Mathématiques 1
Méthodes numériques en Java

Institut Paul Lambin

19 novembre 2021

Théorie VS Pratique

Théorie

Recherche d'une racine de $f(x)$ sur un intervalle $[a,\ b]$

1. Hypothèses pour assurer l'existence de la racine $\rightarrow$ **Bolzano**
   - $f(x)$ est continue sur $[a,\ b]$
   - $f(a) \cdot f(b) < 0$

2. **Newton** : Hypothèses supplémentaires pour assurer la convergence
   - $f(x_0) \cdot f''(x_0) > 0$
   - $f'(x)$ de signe constant sur $[a,\ b]$
   - $f''(x)$ de signe constant sur $[a,\ b]$

Remarques :

1) Bolzano vérifiable avant de commencer les itérations
2) Newton :
   a) Hypothèse sur l'approximation initiale à vérifier avant les itérations
   b) Hypothèses sur les dérivées **à vérifier pendant les itérations** !

Théorie VS Pratique

Pratique

- Théorie : convergence si les hypothèses sont vérifiées !
  - $\rightarrow$ Problème : généralement infinité d'itérations pour atteindre la racine.

- Pratique :

1) Borne sur le nombre d'itérations : `MAX_ETAPE` (classe `Polynome`)
2) Critère d'arrêt : nombre de décimales exactes d :
   - **Bissection** : nombre d'itérations : $n \geq \log_2\left(10^d \cdot (b-a)\right)$
     - $\rightarrow$ calculable avant de commencer les itérations
     - $\rightarrow$ si $\log_2\left(10^d \cdot (b-a)\right) >$ `MAX_ETAPE` $\rightarrow$ `NumeriqueException`.
   - **Newton** : $d \leq \log_{10}\left(\dfrac{0.5 \cdot |f'(\text{extrémité fixe})|}{|f(x_n)|}\right)$
     - $\rightarrow$ à vérifier à chaque itération !
     - $\rightarrow$ si condition pas vérifiée après `MAX_ETAPE` itérations $\rightarrow$ `NumeriqueException`.

Théorie VS Pratique

Pratique : Instabilités Numériques

- Théorie : précision illimitée
- Sur machine :
  - → Précision limitée !
  - → Instabilités numériques !
  - → `NumeriqueException`

Théorie VS Pratique

Bissection : Instabilités Numériques

- Théorie : d décimales exactes après n itérations si $n \geq \log_2\left(10^d \cdot (b-a)\right)$

- Pratique :
  1) On prend $n =$ le plus petit entier plus grand que $n \geq \log_2\left(10^d \cdot (b-a)\right)$
  2) Si $n >$ `MAX_ETAPE` $\rightarrow$ `NumeriqueException`
  3) On fait n itérations de la bissection
  4) Après n itérations on vérifie l'erreur
     - $\rightarrow$ Si $\frac{b_n - a_n}{2} > 0.5 \cdot 10^{-d} \rightarrow$ `NumeriqueException`
     - $\rightarrow$ On peut simplifier par 0.5 : si $b_n - a_n > 10^{-d} \rightarrow$ `NumeriqueException`

Théorie VS Pratique

Newton : Instabilités Numériques

- Théorie : Si les hypothèses sont vérifiées alors
  1) convergence vers l'unique racine dans l'intervalle $[a, b]$
  2) $f(x_n)$ tend vers 0 de manière décroissante : $|f(x_{n+1})| \leq |f(x_n)| \ \forall n$.
  3) tous les x_n sont du même côté de la racine : $f(x_{n+1}) \cdot f(x_n) > 0 \ \forall n$
  4) x_n se rapproche de l'extrémité fixe à chaque itération :

$$|x_{n+1} - \textit{extrémité}_{\textit{fixe}}| \leq |x_n - \textit{extrémité}_{\textit{fixe}}| \ \forall n$$

- Pratique : A chaque itération on vérifie
  1) Les conditions supplémentaires de convergence :
     - $f'(x)$ de signe constant : si $f'(a) \cdot f'(x_n) < 0 \rightarrow$ `NumeriqueException`
     - $f''(x)$ de signe constant : si $f''(a) \cdot f''(x_n) < 0 \rightarrow$ `NumeriqueException`
  2) Les instabilités numériques
     - si $|f(x_{n+1})| > |f(x_n)| \rightarrow$ `NumeriqueException`
     - si $f(x_{n+1}) \cdot f(x_n) < 0 \rightarrow$ `NumeriqueException`
     - si $|x_{n+1} - \textit{extrémité}_{\textit{fixe}}| > |x_n - \textit{extrémité}_{\textit{fixe}}| \rightarrow$ `NumeriqueException`

La classe Polynome

La classe Polynome fournit

- le int "constant" MAX_ETAPE :

```
public static final int MAX_ETAPE
```

- les attributs :

```
private double[] coefficients
 // Les coefficients du polynôme

public int degre
 // le degré du polynôme
```

La classe Polynome

La classe Polynome fournit

- les constructeurs :

```
public Polynome(int degre)
 // Construit le polynôme x^degre
 // Produit une "IllegalArgumentException" si degre < 0

public Polynome()
 // Construit le polynôme nul (0).
```

La classe Polynome

La classe Polynome fournit

- les méthodes :

```
public void setCoefficient(int degre, double coefficient)
 // Affecte la valeur coefficient au
 // coefficient de degré degre.
 // Lance une "IllegalArgumentException" si degre < 0
 // ou si degre est supérieur au degré du polynôme
public double evalueEn(double x)
 // renvoie la valeur du polynôme calculée
 // au point d'abscisse x
```

La classe Polynome

La classe Polynome fournit

- les méthodes :

```
public int getDegre()
 // Renvoie le degré du polynôme

public Polynome polynomeDerive()
 // renvoie le polynôme qui est la dérivée du
 // polynôme courant

public String toString()
 // renvoie renvoie une chaîne de caractères représentant
 // le polynôme courant
```

La classe Polynome

On vous demande d'implémenter les méthodes suivantes

- les méthodes :

```
public double racineParBissection(double a, double b, int d)
                                        throws NumeriqueException
 // renvoie une approximation à minimum d
 // décimales exactes de la racine contenue
 // dans l'intervalle [a ; b] ou [b ; a],
 // obtenue par la méthode de la bissection.
 // Lance une NumeriqueException en cas de problème.
```

La classe Polynome

-

```
public double racineParNewton(double a, double b, int d)
                                        throws NumeriqueException
 // renvoie une approximation à minimum d
 // décimales exactes de la racine contenue
 // dans l'intervalle [a ; b] ou [b ; a],
 // obtenue par la méthode de Newton(-Raphson).
 // Lance une NumeriqueException en cas de problème.
```

racineParBissection(double a, double b, int d) : Pseudo-code

// 1) Vérifier si a $\leq$ b

Si a>b alors interversion des valeurs de a et b

// 2) Vérifier si une des bornes n'est pas la racine

Si $f(\texttt{a}) = 0$ alors racine=a $\rightarrow$ fin de la méthode

Si $f(\texttt{b}) = 0$ alors racine=b $\rightarrow$ fin de la méthode

// 3) Vérifier Bolzano

Si $f(\texttt{a}) \cdot f(\texttt{b}) > 0$ alors NumeriqueException

// 4) Calcul du nombre d'itérations

n = le plus petit entier $n \geq \log_2\left(10^d \cdot (b-a)\right)$

Si n>MAX_ETAPE alors NumeriqueException

// 5) Approximation initiale

racine = (a+b)/2

racineParBissection(double a, double b, int d) : Pseudo-code

```
// 6) Itérations
```
Pour `i` de `1` à `n` on fait

    Si f(`racine`) $= 0$ alors fin de la méthode

    Mise à jour de l'intervalle `[a, b]` en fonction de f(`a`), f(`b`) et f(`racine`)

    Calcul de la nouvelle approximation : `racine = (a+b)/2`

```
// 7) Vérification finale du nombre de décimales exactes
```
Si le nombre de décimales exactes $<$ `d` alors `NumeriqueException`

Sinon On renvoie `racine`

racineParNewton(double a, double b, int d) : Pseudo-code

```
// 1) Vérifier si une des bornes n'est pas la racine
Si f(a) = 0 alors racine=a → fin de la méthode
Si f(b) = 0 alors racine=b → fin de la méthode
// 2) Vérifier Bolzano
Si f(a) · f(b) > 0 alors NumeriqueException
// 3) Vérification des hypothèses supplémentaires
Si f'(a) · f'(b) ≤ 0 alors NumeriqueException
Si f''(a) · f''(b) ≤ 0 alors NumeriqueException
// 4) Approximation initiale et extremite fixe
Si f(a) · f''(a) > 0 alors
    racine = a
    extremiteFixe = b
Sinon
    racine = b
    extremiteFixe = a
```

racineParNewton(double a, double b, int d) : Pseudo-code

```
// 5) Itérations
n=0
```

Tant que `n<MAX_ETAPE et` nombre de décimales exactes $<$ `d`

Nouvelle approximation : `racine = racine-`f`(racine)/`f'`(racine)`

Si f(`racine`) $= 0$ alors fin de la méthode

Si les hypothèses supplémentaires ne sont pas vérifiées en $x =$`racine`

alors `NumeriqueException` (2 cas)

Si instabilités numériques alors `NumeriqueException` (3 cas)

```
// 6) Vérification finale du nombre de décimales exactes
```

Si le nombre de décimales exactes $\geq$ `d` alors on renvoie `racine`

Sinon `NumeriqueException`

Conseils

- Donnez des noms parlants à vos variables
- Évitez de calculer plusieurs fois la même chose
  - → Utilisez une variable afin de pouvoir réutiliser un résultat au lieu de le recalculer !
  - → Regardez bien ce qui peut être calculé une seule fois avant les itérations

Fonctions Java utiles

1) `Math.floor(nombre)` :
   arrondi `nombre` à l'**entier inférieur** le plus proche

   Exemples :

   - `Math.floor(1.25)` $\rightarrow$ 1
   - `Math.floor(254.73)` $\rightarrow$ 254
   - `Math.floor(-32.236)` $\rightarrow$ -33
   - `Math.floor(-1.536)` $\rightarrow$ -2

Fonctions Java utiles

2) `Math.ceil(nombre)` :
arrondi `nombre` à l'**entier supérieur** le plus proche

Exemples :

- `Math.ceil(1.25)` $\rightarrow$ 2
- `Math.ceil(254.73)` $\rightarrow$ 255
- `Math.ceil(-32.236)` $\rightarrow$ -32
- `Math.ceil(-1.536)` $\rightarrow$ -1

Fonctions Java utiles

3) `Math.log(nombre)` :
   logarithme népérien de `nombre` ($\ln($`nombre`$)$)

   Exemples :

   - `Math.log(e)` $\rightarrow 1$
   - `Math.log(10)` $\rightarrow 2.302585092994$

4) `Math.log10(nombre)` :
   logarithme en base 10 de `nombre` ($\log_{10}($`nombre`$)$)

   Exemples :

   - `Math.log10(e)` $\rightarrow 0.43429448190$
   - `Math.log10(10)` $\rightarrow 1$

5) Pour calculer $\log_2(x)$: `Math.log(x)/Math.log(2)`

   Exemples :

   - `Math.log(8)/Math.log(2)` $\rightarrow \log_2(8) = 3$
   - `Math.log(10)/Math.log(2)` $\rightarrow \log_2(10) = 3,3219280948$

Fonctions Java utiles

6) `Math.abs(nombre)` :
valeur absolue de `nombre` ($|$`nombre`$|$)

Exemples :

- `Math.abs(2)` $\rightarrow$ 2
- `Math.abs(-3)` $\rightarrow$ 3