RAPPORT FINAL
CALCUL SCIENTIFIQUE PARALLÈLE

Sujet: Application du calcul parallèle au calcul des valeurs propres d'une matrice carrée

VUONG Thi Anh Tuyet

Table des matières

Chapitre 1

Théorie sur le calcul numérique des valeurs propres

1.1 Méthode de puissances itérées

La méthode des puissances itérées est largement utilisée pour calculer une valeur propre d'une matrice carrée grâce à sa simplicité et efficacité. Soit une matrice carrée A et un vecteur quelconque b, A possède une valeur propre λ_1 qui est strictement plus grande que toutes les autres valeurs propres en valeur absolue. Cette méthode consiste à choisir un vecteur w quelconque et à succestivement multiplifer b par A : $b_{n+1} = Ab_n$, alors on a :

$$\lim_{n \to +\infty} \frac{w^t b_{n+1}}{w^t b_n} = \lambda_1$$

De plus, la suite des vecteurs normalisés $\frac{b_n}{\|b_n\|}$ converge vers le vecteur propre associé à λ_1. Autrement dit, la suite de vecteur définie par

$$b_k = \frac{A^k b_0}{\|A^k b_0\|}$$

converge vers le vecteur propre associé à la plus grande valeur propre de A en valeur absolue. Cette formule est pratique dans le cas où on veut contrôler l'incertitude de b par le nombre d'itérations. La seule difficulté ici est de calculer A^N avec A très large. Afin d'avoir un temps de calcul correct, la mise en place d'un algorithme de calcule en parallèle est nécessaire et sera détaillée dans la deuxième partie du rapport.

Plus l'écart entre λ_1 et les autres valeurs propres est grand, plus la vitesse de convergence de la suite est élevée. En revanche, un point faible majeur de cette méthode est qu'elle ne calcule que la

plus grande valeur propre en valeur absolue. Pour calculer les autres valeurs propres, il faut faire appel au théorème suivant :

Supposons que la matrice carrée A ait une valeur propre λ_p qui est plus proche de p que toutes les autres valeurs propres. Définissons la suite suivante avec un vecteur u_0 de départ quelconque :

$$b_{n+1} = (A - pI)^{-1} b_n$$

Soit w un vecteur quelconque, alors :

$$\lim_{x \to +\infty} \frac{w^t b_{n+1}}{w^t b_n} = \frac{1}{\lambda_p - p}$$

De plus, la suite des vecteurs normalisés $\frac{b_n}{\|b_n\|}$ converge vers le vecteur propre associé à λ_p

On peut constater que la méthode des puissances itérées est seulement puissance pour trouver la plus grande valeur propre en valeur absolue. En effet, afin d'utiliser le théorème cité ci-dessus, il faut connaitre en amont une valeur p proche de λ_p, en plus l'algorithme évoque l'inversion d'une matrice, ce qui est une opération coûteuse. Dans le cadre de ce projet, je ne calcule que la plus grande valeur propre avec la méthode de puissances itérées. Les deux méthodes présentées ci-après permettent de calculer toutes les valuers propres d'une matrice carrée.

1.2 Méthode d'itération de Jacobi

Cette méthode cherche à diagonaliser une matrice carrée A symétrique, il donne à la fois les valeurs propres et les vecteurs propres associées. Dans le cadre de ce projet je m'intéresse seulement à l'ensemble des valeurs propres. Le principe est de multiplier A à gauche et à droite par des matrices de rotation $R(i, j, \theta)$ afin d'annuler le coefficient à la position (i,j) et (j,i). Posons G la matrice de rotation cherchée et

$$B = G^t A G$$

B est symétrique et elle a les mêmes valeur propres que la matrice A. Afin que b_{ij} et b_{ji} soient nuls, on trouve que G devrait être de la forme :

$$\begin{pmatrix} 1 & ... & ... & ... & ... & ... \\ ... & c & ... & ... & s & ... \\ ... & ... & 1 & ... & ... & ... \\ ... & ... & ... & 1 & ... & ... \\ ... & -s & ... & ... & c & ... \\ ... & ... & ... & ... & ... & 1 \end{pmatrix}$$

avec

$$s = (\frac{1}{2} - \frac{\beta}{2\sqrt{1+\beta^2}})^{1/2}$$

$$c = (\frac{1}{2} + \frac{\beta}{2\sqrt{1+\beta^2}})^{1/2}$$

$$\beta = \frac{a_{ii} - a_{jj}}{2a_ij}$$

En revanche, on ne peut pas effectuer $n(n-1)/2$ opérations comme ci-dessus pour éliminer tous les coefficients hors de la diagonale. En effet, une fois qu'un couple de coefficient b_{ij} et b_{ji} est annulé à l'itération k, ils peuvent réapparaître non nuls aux itérations suivantes . Donc à la fin on ne peut jamais obtenir une matrice parfaitement diagonale. Pourtant, grâce au théorème suivant,on sait que la matrice A tend vers une matrice diagonale :

Soit $B = G^tAG$ une itération de Jacobi alors la somme des carrées de coefficients sur la diagonale croît d'une valeur $2a_{ij}^2$. Comme la somme des carrées de tous les éléménts de A et de B coincident, on a la somme des carrées de coefficients hors de la diagonale décroît d'une valeur $2a_{ij}^2$.

Autrement dit les coefficients hors de la diagonale de A sont de plus en plus proche de 0 apès chaque itération. De plus, la convergence est plus vite avec a_{ij} plus grand. L'iteration est :

$$A_0 = A$$

$$A_{n+1} = G_n^t A_n G_n$$

avec G_n la matrice de rotation qui annule le plus grand coefficient hors de la diagonale de A_n. A_n tend vers une matrice diagonale avec les valeurs propres sur la diagonale.

1.3 Méthode de décomposition QR

1.3.1 La décomposition QR

La décompostion QR d'une matrice carrée permet d'écrire $A = QR$ avec Q une matrice orthogonale et R une matrice triangulaire supérieure. Il existe des algorithmes différents pour calculer cette factorization, dans ce projet je choisis d'utiliser la méthode de réflexion de Householder.

La matrice de réflexion de Householder est de la forme $H = I - 2u^tu$ avec $\|u\| = 1$. Elle est orthogonale et symétrique. Soit un vecteur colonne x non nul, on peut trouver une matrice de réfexion

de Householder H pour annuler tous les coefficients de x sauf le premier. C'est à dire :

$$Hx = \|x\|e_1$$

En remplaçant l'expression de H dans la formule ci-dessus, on obtient :

$$u = \frac{x \pm \|x\|e_1}{\|x \pm \|x\|e_1\|}$$

Cette formule est valable tant que le vecteur $x \pm \|x\|e_1$ est non nul. Je choisit x + signe$(x_1)\|x\|e_1$ afin que le premier élément de ce dernier soit non nul.

L'idée est d'utiliser les réflexions de Householder pour ramemer la matrice A à une matrice triangulaire supérieur R par des multiplications à gauche. Si on note x_1 la première colonne de A et $H(x_1) = H_1$ sa matrice de Householder alors

$$A_1 = H_1 A = \begin{pmatrix} * & * & * & ... & * & * \\ 0 & * & * & ... & * & * \\ 0 & * & * & ... & * & * \\ ... & ... & ... & ... & ... & ... \\ 0 & * & * & ... & * & * \end{pmatrix}$$

Pour l'itération suivante on considère le vecteur x'_2 qui est la deuxième colonne de A privée du premier élément. On calcule sa matrice de Householder $H'(x'_2)$ de taille $(n-1)(n-1)$ et on pose :

$$H_2 = \begin{pmatrix} 1 & 0 & 0 & & ... & & 0 & 0 \\ 0 & & & & & & & \\ 0 & & & & H'(x'_2) & & & \\ ... & & & & & & & \\ 0 & & & & & & & \end{pmatrix}$$

Alors on a :

$$A_2 = H_2 A_1 = H_2 H_1 A = \begin{pmatrix} * & * & * & ... & * & * \\ 0 & * & * & ... & * & * \\ 0 & 0 & * & ... & * & * \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & * & ... & * & * \end{pmatrix}$$

On répète l'opération $n-1$ fois et à la fin on obtient $A_n = R$ triangulaire supérieur. En posant $Q = H_1 H_2 ... H_{n-1}$ on a donc :

$$A = QR$$

Les matrices de Householder sont orthogonales, donc la matrice Q est bien orthogonale aussi.

1.3.2 Calcul de valeurs propres

Une fois que l'algorithme pour la décomposition QR est mise en place, le calcul de valeurs propres est assez simple. On va construire une suite de matrices qui converge vers une matrice triangulaire supérieur avec les valeurs propres de la matrice de départ sur la diagonale.

Soit A la matrice à calculer les valuers propres. On part avec $A_0 = A$ et on effectue une décomposition QR de A_0 :

$$A_0 = Q_0 R_0$$

On remarque que $R_0 Q_0 = Q_0^t A_0 Q_0$ est une matrice conjuguée à A_0 et elle a les mêmes valeurs propres que A_0.

On pose en suite $A_1 = R_0 Q_0$ et on recommence la décomposition QR

Formule générale de l'itération k : $A_{k+1} = R_k Q_k$

A la fin on laisse $A_n = R_{n-1} Q_{n-1}$ et on on obtient les valeurs propres de A sur la diagonale de A_n

Chapitre 2

Calculer le produit matriciel en parallèle

2.1 Algorithme de base

La formule usuelle de calcul matriciel est :

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

Avec $C_{i,j}$ est le coefficient situé sur la ligne i et sur la colonne j de la matrice C. Il faut donc 3 boucles sur n pour calculer la totalité des coefficients de C :

```
for i from 1 to n do
  for j from 1 to n do
    for k from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

Remarquons que cet algorithme calcule en fait C = C + A*B. Afin d'obtenir C = A*B, il suffit d'initialiser les coefficients de C à 0.

2.2 Analyse de la dépendance entre les itérations

Dans la boucle k, considérons deux itérations k_1 et k_2 :

Il y a une dépendance de sortie car $OUT(k_1) = OUT(k_2)$: deux itérations écrivent sur la même variable $C[i][j]$

Il y a une dépendance de données car $OUT(k_1) \cap IN(k_2) = \{C[i][j]\}$

Donc on ne peux pas paralléliser la boucle k.

Itération $(i, j) : C[i][j] = C[i][j] + A[i][k] * B[k][j]$

Itération $(i \pm k_1, j \mp k_2) : C[i \pm k_1][j \mp k_2] = C[i \pm k_1][j \mp k_2] + A[i \pm k_1][k] * B[k][j \mp k_2]$

Entre la boucle i et j , il n'y a pas de dépendances de données ou de sorties entre les itérations (i,j) et $(i \pm k_1, j \mp k_2)$ (avec k_1, k_2 strictement positifs) donc on peut permuter les deux boucles. De même, on peut aussi permuter les boucles k et i ou k et j. La parallélisation est possible sur la boucle i ou j.

2.3 Algorithme par lignes ou par colonnes et ses désavantages

Si on parallélise la boucle i (resp.boucle j), il s'agit d'un algorithme par lignes (resp. par colonnes). Chaque processeur possède donc un certain nombre de lignes de deux matrices globales A et B, puis en échangeant ses coefficients avec les autres, il effectue le calcul pour obtenir un résultat local (les lignes correspondantes de la matrice du résultat C). Considérons d'abord un premier algorithme pour un processus p où la boucle k est à l'intérieur de la boucle j :

```
for i from i1 to i2 do
  for j from 1 to n do
    for k from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

A chaque itération k, le processus accès à la mémoire pour récupérer deux coefficients $A[i][k]$ et $B[k][j]$ donc en total 2n accès. À chaque itération j , en plus de 2n accès évoqués par la boucle k, le processus doit récupérer le coefficient $C[i][j]$ tout au début du calcul. En fin il lui reste une dernière écriture sur la mémoire pour le résultat final de $C[i][j]$. Donc, $2n + 2$ accès en mémoire est effectué à la fin de chaque boucle j.

$$C[i][j] = C[i][j] + A[i][k] * B[k][j]$$

$$1write - 1read - - - 2nread$$

Dans le deuxième cas où la boucle j est à l'intérieur de la boucle k :

```
for i from i1 to i2 do
  for k from 1 to n do
```

```
    for j from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

A chaque itération j, le processeur accès 2n fois à la mémoire pour lire la valeur de $C[i][j]$ et $B[k][j]$ mais aussi n fois pour écrire la nouvelle valeur de $C[i][j]$. A chaque itération k, le processeur doit aussi récupérer une fois la valeur de $A[i][k]$, donc $3n + 1$ accès en total.

$$C[i][j] = C[i][j] + A[i][k] * B[k][j]$$
$$nwrite - nread - 1read - nread$$

D'un premier coup d'oeil, le premier l'algorithme semble être le meilleur choix mais en réalité, il faut prendre en compte aussi le temps d'accès. Dans le premier algorithme, il y a les accès par ligne et par colonne (dans la boucle k) tant dis qu'il y a que les accès par colonne dans le deuxième algorithme. Si les données sont rangées par colonne (elles sont sur les cases contiguës dans la mémoire) le deuxième algorithme est plus avantageux car ses accès à la mémoire sont plus rapide que ceux du premier.

Le même raisonnement est valable pour un algorithme par colonne où la boucle j est à l'extérieur, on se retrouve dans les deux sous-cas qui ne sont pas optimaux.

2.4 Algorithme par blocs

Dans les deux cas précédents, seul le coefficient $C[i][j]$ ou $A[i][k]$ est conservé dans le registre du processeur, les autres coefficients sont relu/réécrit à chaque itération. Ce n'est pas tout à fait le cas en pratique. En effet, chaque processeur procède une mémoire cache près de lui afin de stocker temporairement les données dont il a besoin après les avoir récupérées de la mémoire centrale. Tant que les matrices sont de dimension assez petite, leurs coefficients sont enregistrés dans la mémoire cache et l'accès aux données dans ce cas est beaucoup plus rapide. Il est donc fortement plus avantageux que chaque processeur manipule une matrice locale de petite taille afin qu'il puisse stocker les coefficients dans sa mémoire cache : il s'agit d'un algorithme par blocs.

Dans ce type d'algorithme, les matrices A et B (de dimension n) sont divisées en $p = q^2$ matrices carrées. Supposons que q divise

n, la dimension de chaque matrice est n/q. (Si ce n'est pas le cas, quelques solutions sera proposées dans la partie 3).

$$\begin{pmatrix} A_{11} & ... & A_{1J} & ... & A_{1q} \\ ... & ... & ... & ... & ... \\ A_{I1} & ... & A_{IJ} & ... & A_{Iq} \\ ... & ... & ... & ... & ... \\ A_{q1} & ... & A_{qJ} & ... & A_{qq} \end{pmatrix}$$

$$\begin{pmatrix} B_{11} & ... & B_{1J} & ... & B_{1q} \\ ... & ... & ... & ... & ... \\ B_{I1} & ... & B_{IJ} & ... & B_{Iq} \\ ... & ... & ... & ... & ... \\ B_{q1} & ... & B_{qJ} & ... & B_{qq} \end{pmatrix}$$

L'algorithme par bloc est :

```
for I from 1 to q do
  for J from 1 to q do
    for K from 1 to q do
      C[I][J] = C[I][J] +  A[I][K]*B[K][J];
    end for
  end for
end for
```

où $C[I][J] = C[I][J] + A[I][K] * B[K][J]$ est un calcul matriciel usuel réalisé avec un algorithme par ligne ou par colonne mentionné auparavant. Ce calcul ne pose pas de problème en terme d'optimalité parce que les matrices $A[I][K]$ et $B[I][J]$ sont de taille assez petite pour être stockées dans la mémoire cache du processeur.

Les coefficients de la matrice extraite peut être déduits du coefficient de la matrice globale grâce à la formule suivante :

$$A_{IJ}[i][j] = A[(I-1) * n/q + i][(I-1) * n/q + i]$$

En calcul parallèle, chaque processeur est assigné à un couple d'indice (I,J)(la position d'un bloc dans les matrices globales A,B). Ce couple (I,J) peut être déterminé grâce à son rang *rank* :

$$rank = (J-1) * q + (I-1)$$

ou bien :

$$I = rank\%q + 1$$
$$J = rank/q + 1$$

Chaque processeur possède donc un bloc d'indice (I,J) de la matrice A et un bloc de même indice de la matrice B. L'algorithme pour chaque processeur est réduit à

```
for K from 1 to q do
  C[I][J] = C[I][J] +  A[I][K]*B[K][J];
end for
```

À la fin de cet algorithme, le processus retourne le bloc C_{IJ} de la matrice du résultat. Pendant ses calculs, il fait appel aux blocs A[I][K] et B[K][J] qu'il ne possède pas. Remarquons que les blocs A[I][K] sont sur la même ligne et les blocs B[K][J] sont sur la même colonne que sa position dans la matrice des blocs. Il suffit donc pour le processus (I,J) de récupérer tous les blocs de la matrice A qui sont sur la même ligne que lui et tous les blocs de la matrice B qui sont sur la même colonne que lui.

Chapitre 3

Implémentation avec MPI. Explication du code

Les explications sur le déroulement du programme se trouve dans le fichier code source sous forme commentaire et dans un fichier README dans le même dossier que les fichiers de code. Je vous présente ci-dessous quelques points généraux et des remarques supplémentaires sur le programme.

3.1 Structure générale du programme

Dans chaque fichier de chaque méthode, plusieurs étapes sont transformées en fonction pour des question de la claireté de la fonction main. Leurs protoypes sont déclarées au début du fichier et le corps de ces fonctions avec la description sont placés après la fonction main.

Le code concernant le calcul en parallèle se trouve principalement dans le programme Produit_2Matrices_Parallele mais l'initiation de MPI et ses varibles se trouve dans la fonction main

3.2 La fonction *Produit_2Matrices_Parallele*

Ce programme calcule le produit de 2 matrices carrées **m1** et **m2** en parallèle. Elle est utilisée pour les deux méthodes : Jacobi et décomposition QR.

Dans le cadre de ce projet, les blocs des matrices **m1** et **m2** ne sont pas disponibles sur les processeurs comme en pratique. Il est nécessaire d'avoir un processeur maître (dans mon programme c'est le processus de rang 0) qui divise les matrices globales **m1** et **m2** en blocs puis envoie les blocs aux autres processeurs. C'est aussi lui qui

récupère tous les résultats locaux et les combiner pour reconstruire un résultat global. Après avoir envoyé les blocs aux autres processeurs, il effectue lui même un calcul sur les blocs d'indice (1,1).

3.2.1 Commutateur par ligne ou par colonne. Broadcasting.

À la fin de la partie 2.4 nous avons vu que chaque processus doit récupérer les blocs de la même ligne ou colonne que lui. Ceci est possible grâce à la fonction *MPI_ Comm_ Split(commutateur_ initial , couleur , critère , commutateur_ final)* de MPI. Tous les processus du *commutateur_ initial* doit appeler cette fonction et il se retrouvera ensuite dans le *commutateur_ final* avec tous les processus ayant la même *couleur* que lui. De plus, les processus dans le *commutateur_ final* sont attribués leur nouveau rang basé sur la valeur croissante du *critère*. Donc pour créer un commutateur par ligne, il suffit de mettre I comme *couleur* et J comme *critère* :
MPI_ Comm_ Split(MPI_ COMM_ WORLD , I , J , commutateur_ ligne) ;

De même pour le commutateur par colonne :
MPI_ Comm_ Split(MPI_ COMM_ WORLD , J , I , commutateur_ colonne) ;

Un processus d'indice (I_0, J_0) a le rang J_0-1 dans le commutateur par ligne et le rang $I_0 - 1$ dans le commutateur par colonne.

Pour recevoir les blocs provenant des autres processus mais aussi envoyer ses matrices locales à tout le monde dans un commutateur, chaque processus utilise la même fonction *B_ cast(B_ temps, longueur, type, K, commutateur)*. Il envoie les données dans *B_ temps* si son rang dans le commutateur est égale à K, et dans les autres cas il reçoit les données puis les stocke dans *B_ temps*.

3.2.2 Application au calcul de la puissance N d'une matrice : la fonction *puissanceMatrice*

La fonction **puissanceMatrice** permet de calculer la puissance N d'une matrice en parallèle. Elle est utilisée pour la méthode de puissances itérées.

Après avoir implémenter l'algorithme parallèle pour calculer le produit de deux matrices, il est assez direct d'en déduire l'algorithme pour calculer la puissance N d'une matrice A. Le principe est qu'à chaque fois, on remplace l'une des deux matrices de départ par le résultat de l'itération précédente. Les itérations sur N s'effectue localement au niveau de chaque processus.

```
m1 = m2 = A
for m from 1 to N do
  for I from 1 to q do
    for J from 1 to q do
      for K from 1 to q do
         res[I][J] = res[I][J] +  m1[I][K]*m2[K][J];
      end for
    end for
  end for
  m1 = res
end for
```

Au début du programme, chaque processus possède 2 matrices $m1\_local$ et $m2\_local$ égales à A[I][J]. Après que le calcul en parallèle soit effectué, chaque processus obtient un res_local correspond à $A[I][J]^2$. La matrice $m1\_local$ est ensuite remplacée par ce résultat local et la procédure de calcul en parallèle recommence. Cette procédure est répété N fois et à chaque fois $m1\_local = A[I][J]^k$ est remplacé par $res\_local = A[I][J]^{k+1}$. On obtient à la fin $res\_local = A[I][J]^N$ Ces résultats locaux sont récupérés par le processus 0 pour reconstruit A^N

3.3 Allocation de la mémoire

Pour des raisons pratiques, deux fonctions permettant d'allouer un tableau et de libérer la mémoire allouée à un tableau sont écrites dans mon programme. À noter que cette fonction d'allocation est spéciale pour allouer un tableau sur une mémoire contiguë. Une première tentation d'allouer la mémoire avec un tableau des pointeurs a toujours donné "segmentation fault" donc a été remplacée par cette solution.

Chapitre 4

Perspectives d'amélioration du programme

4.1 Sur la dimension de la matrice à computer

Rappelons que le programme pour ce projet ne fonctionne que dans le cas le nombre de processus est $p = q^2$ avec q un entier et q divise n pour avoir un calcul correct de la dimension des blocs. Dans le cas q ne divise pas n, on peut considérer une des solutions suivantes :

- Parmi les processeurs, un groupe calculera les matrices non-carrées de taille plus petite que les autres. Cette solution est possible à implémentée mais n'est pas très pratique : elle détruit l'uniformité du programme et donc rendre difficile voire impossible la communication interne dans les communicateurs ligne ou colonne. De plus, cette solution n'améliore pas le temps de calcul global. En effet, si tous les processus ont la même vitesse de calcul alors ils mettent un même temps pour effectuer leur calcul local. Le fait que certains processus calculent les matrices plus petites ne rapporte rien parce que le processus 0 doit quand même attendre les processus normaux pour combiner le résultat.

- Compléter la matrice de départ par des lignes et des colonnes contenant que des zéros afin d'obtenir une matrice globale de dimension n divisible par q. Comme expliqué ci-dessus, le fait d'ajouter les lignes ne modifie pas le temps de calcul global parce que c'est toujours le temps d'un processus, à condition que tous les processus font la même chose. Le programme est la même pour tous les processus et plus facile à contrôler. Cette solution semble raisonnable.

4.2 Sur la gestion de la mémoire

Comme la matrice A est utilisée sous forme bloc, le mieux c'est d'allouer la mémoire pour lui sous forme bloc aussi. Cela est possible en utilisant les tableaux à 4 dimensions A[i][j][I][J].

4.3 Programmation modulaire

Les trois fichiers utilisent en commun beaucoup de fonctions, notamment la fonction Produit_2Matrices_Parallèle qui est très longue. J'ai essayé de les mettre dans un fichier afin d'inclure simplement le fichier header pour les 3 méthodes mais cela a donné les erreurs de compilation inexplicable. Ce problème n'est pas encore résolu, mais une fois que la programmation modulaire est mise en place, l'application des fonctions écrites dans ce projet sera beaucoup plus étendu parce qu'on peut en servir dans les autres projets.

Chapitre 5

Bibliographie

Frédéric Magoulès, François-Xavier Roux, *Calcul scientifique parallèle.*
Editeur : Dunod, Paris 2013

Moler, C.*Experiments with MATLAB, Chapter 7 : Google PageRank*,
MathWorks, Inc., 2011
(https ://www.mathworks.fr/moler/exm/chapters/pagerank.pdf)

Laszlo Erdos *Linear Algebra for Math2601 : Numerical Methods, Chapter 5 : Numerical computation of eigenvalues, Chapter 3 : QR factorization revisited* Version : August 12, 2000 (https ://www-old.math.gatech.edu/academic/courses/core/math2601/Web-notes/)