour les matrices de type 4:
Pour une matrice de taille 200 x 200 : convergence non atteinte

En comparant les résultats obtenus à la Q1, cette fonction n'est pas encore aussi rapide que eig mais s'en rapproche. Globalement, cette verision est un peu moins de deux fois plus rapide que la version 2 et 4 fois plus rapide que la version 1
De plus, on observe une précision entre 5*10^-10 et 1*10^-8 pour les vecteurs propre, ce qui est conforme aux hypothèse de la Q12