TP 2 - Calcul Numérique

Exercice 6 : Prise en main de Scilab

> *Commandes utilisées ici*

Exercice 7 : Matrice random et problème "jouet"

> *Commandes utilisées ici*

Après avoir résolu le système **Ax=b**, nous cherchons à calculer l'erreur avant et arrière avec la fonction `\`.

Considérons la fonction `error_calc` :

```
format('v',25);

function [front_error, backward_error, regle_generale]=error_calc(size)
  A = rand(size,size)
  xex = rand(size,1)
  b = A * xex
  xerr = A\b
  front_error = norm(xex - xerr)/norm(xex)
  backward_error = norm(b-A*xerr)/(norm(A)*norm(xerr))
  regle_generale = front_error <=cond(A)*backward_error
endfunction
```

Cette fonction basique renvoie l'erreur avant, arrière et une vérification de la règle générale.

Nous allons maintenant utiliser cette fonction pour effectuer des mesures sur l'erreur de calcul.

```
function [] = backslash_error_calc(matrix_size_range, runs)
  [x, range] = size(matrix_size_range);
  back = zeros(range,1)
  front = zeros(range,1)
  filename = "errors_backslash.dat";
  [f, mode_f] = mopen(filename, "w");
  for i=matrix_size_range
   printf("Running on %d ...\n", i);
   front_err_runs = zeros(runs,1)
   back_err_runs = zeros(runs,1)
   for j=1:runs
    [back_err_runs(j,1), front_err_runs(j,1), r] = error_calc(i)
   end
   front(i,1) = mean(front_err_runs);
```

```
    back(i,1) = mean(back_err_runs);
    mfprintf(f, "%d %.17f %.17f\n", i, front(i,1), back(i,1));
  end
  mclose(f);
  printf("Data written to %s\n", filename);
endfunction
```

Les résultats sont écrits dans le fichier `errors_backslash.dat`.

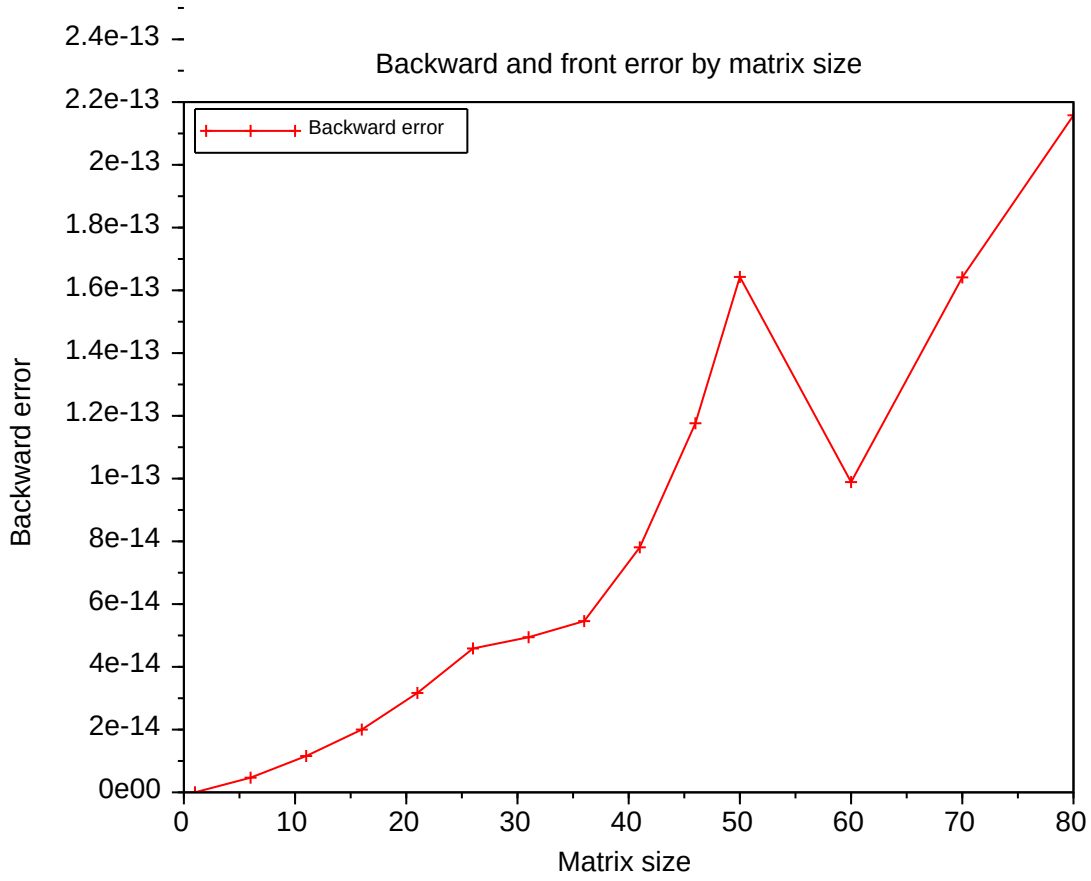
> La fonction `plot_backslash_errors()` crée un graphique à partir de ce fichier. Les erreurs avant et arrières sont affichées séparément, afin de voir la forme du graphe formé.

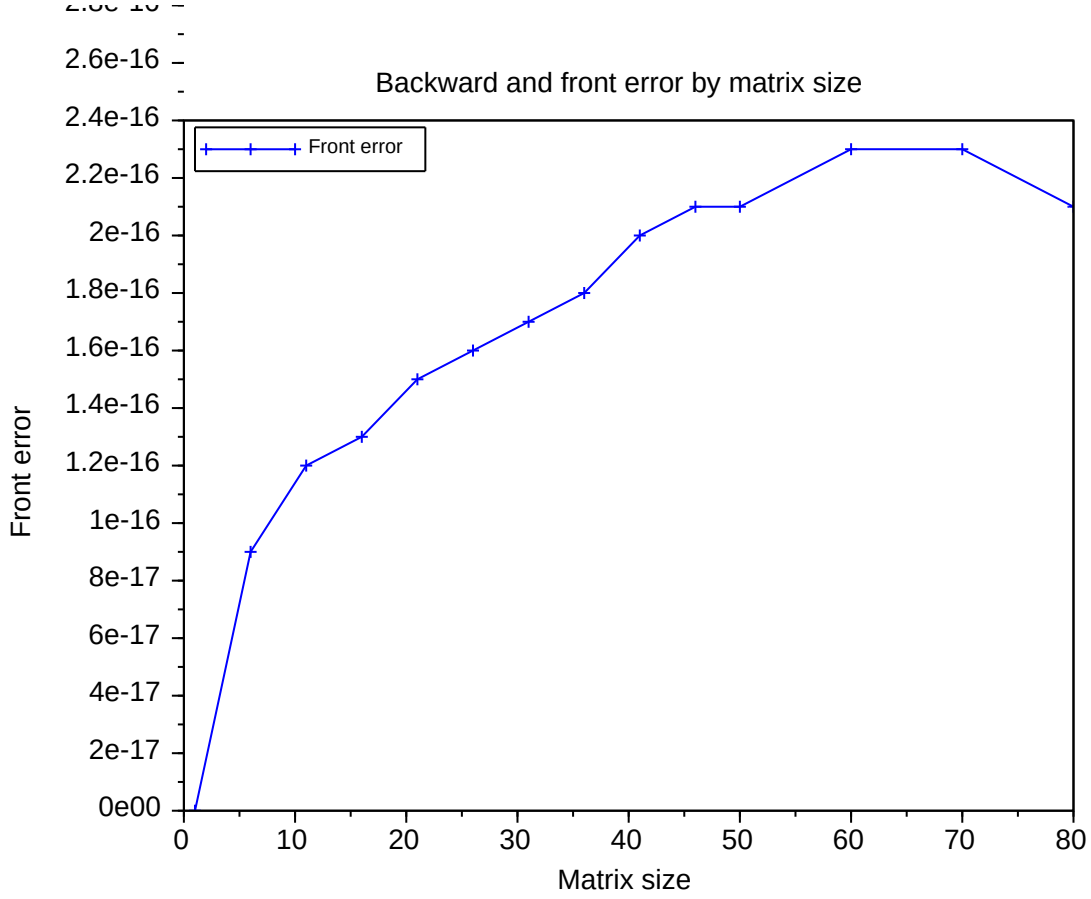
```
function plot_backslash_errors(filename)
clf();
data = csvRead('errors_backslash.dat', ' ');
xrange = data(:,1)
plot(xrange, data(:, 2), '+-b');
plot(xrange, data(:, 3), '+-r');
xlabel('Matrix size');
ylabel('Backward and front errors');
title('Backward and front error by matrix size')
h1 = legend(["Front error"; "Backward error"], pos=2);
a = gca();
a.font_size = 3;
a.x_label.font_size = 3;
a.y_label.font_size = 3;
a.title.font_size = 3;
xs2svg(gcf(), 'backslash_errors.svg');
endfunction
```

Enfin, on peut réaliser la fonction suivante pour simplifier l'utilisation :

```
function [] = run_backslash_errors(matrix_size_range, runs)
backslash_error_calc(matrix_size_range, runs);
plot_backslash_errors();
endfunction
```

> On lance un test sur des matrices de taille 1 à 100 avec 10000 itérations via la commande `run_backslash_errors([1:5:50,50:10:80],10000);`
>
> Voici les résultats d'erreurs avant et arrières :



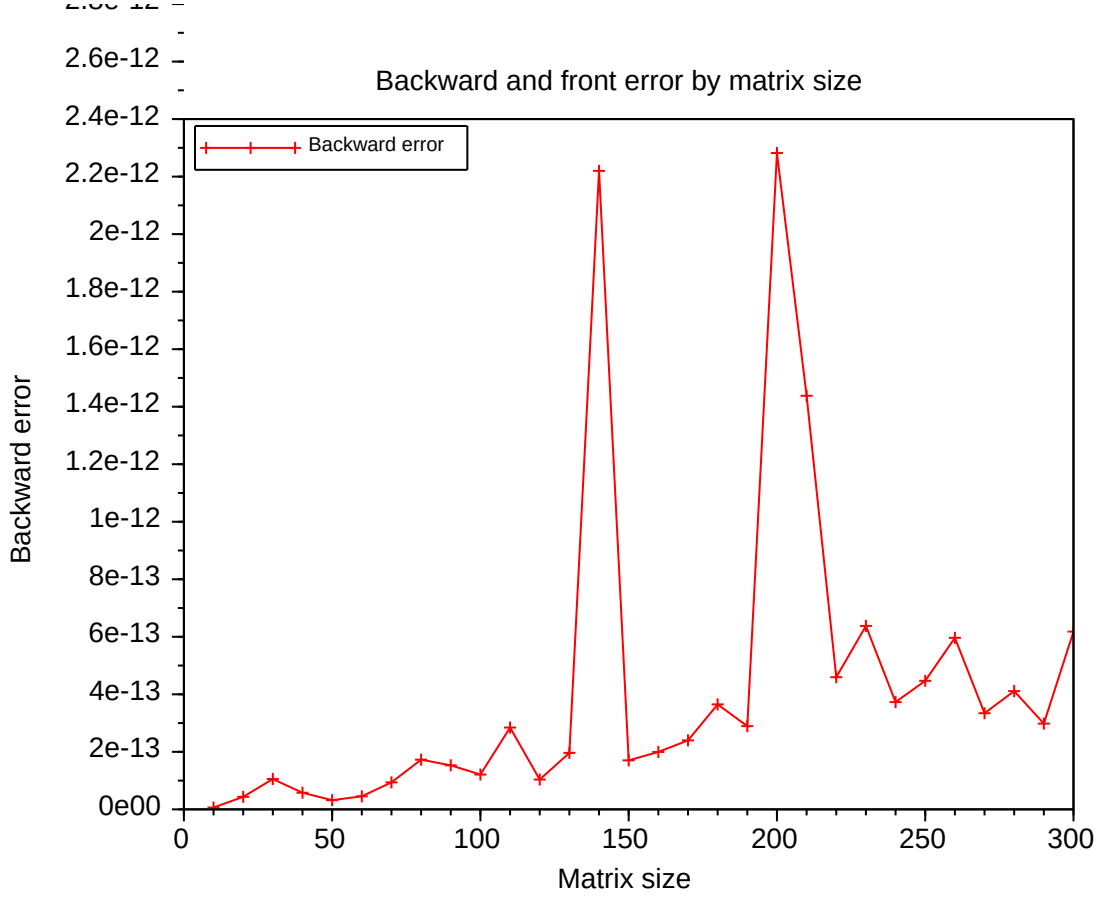


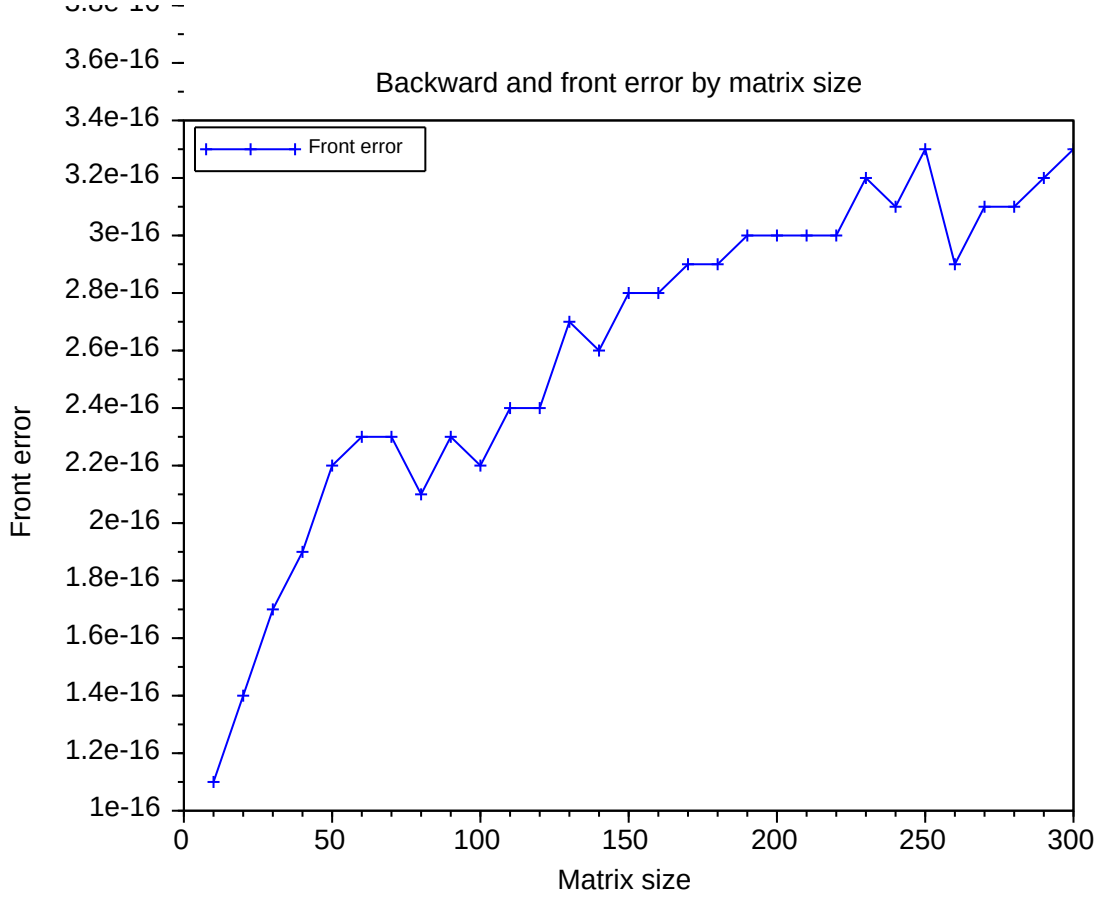
> On observe une augmentation en parallèle de l'erreur avant et arrière en fonction de la taille de la matrice.

> Pour des matrices de large taille, on a toujours de très loin une erreur arrière supérieure à l'erreur avant, de par un facteur 1000.

> Par ailleurs, sur certains tests de calcul d'erreur avant et arrière sur des matrices de petites tailles, il est possible d'obtenir une erreur avant nulle.
>
> Cela est causé par la machine, lors d'une erreur d'imprécision machine, si celle-ci est faible, elle peut s'auto compenser, et donc indiquer une erreur nulle.
>
> Plus la matrice a une taille faible, plus cet évènement est probable.

Après une exécution un peu moins précise *(10 à 300, avec 100 itérations commande :* `run_backslash_errors(10:10:300, 100)`*)*, mais sur un échantillon de tailles de matrice plus large, on observe les résultats suivants :





> Malgré un résultat quelque peut aléatoire sur les itérations 140 et 200 pour lesquelles l'erreur arrière est bien plus importante que je ne saurais expliquer,

> On peut observer globalement une augmentation linéaire sur l'erreur arrière, et une augmentation logarithmique.

> 6. Tests sur des matrices de tailles 100 1000 et 10000

```
// Taille de matrice : 100
--> [front_error, backward_error, regle_generale] = error_calc(100)
front_error  =
   0.000000000000498547741

backward_error  =
   0.000000000000002194836

regle_generale  =
  T

// Taille de matrice : 1000
--> [front_error, backward_error, regle_generale] = error_calc(1000)

front_error  =
   0.000000000002216511794

backward_error  =
   0.000000000000006281187

regle_generale  =
  T

// Taille de matrice : 10000
--> [front_error, backward_error, regle_generale] = error_calc(10000)

front_error  =
   0.000000000122077212336

backward_error  =
   0.000000000000035563504

regle_generale  =
  T
```

> la règle générale `front_error <=cond(A)*backward_error` est toujours vérifiée sur de large tailles de matrices, et sa vérasité augmente aussi parallèlement à la taille de la matrice.
>
> On peut observer qu'avec l'augementation de la taille de matrice, le nombre d'opérations augmente également, et par conséquent l'erreur avant et arrière.
>
> Par ailleurs, avec analyse de `cond(A)*backward_error - front_error`, on peut aussi observer une augmentation de la véracité de la règle générale `front_error <=cond(A)*backward_error`

Ex 8

> *Commandes utilisées ici*

Implémentation des algorithmes 7, 8 et 9 dans les fichiers matmat

> Exemple de vérification des fonctions..

```
A = [1,2,3;4,5,6]
B = [7,8;9,10;11,12]
matmab3b(A,B)
matmab2b(A,B)
matmab1b(A,B)

matmab3b(A,B)
ans  =
   58.    64.
   139.   154.

matmab2b(A,B)
ans  =
   58.    64.
   139.   154.

--> matmab1b(A,B)
ans  =
   58.    64.
   139.   154.
```

> Sur les premiers tests, en utilisation simple de la fonction apellée benchmark on observe une nette augmentation des performances pour les versions 2 et 1 boucles. Cette fonction va servir de base à une mise en profondeur des tests de performances

```
// Fonction mesure de performance sur un produit matrice matrice m,n,p
function [disp, avg, med, mn, v2, v3] = benchmark(runs, m,n,p)
t = zeros(3,runs);
for i=1:runs
   A = rand(m,p);
   B = rand(p,n);
   tic(); matmat1b(A,B);
   t(1,i) = toc();

   tic(); matmat2b(A,B);
   t(2,i) = toc();

   tic(); matmat3b(A,B);
   t(3,i) = toc();
end
```

```
disp = zeros(3,1);
avg = zeros(3,1);
med = zeros(3,1);
mn = zeros(3,1);

for i = 1:3
   disp(i,1) = max(t(i,:)) - min(t(i,:));
   avg(i,1) = (min(t(i,:)) + max(t(i,:))) / 2;
   med(i,1) = median(t(i,:));
   mn(i,1) = mean(t(i,:));
end

v2 = mn(3,1)/mn(2,1);
v3 = mn(3,1)/mn(1,1);
endfunction
```

> Fonction `data_matmat_benchmark` pour de multiples mesures de performances en fonction de la taille de matrice.

```
function [disp, avg, med, mn, v2, v3] =
data_matmat_benchmark(matrix_size_range, runs)
[x, range] = size(matrix_size_range)
disp = zeros(3, range);
avg = zeros(3, range);
med = zeros(3, range);
mn = zeros(3, range);
v2 = zeros(1, range);
v3 = zeros(1, range);
step = 1;
for i=matrix_size_range
   printf("Running %d*%d*%d ...\n", i,i,i);
   [disp(:,step), avg(:,step), med(:,step), mn(:,step), v2(1,step),
v3(1,step)] = benchmark(runs, i,i,i);
   step = step + 1;
end
endfunction
```

> Fonction `plot_matmat_benchmark` pour plot les résultats du test de performances

```
function plot_matmat_benchmark(matrix_size_range, v2, v3)
clf();
plot(matrix_size_range, 1, '+-b');
plot(matrix_size_range, v2, '+-g');
plot(matrix_size_range, v3, '+-r');
//errbar(matrix_size_range, avg, disp/2, disp/2);
//plot(matrix_size_range, disp, 'r.+');
xlabel('Size of matrix');
ylabel('Execution time speedup by version');
title('Execution time on matrix, matrix multiplication operation by version
```

```
and by matrix size')
h1 = legend(["Reference 3 loops";"SpeedUp 2 loops"; "SpeedUp 1 loops"],
pos=2);
a = gca();
a.font_size = 3;
a.x_label.font_size = 3;
a.y_label.font_size = 3;
a.title.font_size = 3;
endfunction
```

> Fonction principale `run_matmat_benchmark` pour lancer la mesure de performances

```
function [] = run_matmat_benchmark(matrix_size_range, runs)
[disp, avg, med, mn, v2, v3] = data_matmat_benchmark(matrix_size_range,
runs)
plot_matmat_benchmark(matrix_size_range, v2, v3)
endfunction
```

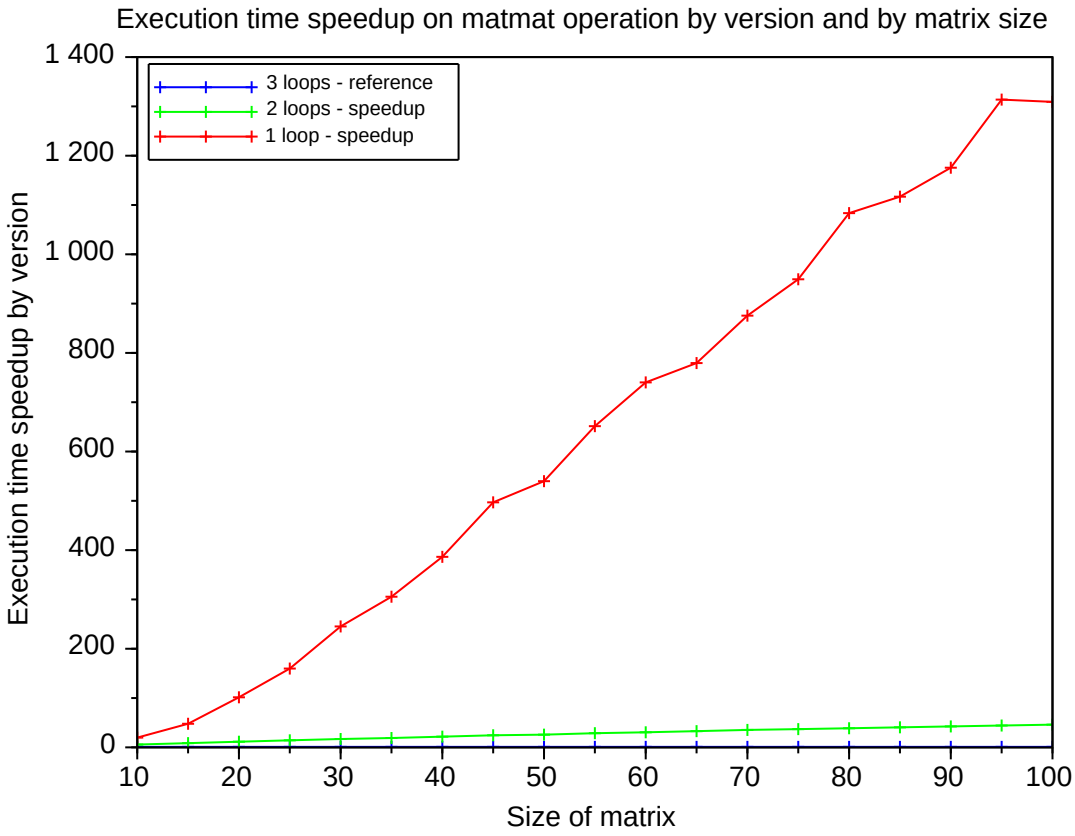
> **Maintenant, tests benchmark complets**

```
exec('ex8_produit_matrice_matrice.sci', -1);
// run_matmat_benchmark(10:5:100, 20)
// run_matmat_benchmark([10:10:100,110:20:200],10)
```

> Afin de rendre la mesure la plus stable possible, la fréquence du processeur ainsi que les cores de scilab été fixés et assignés manuellement.

```
sudo cpupower -c 3,4 frequency-set -g userspace && sudo cpupower -c 3,4
frequency-set -f 3.30GHz
sudo taskset -c 3,4 scilab -f
```

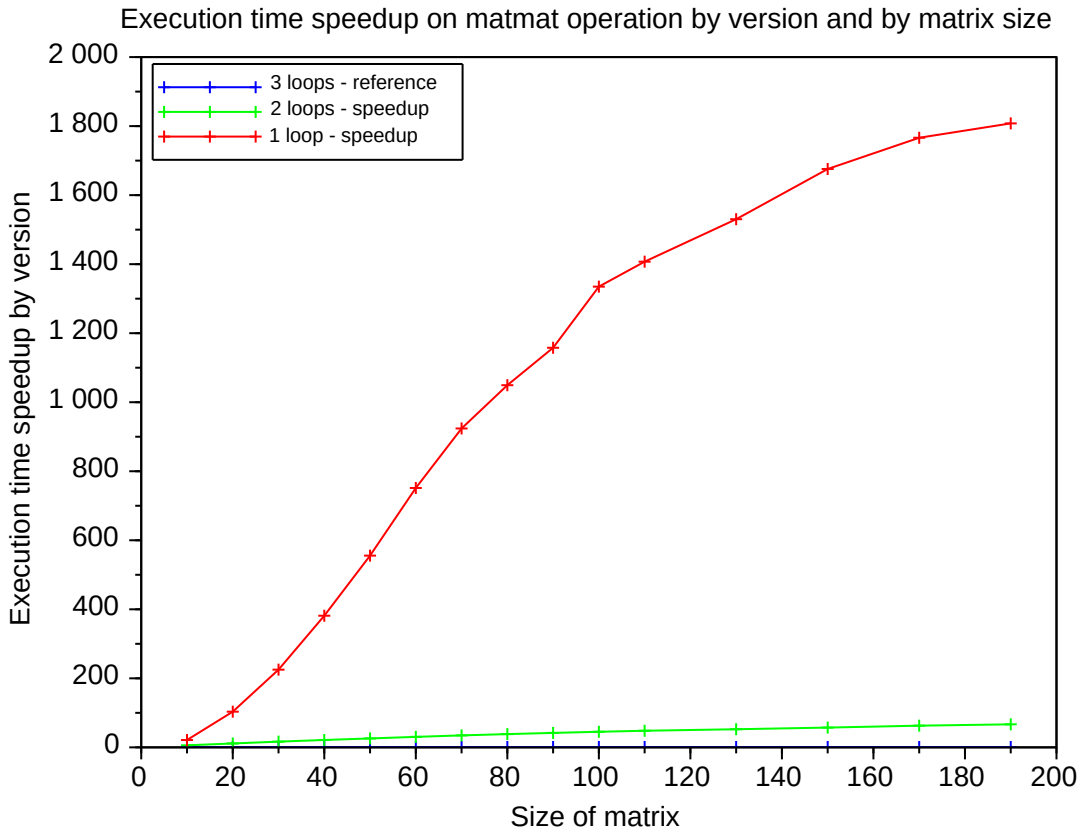
> En utilisant la commande suivante `run_matmat_benchmark(10:5:100, 20)`, on observe les résultats suivants :



> Dans le graphique, le temps d'exécution moyen de la fonction à 3 boucles sur `20` itérations sert de référence pour calculer l'accélération lors de l'utilisation des fonctions à 2 ou 1 boucle(s).

> On observe jusqu'à 1300 fois plus rapide pour la version 1 boucle que celle à 3 boucles pour des matrices de tailles 100.
>
> Plus la taille des matrices sur lesquelles nos fonctions sont exécutés est grande, plus l'augmentation de vitesse de ces fonctions est important.
>
> L'accélération causée par l'utilisation des fonctions à 2 ou 1 boucle(s) semble linéaire, avec un facteur bien plus important sur la fonction à une seule boucle.

> Cependant, avec une exécution plus longue sur une plage de taille de matrice plus grande *(10 à 200 avec 10 itérations* `run_matmat_benchmark([10:10:100,110:20:200],10)`*)* :



> On observe alors plutôt une accélération logarithmique pour la fonction à 1 boucle, qui semblerai se stabiliser vers une accélération de 2000x plus rapide que la version à 3 boucles.

TP3

Exercice 2 - Système triangulaire

Commandes utilisées ici

> Algorithme de remontée `usolve.sci`

```
function [x] = usolve(U,b)

[n,n]=size(U);
x = zeros(n,1);
x(n) = b(n)/U(n,n);
for i=n-1:-1:1
```

```
    x(i)=(b(i)-U(i,(i+1):n) *x((i+1):n)) /U(i,i);
end

endfunction
```

Algorithme de descente `lsolve.sci`

```
function [x] = lsolve(L,b)

[n,n] = size(L);
x = zeros(n,1);
x(1) = b(1)/L(1,1);
for i=2:n
    x(i)=(b(i) -L(i,1:(i-1)) *x(1:(i-1)))/L(i,i);
end

endfunction
```

```
// Initialisation d'une matrice carrée A, et d'un vecteur b, de dimensions
n = 5
A = rand(5,5)
b = rand(5,1)

// Génération de la matrice triangulaire supérieure U et la matrice
triangulaire inférieure L
L = tril(A)
U = triu(A)

// Vérification des fonctions usolve et lsolve:
usolve(U,b) == U\b
lsolve(L,b) == L\b


// Calculer l'erreur avant et arrière

usolve_x = usolve(U,b)
lsolve_x = lsolve(L,b)

u_xerr = U\b
l_xerr = L\b

usolve_x == xuerr
lsolve_x == xlerr
```

On vérifie maintenant les fonctions `usolve` et `lsolve`:

```
--> usolve(U,b) == U\b
ans  =
  F
  T
  T
  T
  T

--> lsolve(L,b) == L\b
ans  =
  T
  F
  F
  F
  F
```

```
--> lsolve(L,b)
ans  =
   0.2567970189540171865872
   0.756983024441462 8150906
  -0.041481661515362 887316
   0.489392594114332 5575311
  -2.11244258834476 37275481

--> L\b
ans  =
   0.2567970189540171865872
   0.756983024441462 7040683
  -0.041481661515362 776294
   0.489392594114332 446509
  -2.11244258834476 28393697
```

> Les résultats semblent très proches, mais ne sont pas toujours exacts, c'est pour cela qu'aurais du être présent ici un mesure des erreurs avant et arrière, avec un joli graphique.
>
> Ici, les résultats sont identiques jusqu'à la 10ième décimale en moyenne, dans notre cas, la 12ième décimale diffère

Exercice 3 - Gauss

Commandes utilisées ici

> Implémentation de l'algorithme de Gauss sans pivotage puis résolution par méthode de descente
> `gausskij3b.sci`

```
function [x] = gausskij3b(A,b)

[n, n] = size(A);

mik = 0

for k = 1 : n-1
    for i = k + 1 : n
        mik = A(i,k)/A(k,k);
        b(i)= b(i) - mik*b(k);
        for j = k + 1 : n
            A(i,j)= A(i,j) - mik*A(k,j);
        end
    end
end

x= usolve(A,b)

endfunction;
```

Initilisation de la matrice A et vecteur b, de taille n = 3

```
--> A = rand(3,3)
A  =
   0.459408376365900039672 9   0.855000307317823171615 6
0.524448265321552753448 5
   0.268569542560726404190 1   0.532012196257710456848 1
0.182661246042698621749 9
   0.664757437072694301605 2   0.548700679559260606765 7
0.527910053730010986328 1

--> b = rand(3,1)
b  =
   0.793151611927896738052 4
   0.576225045137107372283 9
   0.055843295063823461532 6
```

Calcul de la valeur réelle et approchée

```
--> xerr = gausskij3b(A,b)
xerr  =
  -0.988832443175836850457 5
   1.739054846033099011037 8
  -0.456599677203230580602 2

--> xex = A\b
xex  =
  -0.988832443175838182725 1
```

```
    1.739054846033098788 9932
   -0.456599677203229026 2899
```

> Maintenant on calcule l'erreur avant et arrière.

```
--> front_error = norm(xex - xerr)/norm(xex)
front_error  =
   0.000000000000001003 5020

--> backward_error = norm(b-A*xerr)/(norm(A)*norm(xerr))
backward_error  =
   0.000000000000000075 9706


--> front_error <= cond(A)*backward_error
rg  =
  T
```

> Test sur la fonction solve_gauss qui permet de mesurer les erreurs arrières et avants.
>
> Voici un exemple avec une taille de matrice de 200.

```
--> [front_error, backward_error, rg] = solve_gauss(200)
front_error  =
   0.000000000002363402 3447

backward_error  =
   0.000000000000014750 0831

rg  =
  T
```

> Avec une meilleure gestion du temps, auraient été ici présents des graphiques, concernant les erreurs arrières et avant.

Exercice 4

```
// Vérification des fonciton `mylu3b` et `mylu1b`
[L,U] = lu(A);
[L2,U2] = mylu3b(A);
[L3,U3] = mylu1b(A);

--> [L, U] == [L2, U2]
ans  =
  F F T T F F F F
  F F F T T F F F
  F F F F T T F F
```

```
  F F F F T T T F

--> [L, U] == [L3, U3]
ans  =
  F F T T F F F F
  F F F T T F F F
  F F F F T T F F
  F F F F T T T F


// Les fonctions `mylu3b` et `mylu1b`
--> [L2, U2] == [L3, U3]
ans  =
  T T T T T T T T
  T T T T T T T T
  T T T T T T T T
  T T T T T T T T
```

> On observe que les fonctions `mylu3b` et `mylu1b` ont un résultat différent de celui obtenu par `lu(A)`. Cela est dû au fait que la fonction `lu` ne renvoie pas toujours le résultat esconté.
>
> Les fonctions `mylu3b` et `mylu1b` identiquement exactement le même résultat.
>
> Avec une meilleure gestion du temps, auraient été présents ici des calculs d'erreur avant et arrière en comparaison avec une matrice [L, U] correctemetn renvoyée par `lu(A)`.

Calcul erreurs

CALCUL ERREUR AVANT ERREUR ARRIERE Erreur avant : différence entre la valeur mathématique et la valeur calculée erreur arrière : la valeur qui permet d'obtenir le résultat calculé (f(x)) : comparer ce x avec le x réel. A x^ = b : on obtient pas de b^ (sur de petites matrices, car le x^ est tellement proche de x que pour l'erreur machine ne change pas le résultat)

Annexes

https://github.com/fm16191/CHPS1-CN-TD2-3