Théophile Avenel,
Mathis Jamin

SRT3 - Travaux pratiques Méthodes Numérique

TP2 Méthodes numériques : Équations non linéaires

RÉSUMÉ

TP2 Méthodes numériques : Équations non linéaires

SOMMAIRE

Théophile Avenel,
Mathis Jamin

Introduction

Dans ce TP, nous allons explorer la résolution des équations non linéaires et l'analyse des systèmes linéaires du deuxième ordre. Nous débuterons par l'estimation numérique des racines d'une fonction non linéaire à l'aide de trois méthodes différentes : Bolzano, la sécante, et Newton. Nous tracerons ensuite les courbes des nombres d'itérations en fonction de l'imprécision pour comparer la vitesse de convergence de chaque méthode. Par la suite, nous analyserons la réponse temporelle d'un système linéaire du deuxième ordre, en utilisant différentes méthodes pour déterminer le temps de réponse. Nous examinerons également une question bonus impliquant la génération et l'analyse de réponses indicielles pour divers coefficients d'amortissement.

Résolution d'une équation non-linéaire simple

Nous allons commencer par utiliser le programme bolza.c. Nous pouvons voir sur la figure 1, le résultat de la racine avec le résultat en 31 étapes.

gcc bolza.c -o bolza -lm
./bolza 1e-9

```
┌──(root㉿kali-athena)-[/home/athenista/Bureau/polytechNantes/Tp/tpMethodesNumériques/tpEquationsNonLineaires/Sources]
└─# ./bolza 1e-9
Estimation initiale : 7.500000e-01, imprecision 7.500000e-01
Etape 1 : estimation = 3.750000000000e-01, imprecision = 3.750000000000e-01
Etape 2 : estimation = 1.875000000000e-01, imprecision = 1.875000000000e-01
Etape 3 : estimation = 2.812500000000e-01, imprecision = 9.375000000000e-02
Etape 4 : estimation = 2.343750000000e-01, imprecision = 4.687500000000e-02
Etape 5 : estimation = 2.578125000000e-01, imprecision = 2.343750000000e-02
Etape 6 : estimation = 2.695312500000e-01, imprecision = 1.171875000000e-02
Etape 7 : estimation = 2.636718750000e-01, imprecision = 5.859375000000e-03
Etape 8 : estimation = 2.607421875000e-01, imprecision = 2.929687500000e-03
Etape 9 : estimation = 2.622070312500e-01, imprecision = 1.464843750000e-03
Etape 10 : estimation = 2.629394531250e-01, imprecision = 7.324218750000e-04
Etape 11 : estimation = 2.633056640625e-01, imprecision = 3.662109375000e-04
Etape 12 : estimation = 2.631225585938e-01, imprecision = 1.831054687500e-04
Etape 13 : estimation = 2.632141113281e-01, imprecision = 9.155273437500e-05
Etape 14 : estimation = 2.632598876953e-01, imprecision = 4.577636718750e-05
Etape 15 : estimation = 2.632827758789e-01, imprecision = 2.288818359375e-05
Etape 16 : estimation = 2.632713317871e-01, imprecision = 1.144409179688e-05
Etape 17 : estimation = 2.632656097412e-01, imprecision = 5.722045898438e-06
Etape 18 : estimation = 2.632627487183e-01, imprecision = 2.861022949219e-06
Etape 19 : estimation = 2.632641792297e-01, imprecision = 1.430511474609e-06
Etape 20 : estimation = 2.632648944855e-01, imprecision = 7.152557373047e-07
Etape 21 : estimation = 2.632645368576e-01, imprecision = 3.576278686523e-07
Etape 22 : estimation = 2.632647156715e-01, imprecision = 1.788139343262e-07
Etape 23 : estimation = 2.632648050785e-01, imprecision = 8.940696716309e-08
Etape 24 : estimation = 2.632647603750e-01, imprecision = 4.470348358154e-08
Etape 25 : estimation = 2.632647827268e-01, imprecision = 2.235174179077e-08
Etape 26 : estimation = 2.632647939026e-01, imprecision = 1.117587089539e-08
Etape 27 : estimation = 2.632647994906e-01, imprecision = 5.587935447693e-09
Etape 28 : estimation = 2.632647966966e-01, imprecision = 2.793967723846e-09
Etape 29 : estimation = 2.632647980936e-01, imprecision = 1.396983861923e-09
Etape 30 : estimation = 2.632647973951e-01, imprecision = 6.984919309616e-10
Etape 31 : estimation = 2.632647977443e-01, imprecision = 3.492459654808e-10
Resultat en 31 etapes : estimation = 2.632647977443e-01, imprecision = 3.492459654808e-10
```

Figure 1 : Exécution du code bolza.c

Nous allons maintenant utiliser le programme sec.c. Nous pouvons voir sur la figure 2, le résultat de la racine avec le résultat en 8 étapes.

gcc sec.c -o sec -lm
./sec 1e-9

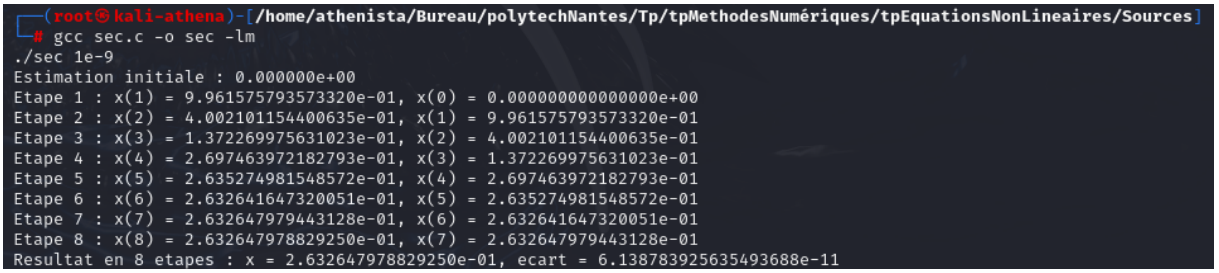


Figure 2 : Exécution du code sec.c

Nous allons utiliser le programme new.c. Nous pouvons voir sur la figure 3, le résultat de la racine avec le résultat en 6 étapes.

gcc new.c -o new -lm
./new 1e-9

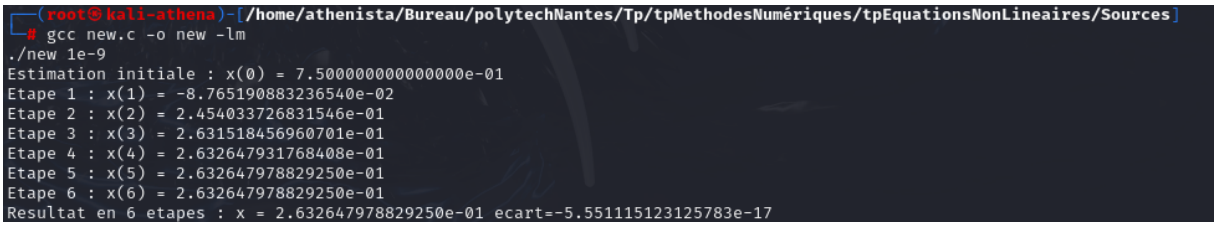


Figure 3 : Exécution du code new.c

2) Nous allons tracer la courbe des nombres d'itérations en fonction de l'imprécision en échelle logarithmique.Nous allons faire varier ε entre 10−1 et 10−12.

Nous avons commencé par écrire un petit script.

```
#!/bin/bash
# Fichier de sortie pour les résultats
output_file="bolza_iterations.dat"

# Supprimer le fichier de sortie s'il existe déjà
rm -f $output_file

# Exécuter le programme pour chaque valeur de epsilon
for e in 1e-1 1e-2 1e-3 1e-4 1e-5 1e-6 1e-7 1e-8 1e-9 1e-10 1e-11 1e-12
do
        # Exécuter le programme et capturer le nombre d'itérations
        iterations=$(./bolza $e)
        # Écrire epsilon et le nombre d'itérations dans le fichier de sortie
```

```
        echo "$e $iterations" >> $output_file
done
```

Nous exécutons ensuite notre script.

```
chmod +x run_bolza.sh
./run_bolza.sh
```

Nous prenons ensuite le .dat et nous le simplifions à la main avec.

```
#Tracez la courbe des nombres d'itérations en fonction de l'imprécision en échelle logarithmique
#iterations  imprecision
4       4.687500000000e-02 #1e-1
8       2.929687500000e-03 #1e-2
11      3.662109375000e-04 #1e-3
14      4.577636718750e-05 #1e-4
18      2.861022949219e-06  #1e-5
21      3.576278686523e-07 #1e-6
24      4.470348358154e-08 #1e-7
28      2.793967723846e-09 #1e-8
31      3.492459654808e-10 #1e-9
34      4.365574568510e-11 #1e-10
38      2.728484105319e-12 #1e-11
41      3.410605131648e-13 #1e-12
```

Nous utilisons ensuite quelques commandes gnuplot.

```
set terminal pngcairo size 800,600
set output 'bolza_iterations.png'
set logscale x 10
set xlabel "Nombre d'iterations"
set ylabel "Epsilon"
set title "Nombre d'iterations en fonction de l'imprecision"
plot "bolza.txt" w l
```

La méthode de Bolzano divise l'intervalle en deux à chaque itération. Nous pouvons voir qu'avec la Bolzano la courbe converge rapidement vers 0 car les premières itérations réduisent rapidement la taille de l'intervalle. De l'itération 4 à 10, elle décroit fortement. A mesure que l'intervalle devient plus petit, chaque division en deux apporte une réduction proportionnellement plus faible, ce qui ralentit le taux de diminution des itérations restantes. (convergence linéaire)

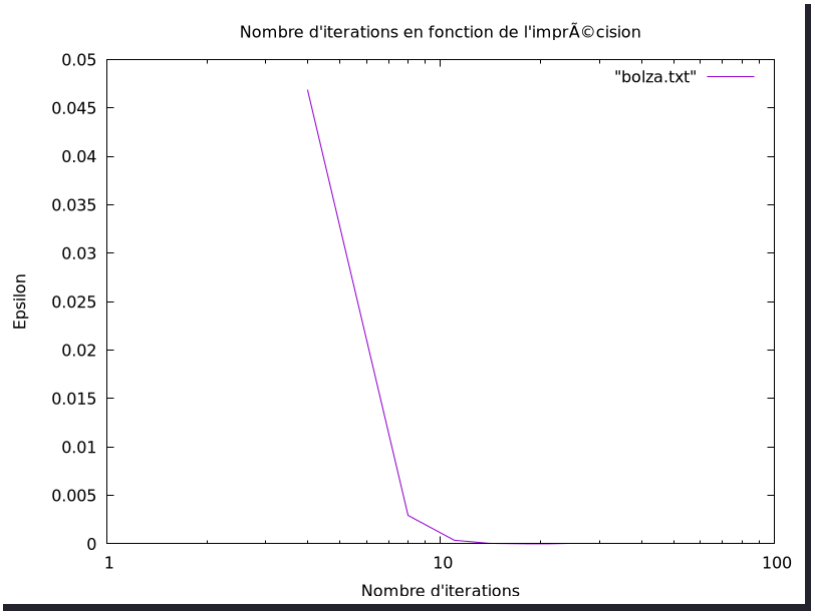


Figure 4 : Courbe d'itérations en fonction de l'imprécision avec le code bolza.c

Nous modifions ensuite le script run_bolza.sh pour créer le script run_sec.sh

chmod +x run_sec.sh
./run_sec.sh

5 6.218899063422100859e-03 #1e-1
5 6.218899063422100859e-03 #1e-2
6 2.633334228520634035e-04 #1e-3
7 6.332123077545936951e-07 #1e-4
7 2.632641647320051e-01 #1e-5
7 6.332123077545936951e-07 #1e-6
8 6.138783925635493688e-11 #1e-7
8 6.138783925635493688e-11 #1e-8
8 6.138783925635493688e-11 #1e-9
8 6.138783925635493688e-11 #1e-10
9 5.551115123125782702e-17 #1e-11
9 5.551115123125782702e-17 #1e-12

Nous utilisons ensuite quelques commandes gnuplot.
set terminal pngcairo size 800,600
set output 'sec_iterations.png'
set logscale x 10
set xlabel "Nombre d'iterations"
set ylabel "Epsilon"
set title "Nombre d'iterations en fonction de l'imprecision"
plot "sec.txt" w l

La méthode de la sécante utilise deux points pour approximer la dérivée et trouve une meilleure approximation de la racine à chaque étape. Cette méthode converge plus rapidement que la méthode de Bolzano. La courbe de la méthode de la sécante est moins raide que celle de Bolzano, indiquant une convergence plus rapide (convergence superlinéaire).

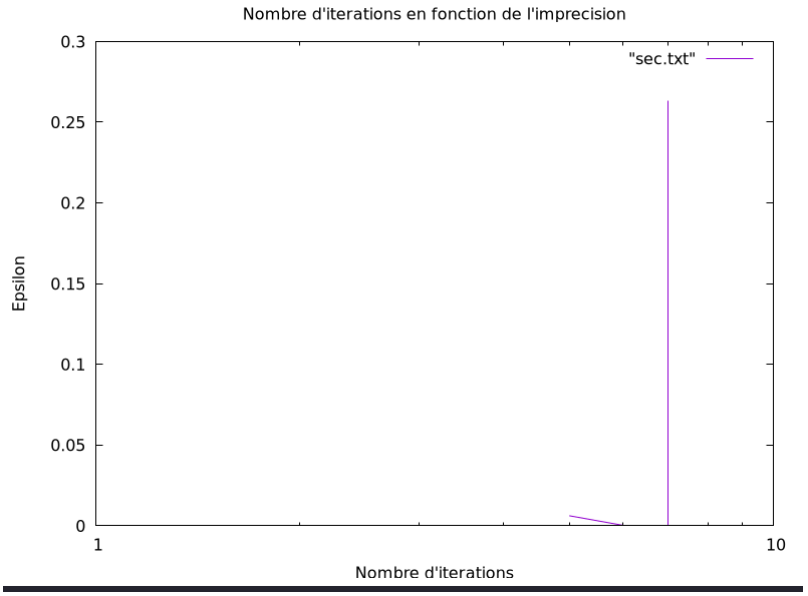


Figure 5 : Courbe d'itérations en fonction de l'imprécision avec le code sec.c

Nous modifions ensuite le script run_bolza.sh pour créer le script run_sec.sh.

```
chmod +x run_new.sh
./run_new.sh
3 1.774847301291554e-02 #1e-1
4 1.129474807706754e-04 #1e-2
4 1.129474807706754e-04 #1e-3
5 4.706084189010085e-09 #1e-4
5 4.706084189010085e-09 #1e-5
5 4.706084189010085e-09 #1e-6
5 4.706084189010085e-09 #1e-7
5 4.706084189010085e-09 #1e-8
6 -5.551115123125783e-17 #1e-9
6 -5.551115123125783e-17 #1e-10
6 -5.551115123125783e-17 #1e-11
6 -5.551115123125783e-17 #1e-12
```

Nous utilisons ensuite quelques commandes gnuplot.

```
set terminal pngcairo size 800,600
set output 'new_iterations.png'
set logscale x 10
set xlabel "Nombre d'iterations"
set ylabel "Epsilon"
```

set title "Nombre d'iterations en fonction de l'imprecision"
plot "new.txt" w l

La méthode de Newton-Raphson utilise la dérivée de la fonction pour trouver une meilleure approximation de la racine. La courbe montre une décroissance très rapide, plus rapide que celle de la méthode de la sécante. (convergence quadratique)

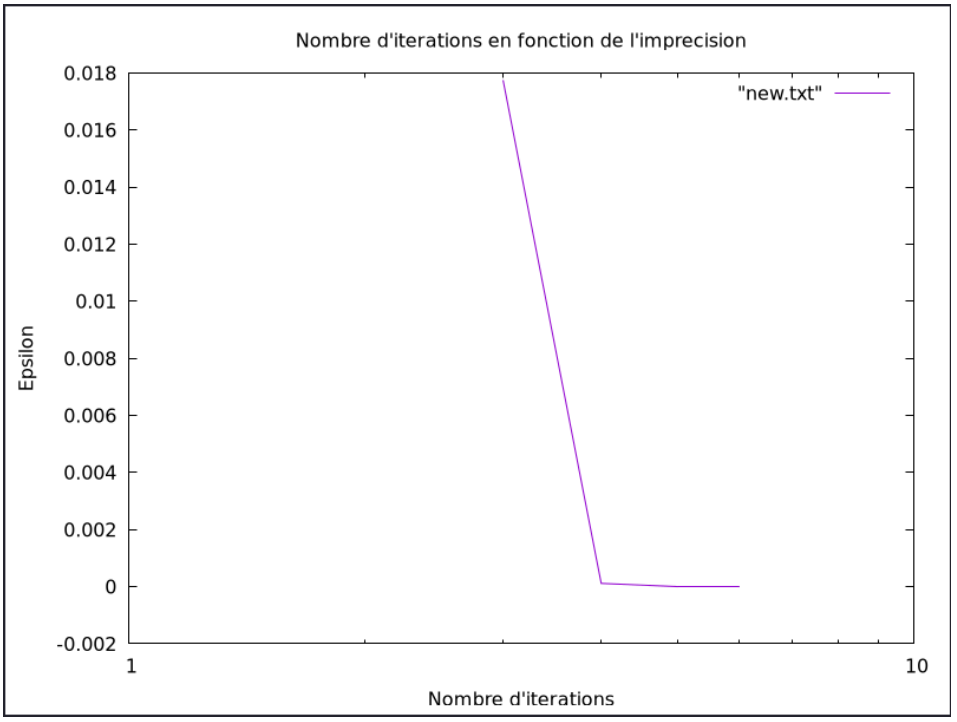


Figure 6 : Courbe d'itérations en fonction de l'imprécision avec le code new.c

Système linéaire du deuxième ordre

1) Nous avons ajouté des commentaires au fichier tr.c.

```
33  // fonctions reponse a un echelon : fg pour z>1, fu pour z=1 et fp pour z<1
34  // un seul argument, le temps, l'amortissement est une variable globale
35  // Fonctions de réponse à un échelon
36  double fg(double); // pour z > 1
37  double fu(double); // pour z = 1
38  double fp(double); // pour z < 1
39
40  // Fonctions de réponse à une impulsion, dérivées des précédentes
41  double dfg(double);
42  double dfu(double);
43  double dfp(double);
44
45  // fonction a resoudre par les methodes de recherche de racine et sa derivee
46  double equa(double);
47  double dequa(double);
```

Figure 7 : Code commenté source tr.c

```c
// Fonctions de réponse à un échelon pour z > 1
double fg(double t) {
    if (t >= 0) return (1.0 - exp(-z*t) * sinh(c*t + log(z + c)) / c);
    else return 0;
}

// Dérivée de fg
double dfg(double t) {
    if (t >= 0) return (exp(-z*t) * sinh(c*t) / c);
    else return 0;
}

// Fonctions de réponse à un échelon pour z = 1
double fu(double t) {
    if (t >= 0) return (1.0 - (1.0 + t) * exp(-t));
    else return 0;
}

// Dérivée de fu
double dfu(double t) {
    if (t >= 0) return (t * exp(-t));
    else return 0;
}

// Fonctions de réponse à un échelon pour z < 1
double fp(double t) {
    if (t >= 0) return (1.0 - exp(-z*t) * sin(c*t + atan(c/z)) / c);
    else return 0;
}
```

Figure 8 : Code commenté source tr.c

```c
// Dérivée de fu
double dfu(double t) {
    if (t >= 0) return (t * exp(-t));
    else return 0;
}

// Fonctions de réponse à un échelon pour z < 1
double fp(double t) {
    if (t >= 0) return (1.0 - exp(-z*t) * sin(c*t + atan(c/z)) / c);
    else return 0;
}

// Dérivée de fp
double dfp(double t) {
    if (t >= 0) return (exp(-z*t) * sin(c*t) / c);
    else return 0;
}

// Fonction à résoudre par les méthodes de recherche de racine
double equa(double t) {
    if (z < 1) return (1.0 - fp(t) - dec);
    else if (z == 1) return (1.0 - fu(t) - dec);
    else return (1.0 - fg(t) - dec);
}

// Dérivée de la fonction à résoudre
double dequa(double t) {
    if (z < 1) return (-dfp(t));
    else if (z == 1) return (-dfu(t));
    else return (-dfg(t));
}
```

Figure 9 : Code commenté source tr.c

2) Nous donnons l'équation qui est résolue dans le code « tr.c » en fonction de sY (t) et δ.

$$\text{equa}(t) = \begin{cases} 1.0 - \text{fp}(t) - \text{dec}, & \text{si } z < 1 \\ 1.0 - \text{fu}(t) - \text{dec}, & \text{si } z = 1 \\ 1.0 - \text{fg}(t) - \text{dec}, & \text{si } z > 1 \end{cases}$$

Figure 10 : Équation résolue

L'équation que le code résout pour déterminer le temps de réponse t_r est basée sur la fonction de réponse à un échelon s(t) du système du deuxième ordre. Le but est de trouver le temps t tel que la réponse atteigne une certaine fraction δ de sa valeur finale.

Pour un système du deuxième ordre, la réponse à un échelon dépend de l'amortissement z.
L'équation générale que le code cherche à résoudre est :

$$1 - s(t) - \delta = 0$$

où s(t) est la réponse du système qui varie en fonction de l'amortissement z :

Pour z<1 (sous-amorti) :

$$s(t) = 1 - \frac{e^{-zt}\sin(ct + \arctan(\frac{c}{z}))}{c}$$

Pour z=1 (critique) :

$$s(t) = 1 - (1+t)e^{-t}$$

Pour z>1 (sur-amorti) :

$$s(t) = 1 - \frac{e^{-zt}\sinh(ct + \log(z+c))}{c}$$

Validation:

Pour z<1

$$\text{equa}(t) = 1.0 - \left(1 - \frac{e^{-zt}\sin(ct + \arctan(\frac{c}{z}))}{c}\right) - \delta$$

$$\text{equa}(t) = \frac{e^{-zt}\sin(ct + \arctan(\frac{c}{z}))}{c} - \delta$$

Pour z=1 :

$$\text{equa}(t) = 1.0 - \left(1 - (1+t)e^{-t}\right) - \delta$$

$$\text{equa}(t) = (1+t)e^{-t} - \delta$$

Pour z>1 :

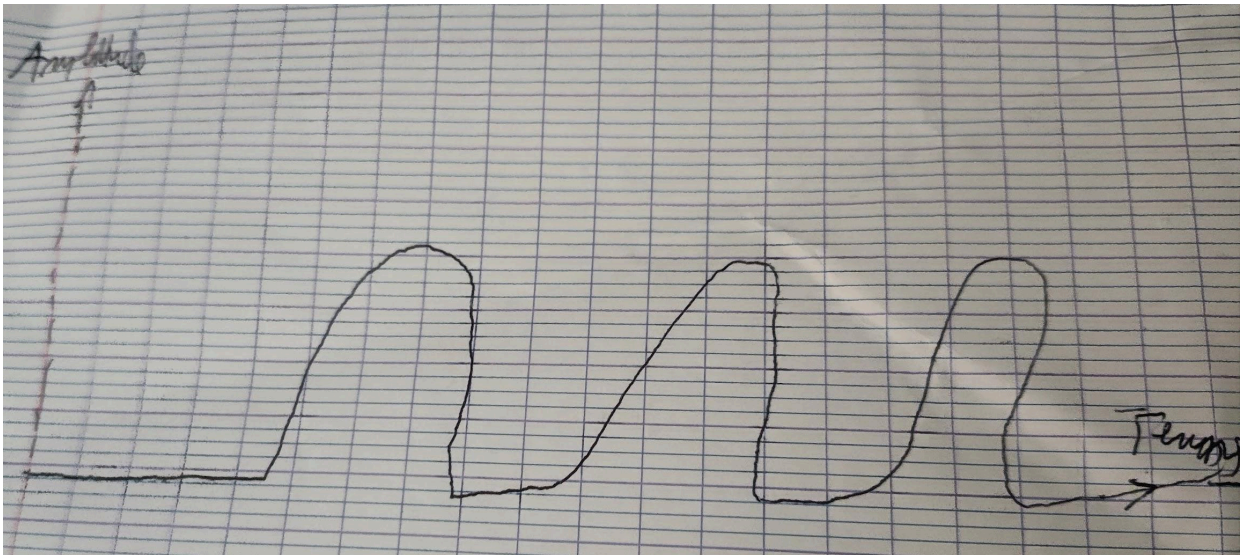
$$\text{equa}(t) = 1.0 - \left(1 - \frac{e^{-zt}\sinh(ct + \log(z+c))}{c}\right) - \delta$$

$$\text{equa}(t) = \frac{e^{-zt}\sinh(ct + \log(z+c))}{c} - \delta$$

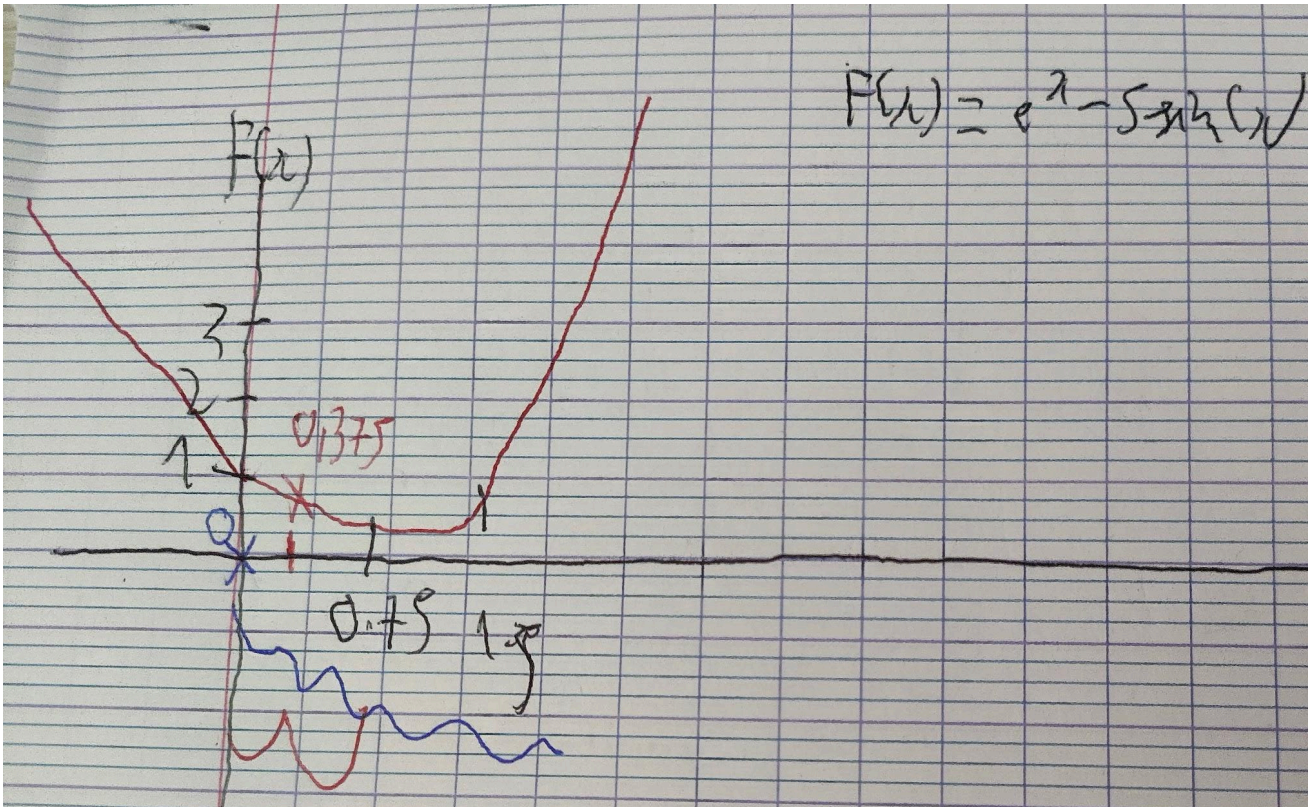
La fonction equa(t) retourne bien $1.0 - s(t) - \delta$ pour les trois cas $z<1$, $z=1$ et $z>1$, ce qui est conforme à l'équation que nous cherchons à résoudre.

3a) Dessinez à la main une courbe de réponse indicielle non amortie et 2 itérations de

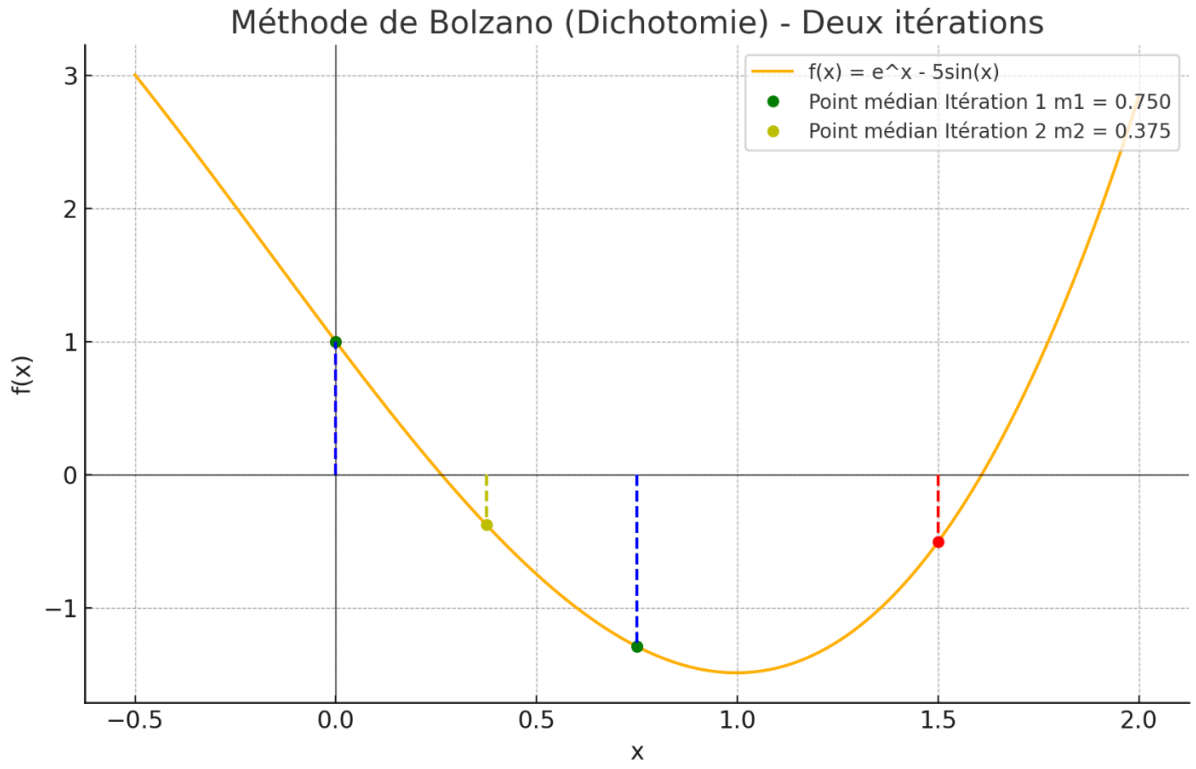
l'algorithme Bolzano. À FAIRE



Théophile Avenel,
Mathis Jamin



Au propre, cela nous donne:



1. **Intervalle initial** $[0, 1.5]$:

La fonction $f(x)$ doit croiser l'axe des abscisses entre $a = 0$ et $b = 1.5$.

Le point médian est $m_1 = \frac{0+1.5}{2} = 0.75$.

2. **Première itération** :

Nous allons commencer par évaluer $f(0.75)$.

Étant donné que $f(0.75) \approx -0.018$ (négatif), le nouvel intervalle sera $[0, 0.75]$.

3. **Deuxième itération** :

Le nouvel intervalle est $[0, 0.75]$, le prochain point médian est $m_2 = \frac{0+0.75}{2} = 0.375$.

Nous allons donc évaluer $f(0.375)$.

Étant donné que $f(0.375) \approx 0.245$ (positif), le nouvel intervalle sera $[0, 0.375]$.

3b) Nous allons exécuter tr en utilisant la méthode de Bolzano.

gcc tr.c racnl.c -o tr -lm
./tr 0.02 0 bolz_002

```
┌──(root㉿kali-athena)-[/home/athenista/Bureau/polytechNantes/Tp/tpMethodesNumériques/tpEquationsNonLineaires/Sources]
└─# ./tr 0.02 0 bolz_002
Le nombre total d'iterations est : 21315
```

Figure 12 : Nombre d'itérations

gcc bin2ascii.c -o bin2ascii -lm
./bin2ascii bolz_002.bin

```
┌──(root㉿kali-athena)-[/home/athenista/Bureau/pol
└─# ./bin2ascii bolz_002.bin
nombre de couples lu: 1000
```

Figure 13 : Nombre de couples lu

gnuplot
plot "bolz_002.txt" w l

Voici la courbe d'itérations en fonction de l'imprécision avec la méthode de Bolzano. Nous pouvons voir qu'avec la Bolzano la courbe converge rapidement.

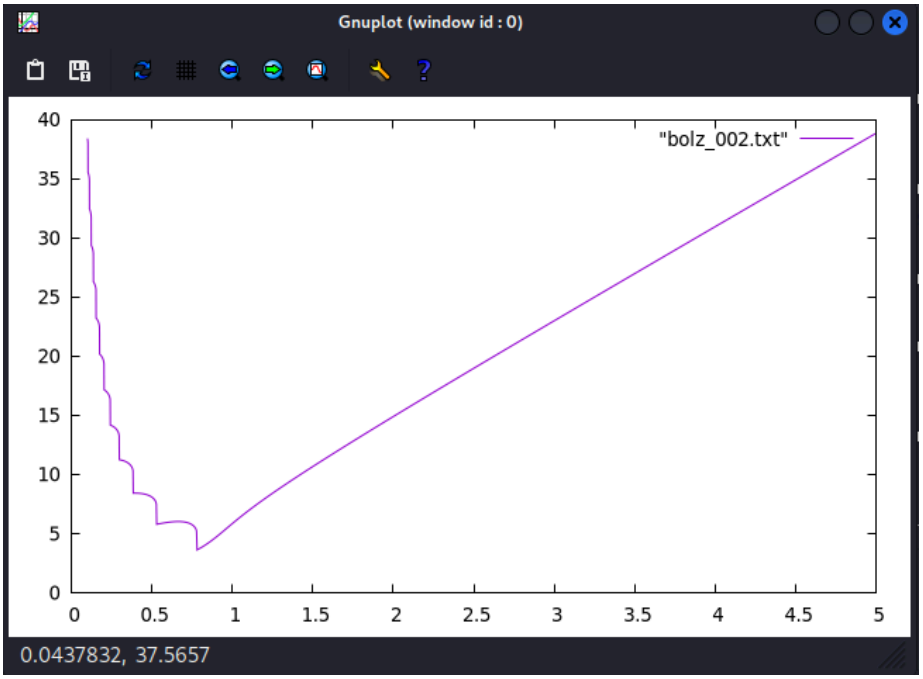
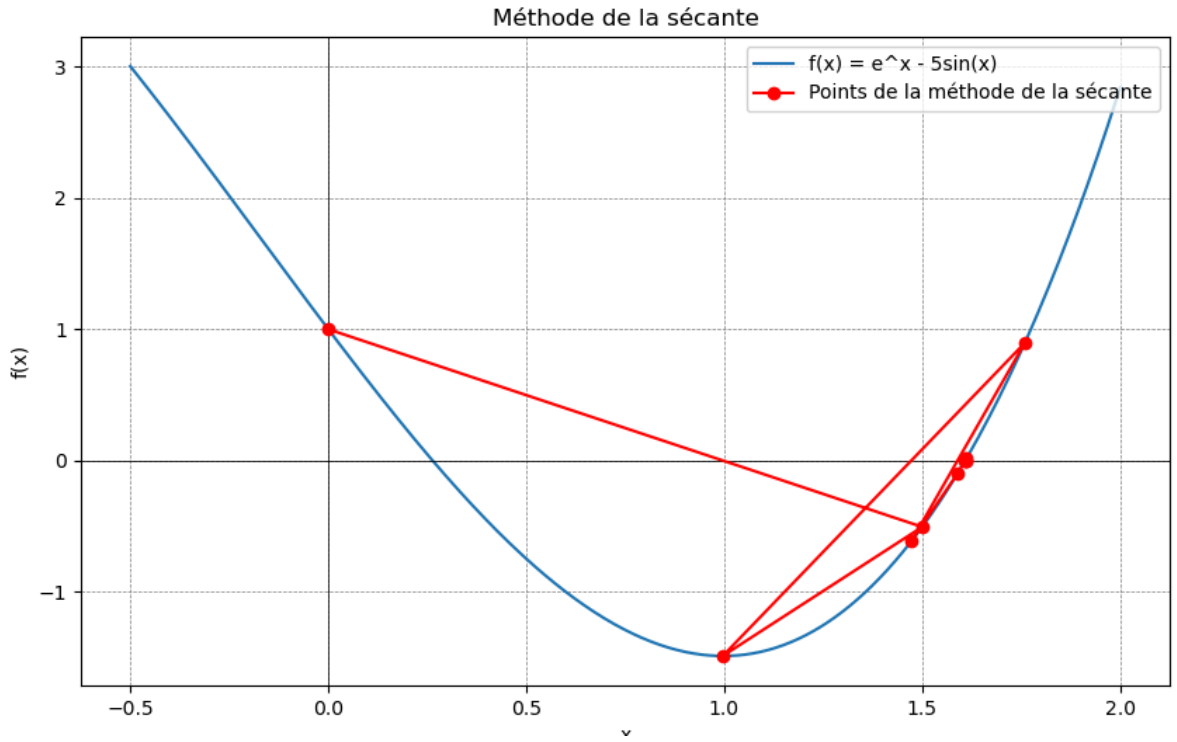


Figure 14 : Courbe avec la méthode de Bolzano

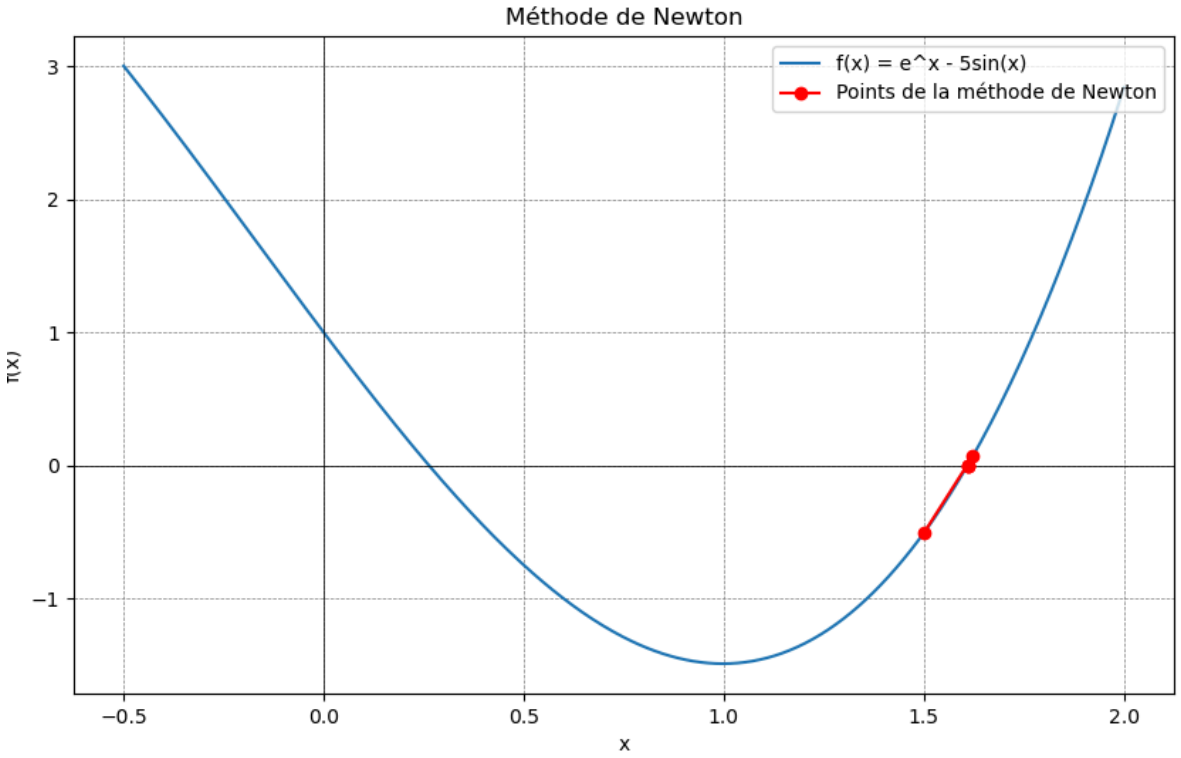
4a) Dessinez à la main une courbe de réponse indicielle non amortie et 2 itérations de l'algorithme de la sécante. A FAIRE



4b) Nous allons exécuter tr en utilisant la méthode de la sécante.

./tr 0.02 1 sec_002

5a) Dessinez à la main une courbe de réponse indicielle non amortie et 2 itérations de l'algorithme de la Newton. A FAIRE



5b) Nous allons exécuter tr en utilisant la méthode de Newton.

./tr 0,02 2 new_002

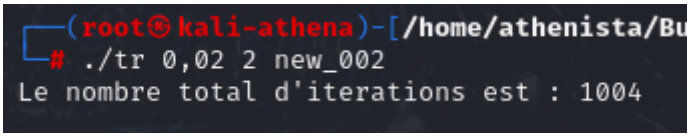


Figure : Nombre d'itérations

./bin2ascii new_002.bin

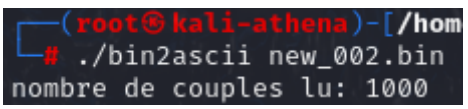
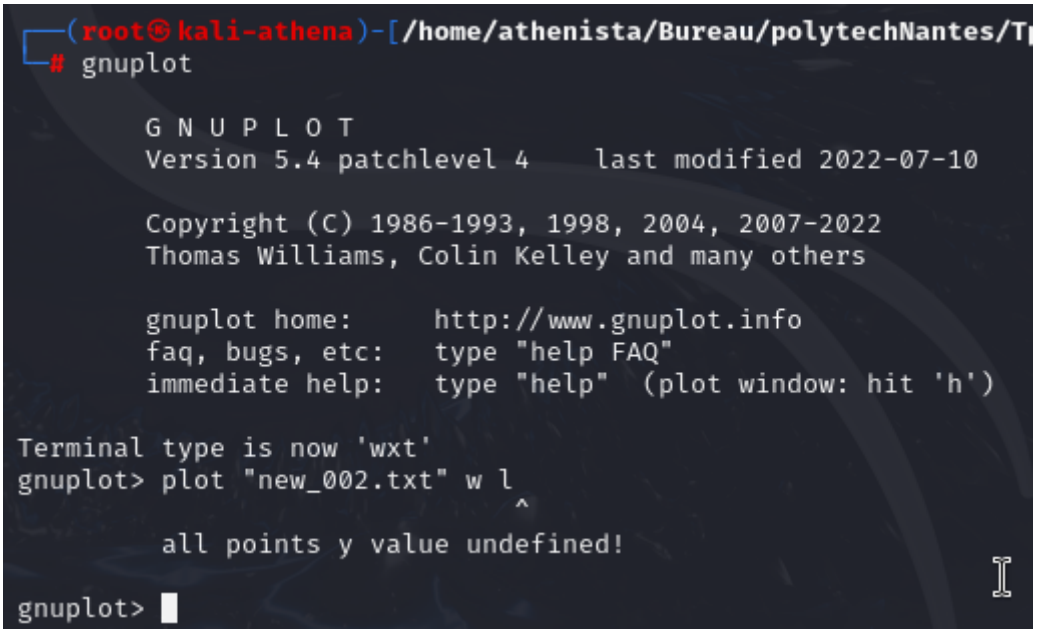


Figure : Nombre de couples lu



Conclusion

Ce TP a permis d'approfondir notre compréhension des méthodes numériques pour la résolution d'équations non linéaires et l'analyse des systèmes linéaires du deuxième ordre. En comparant les méthodes de Bolzano, de la sécante et de Newton, nous avons pu observer leurs vitesses de convergence variées : Bolzano avec une convergence linéaire, la sécante avec une convergence superlinéaire, et Newton avec une convergence quadratique, bien plus rapide mais plus sensible aux estimations initiales. L'analyse des systèmes linéaires a montré l'impact du coefficient d'amortissement sur la réponse temporelle, nous permettant de comprendre comment ajuster les paramètres pour obtenir les performances désirées.