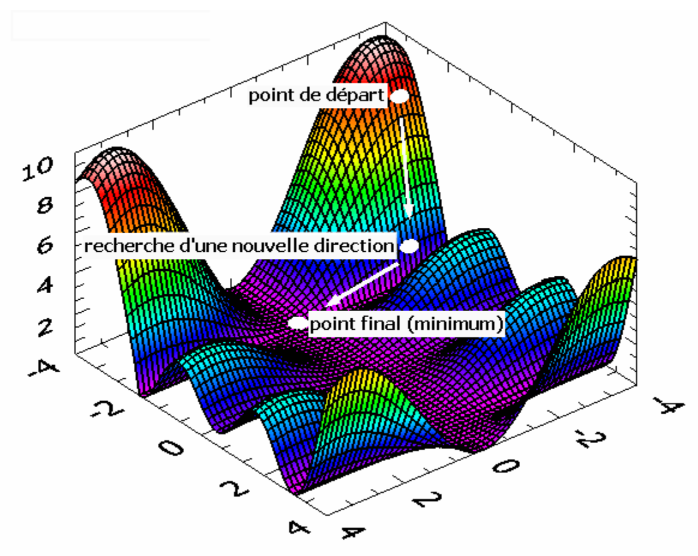
Institut national des sciences appliquées de Rouen

Projet MMSN GM3 - Vague 3 - Sujet 4

Etude des erreurs sur la méthode du Gradient Conjugué



Auteurs :
Thibaut André-Gallis
thibaut.andregallis@insa-rouen.fr
Kévin Gatel
kevin.gatel@insa- rouen.fr

Enseignants :
Bernard Gleyse
bernard.gleyse@insa-rouen.fr

6 Juin 2021

Table des matières

Introduction

Pour commencer, nous avons longtemps pensé que les calculatrices et les ordinateurs étaient les références absolues en termes de calcul mathématique. Qu'ils ne se trompaient jamais pourvu qu'on leur donne le bon calcul. Cependant nous avons par la suite découvert que les réels n'existaient pas en machine et qu'on utilisait les flottants pour les représenter. Nous ne détaillerons pas la construction des flottants, ce n'est pas notre sujet mais il est important de connaître leur existence. De cela nous avons compris qu'il était impossible de représenter tous les réels par les flottants. Ce qui entrainera forcément des erreurs inévitables lors des calculs.

C'est là tout le sujet de notre projet, les erreurs. En effet nous allons nous intéresser aux erreurs de calculs survenus lors de la résolution d'un système linéaire sur machine. Plus particulièrement avec la méthode du gradient conjugué que nous avons déjà étudié auparavant lors d'un précédent projet. Nous allons donc à l'aide de la valeur binaire des nombres et de la connaissance de la construction des flottants pouvoir étudier et quantifier les erreurs survenues lors de la résolution du système linéaire.

Pour obtenir une étude plus large tous les calculs ont été réalisés sur deux machines différentes dont les caractéristiques ont été détaillés dans le fichier "README". Nous pourrons par conséquent les comparer pour remarquer des éventuelles différences d'une machine à l'autre.

1. Présentation du problème

L'objectif est donc d'étudier les erreurs que fait la machine en utilisant l'arithmétique flottante plutôt que l'ensemble théorique des réels.

Ces erreurs seront étudiées sur la solution du problème linéaire $Ax = b$ avec la méthode du gradient conjugué. En choisissant la matrice A de dimension 4 définie comme ci-dessous :

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix}$$

Figure 1.1 – Matrice de Hilbert de dimension 4

Le nombre d'étape pour trouver la solution sera en théorie inférieur ou égale à 4 (assuré par la méthode du gradient conjugué).

En notant $K_2(A)$ le conditionnement 2 de A tel que :[1]

$$K_2(a) = 1.5514 * 10^4$$

On a l'inégalité du conditionnement pour majorer l'erreur :

$$\frac{||\Delta x||_2}{||x||_2} \leq K_2(A) \frac{||\Delta b||_2}{||b||_2}$$

Le test d'arrêt est de la forme

$$tol^2 * (b, b) > (r, r)$$

avec $(\bullet, \bullet)$ le produit scalaire usuel et $tol = 10^{-10}$.

Enfin, le vecteur b est choisi comme ci-dessous :

$$b_i = \sum_{k=1}^{4} A_{ik}$$

de manière à avoir

$$x^T = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

1. conditionnement obtenu sur Matlab

2. Vecteur résidu r

2.1 Préambule

Premier élément d'intérêt de notre projet, le vecteur résidu. Intéressons nous aux erreurs faites par la machine sur ce vecteur. Nous ne reviendrons pas sur le rôle du vecteur résidu dans la méthode du gradient conjugué mais il est essentiel au bon fonctionnement de celle-ci.
Maintenant nous allons expliquer la démarche utilisée pour étudier cette erreur.

```
(gdb) print r(1)
$1 = -2.083333333333333
(gdb) x/tg &r(1)
0x555555561b20: 1100000000000000101010101010101010101010101010101010101010101010
(gdb) print r(2)
$2 = -1.2833333333333332
(gdb) x/tg &r(2)
0x555555561b28: 1011111111110100100010001000100010001000100010001000100010001000
(gdb) print r(3)
$3 = -0.94999999999999984
(gdb) x/tg &r(3)
0x555555561b30: 1011111111101110011001100110011001100110011001100110011001100101
(gdb) print r(4)
$4 = -0.75952380952380949
(gdb) x/tg &r(4)
0x555555561b38: 1011111111101000010011100000010011100000010011100000010011100000
(gdb) print r2
$5 = 7.4665986394557811
(gdb) x/tg &r2
0x7fffffffde48: 0100000000011101110111011100110000000100010100011010010000101111
(gdb)
```

FIGURE 2.1 – Capture d'écran avec le logiciel gdb

Comme on peut le voir ci-dessus, nous avons utilisé le logiciel gdb avec notre programme de gradient conjugué. On utilise le logiciel gdb pour obtenir certaines valeurs pendant le déroulement du programme et pas uniquement à la fin. En ce qui nous concerne nous récupérons les valeurs des composantes du vecteur résidu ainsi que leurs valeurs binaires qui seront importantes pour la suite. la composante r2 est la norme 2 au carré du vecteur résidu et c'est précisément ce que nous allons comparer avec notre calcul pour trouver l'erreur.

```
R(1):
Nombre en écriture binaire : 1100000000000000101010101010101010101010101010101010101010101010

bit de signe : 1
bits d'exposants : 10000000000
bits de mantisses : 0000101010101010101010101010101010101010101010101010

R(1) est négatif car bit de signe=1 ; Normalisé car bits d'exposants différents de 00000000000.

exposant biaisé = 2^10 = 1024
exposant réel = 1024-d = 1024-1023 = 1
mantisse = 2^-5 + 2^-7 + 2^-9 + 2^-11 + 2^-13 + 2^-15 + 2^-17 + 2^-19 + 2^-21 + 2^-23 + 2^-25 +
2^-27 + 2^-29 + 2^-31 + 2^-33 + 2^-35 + 2^-37 + 2^-39 + 2^-41 + 2^-43 + 2^-45 + 2^-47 + 2^-49 +
2^-51 = 93824992236885/2251799813685248 ≃ 0.0416666666666665186

Valeur décimale exacte : ((-1)^1)*(1+0.0416666666666665186)*(2^(1)) = -2.0833333333333330372
```

Figure 2.2 – Conversion binaire réel

Deuxième étape, on retrouve la valeur réel en base 10 de notre nombre à partir de son écriture binaire en flottant.
On décompose le nombre par bits de signes, exposants et mantisses. On recalcule chacun des termes puis la valeur réelle finale grâce au logiciel en ligne wolframalpha.
Maintenant que nous avons la valeur réelle de chaque composante nous allons pouvoir en déduire une valeur calculée par nous même de r2 et la comparé avec celle donnée par l'ordinateur.
C'est ce que nous allons maintenant voir à chaque étape de l'itération du gradient conjugué et observé l'erreur absolue et relative entre la solution de l'ordinateur et la notre.

2.2 Etape 0

```
R(1) = -2.0833333333333330372
R(1)^2 = 4.3402777777777765439

R(2) = -1.2833333333333332149
R(2)^2 = 1.6469444444444441405

R(3) = -0.94999999999999984455
R(3)^2 = 0.90249999999999970465

R(4) = -0.7595238095238094900
R(4)^2 = 0.5768764172335600393

R2= Z = 7.4665986394557811

z = R(1)^2+R(2)^2+R(3)^2+R(4)^2 = 7.4665986394557804284
on ramène z au même nombre de chiffres significatifs que Z :
z = 7.4665986394557804

On calcule l'erreur absolue :
|Z-z|= 7.0×10^-16

On calcule l'erreur relative :
|Z-z|/|z|= 9.3750854144079325033×10^-17
```

FIGURE 2.3 – Fichier d'erreur à l'étape 0

On peut voir que l'erreur absolue est de 10^{-16} et l'erreur relative de 10^{-17}. On est autour de la précision machine pour un double précision et l'erreur relative reste extrêmement faible. On peut être plutôt satisfait du résultat.

2.3 Etape 1

```
R(1) = 0.0755642187648017937 82
R(1)^2 = 0.00570995115753482352 92

R(2) = -0.0414933624002828960 19
R(2)^2 = 0.00172169912328121037408

R(3) = -0.0602219126779675040 24
R(3)^2 = 0.00362667876659274319203

R(4) = -0.0618345640342945479 52
R(4)^2 = 0.00382351330931127284155

R2= Z = 0.0148818423567200503 60

z = R(1)^2+R(2)^2+R(3)^2+R(4)^2 = 0.0148818423567200499 3686
on ramène z au même nombre de chiffres significatifs que Z :
z = 0.01488184235672004993 7

On calcule l'erreur absolue :
|Z-z|= 4.23 × 10^-19

On calcule l'erreur relative :
|Z-z|/|z|= 2.8423900069670471420 × 10^-17
```

Figure 2.4 – Fichier d'erreur à l'étape 1

Cette fois l'erreur absolue est de 10^{-19} et la relative de 10^{-17} c'est même moins que l'étape d'avant.

2.4 Etape 2

```
R(1) = -0.00020922549391372280425
R(1)^2 = 4.3775307303441258204 × 10^-8

R(2) = 0.00084327357243061534797
R(2)^2 = 7.1111031795989226947 × 10^-7

R(3) = -0.00010617738802128767750
R(3)^2 = 1.1273637727023083988 × 10^-8

R(4) = -0.00071814222741654887283
R(4)^2 = 5.1572825879880219955 × 10^-7

R2= Z = 1.2818875217891588983 × 10^-6

z = R(1)^2+R(2)^2+R(3)^2+R(4)^2 = 1.281887521789158811212 × 10^-6
on ramène z au même nombre de chiffres significatifs que Z :
z = 1.2818875217891588112 × 10^-6

On calcule l'erreur absolue :
|Z-z|= 8.71 × 10^-23

On calcule l'erreur relative :
|Z-z|/|z|= 6.7946679033455759453 × 10^-17
```

FIGURE 2.5 – Fichier d'erreur à l'étape 2

Une fois de plus l'erreur absolue diminue elle est de $10^{-}-23$ cependant quand on regarde l'erreur relative en 10^{-17} on se rend compte qu'elle na pas diminué.

2.5 Etape 3

```
R(1) = 6.3762514465349204654 × 10^-8
R(1)^2 = 4.0656582509438665698 × 10^-15

R(2) = -7.1261809844009541687 × 10^-7
R(2)^2 = 5.0782455422437752201 × 10^-13

R(3) = 1.7086666279479700587 × 10^-6
R(3)^2 = 2.9195416454630867353 × 10^-12

R(4) = -1.1079901479170818981 × 10^-6
R(4)^2 = 1.2276421678813170240 × 10^-12

R2= Z = 4.6590740258197256465 × 10^-12

z = R(1)^2+R(2)^2+R(3)^2+R(4)^2 = 4.6590740258197251478798 × 10^-12
on ramène z au même nombre de chiffres significatifs que Z :
z = 4.6590740258197251479 × 10^-12

On calcule l'erreur absolue :
|Z-z|= 4.986 × 10^-28

On calcule l'erreur relative :
|Z-z|/|z|= 1.0701697316609505809 × 10^-16
```

FIGURE 2.6 – Fichier d'erreur à l'étape 3

Encore on a une baisse de l'erreur absolue 10^{-28} mais l'erreur relative a tendance à remonter légèrement mais elle reste très faible.

2.6 Etape 4

```
R(1) = -1.8496094094546590994 × 10^-9
R(1)^2 = 3.4210549675432127775 × 10^-18

R(2) = -1.0549208854453946002 × 10^-9
R(2)^2 = 1.1128580745488953568 × 10^-18

R(3) = -7.5272766610903494634 × 10^-10
R(3)^2 = 5.6659893932595479735 × 10^-19

R(4) = -5.8876035491499310013 × 10^-10
R(4)^2 = 3.4663875551962863992 × 10^-19

R2= Z = 5.4471507369376917257 × 10^-18

z = R(1)^2+R(2)^2+R(3)^2+R(4)^2 = 5.44715073693769157157 × 10^-18
on ramène z au même nombre de chiffres significatifs que Z :
z = 5.4471507369376915716 × 10^-18

On calcule l'erreur absolue :
|Z-z|= 1.541 × 10^-34

On calcule l'erreur relative :
|Z-z|/|z|= 2.8290019395834227425 × 10^-17
```

FIGURE 2.7 – Fichier d'erreur à l'étape 4

Encore une fois on a l'erreur absolue 10^{-32} qui diminue mais l'erreur relative reste aux alentours de 10^{-17} comme depuis le début.

2.7 postambule

On a pu voir que l'erreur absolue avait tendance à diminuer et l'erreur relative à stagner autour de 10^{-17}. Il est en fait logique que l'erreur absolue diminue car le vecteur résidu diminue à chaque itération du programme normalement. C'est pourquoi il est souvent plus judicieux de regarder l'erreur relative qui est plus représentative.
A ce propos elle est plutôt très faible et on peut être satisfait du résultat obtenue.

3. Vecteur solution x

Intéressons nous maintenant au vecteur solution x. On rappelle que la solution théorique attendue est le vecteur :

$$x^T = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

Dans cette partie nous allons étudié les erreurs locales sur le vecteur solution $\hat{x}$ obtenu avec la méthode du gradient conjugué :

$$\hat{x}_{n+1} = \hat{x}_n - \alpha * p_n$$

avec

$$\alpha = \frac{(r_n, r_n)}{(Ap_n, p_n)}$$

On se propose d'étudier chaque composante du vecteur sur 4 étapes.

3.1 Etape 1

Pour chaque étape, on récupère les données en binaire grâce à *gdb* puis on les stock dans des fichiers comme ci-dessous :

```
(gdb) print x  //x_0
$6 = (0, 0, 0, 0)
(gdb) x/tg &x(1)
0x555555561af0:        0000000000000000000000000000000000000000000000000000000000000000
(gdb) x/tg &x(2)
0x555555561af8:        0000000000000000000000000000000000000000000000000000000000000000
(gdb) x/tg &x(3)
0x555555561b00:        0000000000000000000000000000000000000000000000000000000000000000
(gdb) x/tg &x(4)
0x555555561b08:        0000000000000000000000000000000000000000000000000000000000000000
```

FIGURE 3.1 – Composantes de x à l'étape 0

```
(gdb) print alpha
$7 = 0.66806923697271459
(gdb) x/tg &alpha
0x7fffffffde70:        0011111111100101011000001101001010111100100010000101111000010010
```

FIGURE 3.2 – Valeur de α à l'étape 0

```
(gdb) print p  //p_0
$8 = (-2.083333333333333, -1.2833333333333332, -0.94999999999999984, -0.75952380952380949)
(gdb) x/tg &p(1)
0x555555561b80:        1100000000000000101010101010101010101010101010101010101010101010
(gdb) x/tg &p(2)
0x555555561b88:        1011111111110100100010001000100010001000100010001000100010001000
(gdb) x/tg &p(3)
0x555555561b90:        1011111111101110011001100110011001100110011001100110011001100101
(gdb) x/tg &p(4)
0x555555561b98:        1011111111101000010011100000010011100000010011100000010011100000
```

FIGURE 3.3 – Composantes de p à l'étape 0

```
(gdb) print x   //x_1
$9 = (1.391810910359822, 0.85735552078165034, 0.63466577512407873, 0.50741449189118082)
(gdb) x/tg &x(1)
0x555555561af0:       0011111111110110010001001101101110000100011000110110000111111101
(gdb) x/tg &x(2)
0x555555561af8:       0011111111101011011011110111010011011000010110011010101111101100
(gdb) x/tg &x(3)
0x555555561b00:       0011111111100100010011110010111010011001100000011000110010010000
(gdb) x/tg &x(4)
0x555555561b08:       0011111111100000001111001011110101010001000001100000110011101111
```

FIGURE 3.4 – Composantes de x à l'étape 1

Une fois les valeurs binaires obtenues, on peut alors calculer leur valeur. Soit par l'ordinateur qui est en double précision donc qui aura des erreurs de l'ordre de 10^{-16}, soit par un plus grand calculateur [1], c'est ce qu'on se propose de faire.

On stock également ces résultats dans des fichiers de manière à les comparer plus tard comme dans l'exemple ci-dessous :

```
x_1(4):
0011111111100000001111001011110101010001000001100000110011101111

 bit de signe : 0
 bits d''exposants : 01111111110
 bits de mantisses : 0000001111001011110101010001000001100000110011101111

 Le nombre est positif car le bit de signe est à 0 ; normalisé car les bits d'exposants sont
différents de : 000..0

 exposant biaisé =
2^( 9)+2^( 8)+2^( 7)+2^( 6)+2^( 5)+2^( 4)+2^( 3)+2^( 2)+2^( 1)+0 = 1022

exposant réel = 1022-d = 1022-1023 =   -1

 mantisse =
2^(- 7) + 2^(- 8) + 2^(- 9) + 2^(-10) + 2^(-13) + 2^(-15) + 2^(-16) + 2^(-17) + 2^(-18) + 2^(-20) +
2^(-22) + 2^(-24) + 2^(-28) + 2^(-34) + 2^(-35) + 2^(-41) + 2^(-42) + 2^(-45) + 2^(-46) + 2^(-47) +
2^(-49) + 2^(-50) + 2^(-51) + 2^(-52) +  0 =  66783805836527/4503599627370496

Valeur décimale exacte : ((-1)^0)*(1+66783805836527/4503599627370496)*(2^(  -1)) =
0.50741449189118081886
```

FIGURE 3.5 – Calcul précis de la 4^{eme} composante de x à l'étape 1

Une fois le calcul précis de toutes les composantes de nos variables, on pourra alors comparer nos valeurs en local de manière à obtenir nos erreurs absolues et relatives. On obtient un fichier du type :

```
x_0 = (0,0,0,0)

alpha = 0.66806923697271458629

p_0= (-2.0833333333333330373,-1.2833333333333332149,-0.94999999999999984457,-0.75952380952380948997)

x_1 = Z = (1.3918109103598219622,0.85735552078165033763,0.63466577512407873485,0.50741449189118081886)

z = x_0 - alpha*p_0 = (1.3918109103598218570, 0.8573555207816503066, 0.6346657751240787531, 0.50741449189118081793)

On calcule l'erreur absolue pour chaque composante, afin d'obtenir le vecteur erreur absolue:
|Z-z|=(1.052×10^-16, 3.103×10^-17, 1.825×10^-17, 9.3×10^-19)

On calcule l'erreur relative pour chaque composante, afin d'obtenir le vecteur erreur relative:
|Z-z|/|z|= (7.5584980126936116695×10^-17,3.6192686986735648614×10^-17,2.8755292494592258677×10^-17,1.8328211252575854608×10^-18)
```

FIGURE 3.6 – Fichier erreur final obtenu pour l'étape 1

On s'intéresse seulement à ces deux lignes pour conclure :

1. *wolframalpha.com*

```
|Z-z|=(1.052×10^-16, 3.103×10^-17, 1.825×10^-17, 9.3×10^-19)
```

Figure 3.7 – Vecteur erreur absolue pour l'étape 1

```
|Z-z|/|z|= (7.5584980126936116695×10^-17,3.6192686986735648614×10^-17,2.8755292494592258677×10^-17,1.8328211252575854608×10^-18)
```

Figure 3.8 – Vecteur erreur relative pour l'étape 1

On remarque que l'erreur relative pour toutes les composantes de $\hat{x}$ est inférieure à l'erreur machine 10^{-16} ce qui est plutôt bien, les calculs se sont bien passés informatiquement.

3.2 Etape 2

De même pour l'étape 2, on obtient les valeurs des différentes variables en binaire, nous calculons leur valeur exacte grâce à un calculateur [2] très performant afin d'avoir une très grande précision puis on compare le résultat final Z que donne le programme avec celui calculé sur le calculateur z. On obtient ces résultats :

```
|Z-z|=(7.91×10^-18, 5.3×10^-17, 4.6×10^-17, 7.26×10^-18)
```

Figure 3.9 – Vecteur erreur absolue pour l'étape 2

```
|Z-z|/|z|= (8.1502959739600690275×10^-18,4.7438527536177694312×10^-17,4.5948761769806162385×10^-17,8.2393453677395306607×10^-18)
```

Figure 3.10 – Vecteur erreur relative pour l'étape 2

Les erreurs sont de l'ordre de 10^{-18} bien inférieur à la précision machine 10^{-16}. Les calculs se sont également bien passés.

3.3 Etape 3

Même chose pour l'étape 3, on obtient :

```
|Z-z|= (4.91 × 10^-17,4.08 × 10^-17,2.25 × 10^-17,3.69 × 10^-17)
```

Figure 3.11 – Vecteur erreur absolue pour l'étape 3

```
|Z-z|/|z|= (4.9068020752442432596×10^-17,4.1101587690637775815×10^-17,2.2109434838433733709×10^-17,3.7328716479168916341×10^-17)
```

Figure 3.12 – Vecteur erreur relative pour l'étape 3

Les erreurs sont cette fois-ci de l'ordre de 10^{-17}, toujours inférieur à 10^{-16} ce qui indique que les calculs se sont bien passés.

3.4 Etape 4

Même démarche pour cette dernière étape, on obtient :

2. sur *wolframalpha.com*

```
|Z-z|= (2.57 × 10^-17,4.94×10^-17,4.9 × 10^-18,5.14 × 10^-17)
```

Figure 3.13 – Vecteur erreur absolue pour l'étape 4

```
|Z-z|/|z|= (2.5700000031647810540×10^-17,4.9400000034808270286×10^-17,4.9000000024584155077×10^-18,5.1400000020277781816×10^-17)
```

Figure 3.14 – Vecteur erreur relative pour l'étape 4

Les erreurs sont de l'ordre de 10^{-17} une nouvelle fois, on conclut alors que l'ensemble des calculs pour les 4 étapes se sont bien déroulés informatiquement.

4. Analyse numérique du problème

Analysons maintenant le problème numériquement. On sait que la solution théorique est

$$x^T = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

Comparons maintenant ce résultat avec celui qu'on a obtenu numériquement au bout de la 4^{eme} étape :

$$\hat{x}^T = \begin{pmatrix} 0.9999999987685677105 & 0.9999999992953791939 & 0.9999999994982825546 & 0.99999999960549057491 \end{pmatrix}$$

On peut maintenant obtenir l'erreur absolue pour chaque composante afin d'obtenir le vecteur absolu : [1]

$$\varepsilon_{abs} = \begin{pmatrix} 1.2314322895 * 10^{-9} & 7.046208061 * 10^{-10} & 5.017174454 * 10^{-10} & 3.9450942509 * 10^{-10} \end{pmatrix}$$

On remarque que l'on obtient le même vecteur pour le vecteur erreur relative puisqu'on divise toutes les composantes par 1 :

$$\varepsilon_{rel} = \begin{pmatrix} 1.2314322895 * 10^{-9} & 7.046208061 * 10^{-10} & 5.017174454 * 10^{-10} & 3.9450942509 * 10^{-10} \end{pmatrix}$$

On observe des erreurs beaucoup plus élevées que celles obtenues localement. En effet pour une étude local on obtenait des erreurs d'ordre de grandeur d'environ 10^{-17} alors qu'ici il est d'environ 10^{-10}. Une différence de 10^7 qui n'est pas négligeable.

Cependant, on peut souligner l'efficacité de la méthode car en seulement 4 itérations l'erreur de la solution obtenue par rapport à celle théorique est de seulement 10^{-9}. Si l'on veut obtenir plus de précision il suffit de diminuer la tolérance et d'observer davantage d'étapes.

1. calcul effectué sur *wolframalpha.com*

Conclusion

Annexe