MÉTHODES NUMÉRIQUES

Nicolas CHIREUX

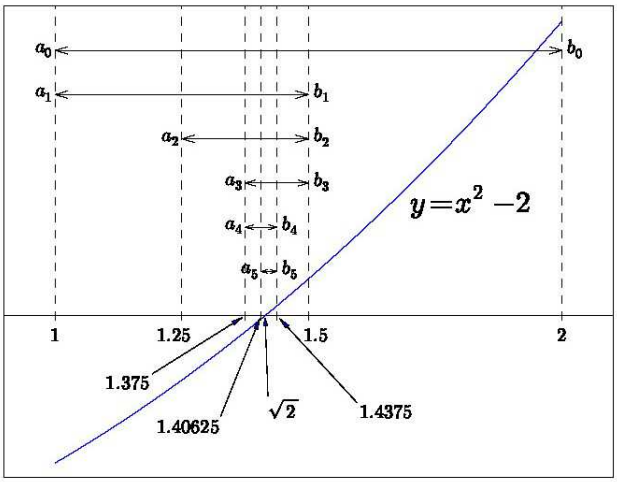
1 Recherche d'un zéro

1.1 Recherche dichotomique d'un zéro

On suppose qu'une fonction continue f prend des valeurs de signe opposé en a et b avec $a < b$. Le théorème des valeurs intermédiaires assure que f s'annule au moins une fois dans cet intervalle.

Notons $m = \frac{a+b}{2}$. Si $f(a)$ et $f(m)$ sont de signe opposé, le TVI nous assure que f s'annule sur $[a, m]$ sinon c'est sur $[m, b]$. Nous réduisons donc l'intervalle de moitié à chaque itération.

En itérant ce procédé, nous construisons une suite de segments $[a_n, b_n]$ contenant un zéro de f de taille divisée par 2 à chaque étape.



Les suites (a_n) et (b_n) sont adjacentes et convergent vers une limite commune l vérifiant $f(l) = 0$. Elles vont nous permettre d'obtenir une approximation d'une zéro de f à ε près.

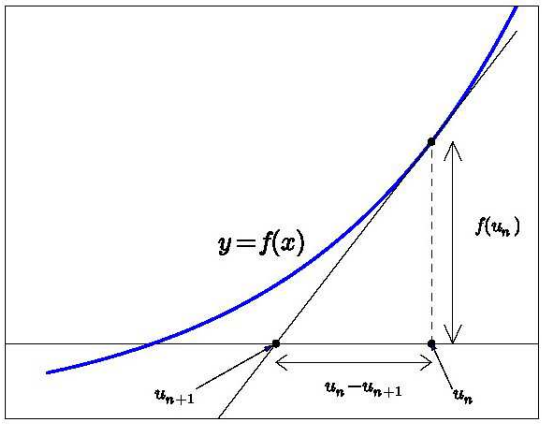
FIGURE 1 – Dichotomie

Data: f,a,b,ε
$g, d \leftarrow a, b$ ♯ bornes gauche et droite du segment;
while $|g - d| > 2\varepsilon$ **do**
  $m \leftarrow \frac{g+d}{2}$;
  if $f(g).f(m) \leq 0$ **then**
    $d \leftarrow m$ ♯ on continue à gauche;
  else
    $g \leftarrow m$ ♯ on continue à droite;
  end
end
Result: $\frac{g+d}{2}$

Algorithm 1: Dichotomie

— Écrire la fonction *dichotomie*(f, a, b, ε).
— Tester la fonction en prenant pour f la fonction sin, $[a, b] = [3, 4]$ pour différentes valeurs de ε.
— Tester la fonction en prenant pour f la fonction $x \to x^2 - 2$, $[a, b] = [1, 2]$ pour différentes valeurs de ε

1.2 Méthode de Newton



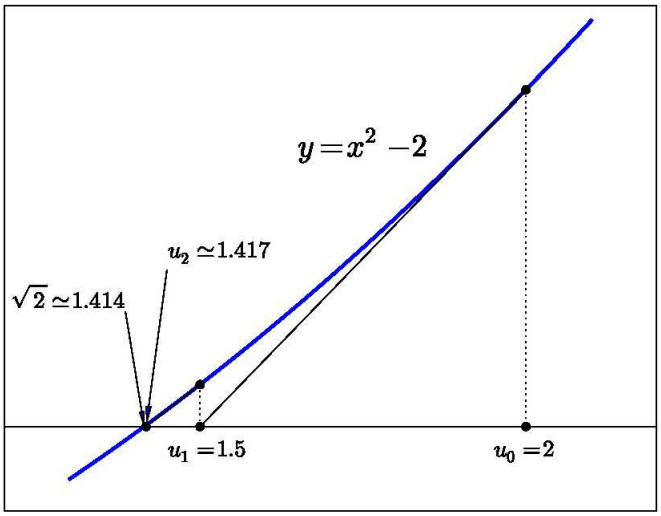
La méthode de Newton consiste à approcher le zéro d'une fonction en partant d'une première valeur u_0 qu'on espère pas trop éloignée d'un zéro et d'itérer le procédé géométrique suivant :

on prend le point du graphe de f d'abscisse u_n, on trace la tangente au graphe - on la supposera non horizontale - et on note u_{n+1} l'abscisse de l'intersection de cette tangente avec l'axe des x.

FIGURE 2 – Méthode de Newton

La pente vaut d'une part $\dfrac{f(u_n)}{u_n - u_{n+1}}$ et d'autre part $f'(u_n)$ alors

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)} \tag{1}$$



Sur l'exemple $x \to x^2 - 2$, voici les trois premiers termes de la suite en prenant $u_0 = 2$. Notez la rapidité de la convergence.

Il faut pour l'algorithme décider d'un test d'arrêt.
Un choix possible est $|u_{n+1} - u_n| \leq \varepsilon$. Cela ne garantit pas que f va posséder un zéro situé à moins de ε de u_{n+1} mais en pratique ça marche plutôt bien.

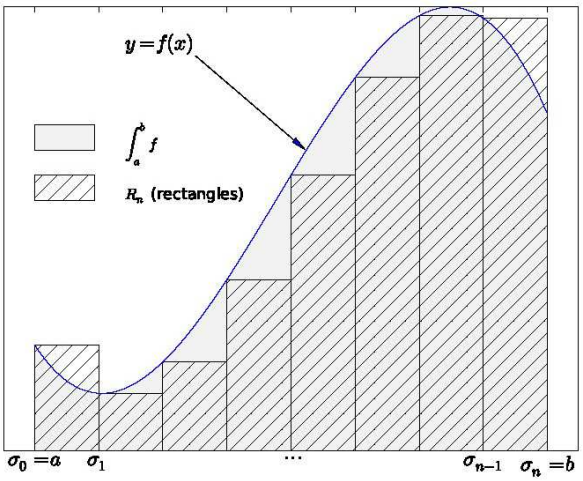
FIGURE 3 – Méthode de Newton

— Écrire la fonction $newton(f, fp, x_0, nb_{iter})$.
— Tester la fonction en prenant pour f la fonction $x \to x^3 - 3x^2 + 1$, $x_0 = 1.5$ pour différentes valeurs de nb_{iter}
— Tester la fonction en prenant pour f la fonction sin, $x_0 = 3$ pour différentes valeurs de nb_{iter}. Comparer avec la dichotomie.

2 Intégration numérique

Les différentes bibliothèques utilisées seront :
— math pour cos, exp, sin, log.
— scipy.integrate pour quad.
— time pour time

2.1 Méthodes des rectangles



Le théorème de Riemann dit que si f est continue (par morceaux) sur un segment $[a,b]$ alors on peut approcher l'aire située sous le graphe de f par la somme des aires des rectangles approchant f en n points uniformément répartis $\sigma_k = a + k\dfrac{b-a}{n}$ où $0 \leq k \leq n-1$:

$$R_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f(\sigma_k) \xrightarrow{n\to+\infty} \int_a^b f(t)dt \qquad (2)$$

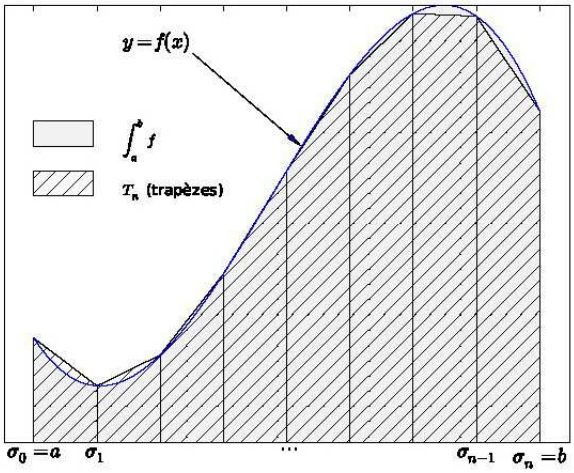
FIGURE 4 – Méthode des rectangles

— Écrire une fonction $def rectangles(f,a,b,n)$ prenant en entrée une fonction , deux bornes et un nombre de pas n, et renvoyant l'approximation R_n de l'intégrale de la fonction.
— Tester la fonction précédente pour évaluer des intégrales connues :

1. $t \to t$ sur $[0,1]$
2. $t \to t^{10}$ sur $[0,1]$
3. $t \to \cos t$ sur $[0,\pi/2]$
4. $t \to \exp t$ sur $[-3,3]$

On commencera par définir en Python les 4 fonctions ; ensuite, on évaluera pour chacune de ces fonctions la différence entre la valeur exacte et la valeur approchée, pour différentes valeurs de n : $n \in \{10, 10^3, 10^5\}$. On pourra enfin comparer au résultat fourni par la fonction **quad** de la bibliothèque **scipy.integrate**.

2.2 Méthodes des trapèzes



On peut aussi décider d'approcher f (puis son intégrale) en l'interpolant sur chaque $[\sigma_k, \sigma_{k+1}]$ par des fonctions affines (plutôt que des constantes), ce qui conduit à considérer l'aire de trapèzes :

$$T_n = \frac{b-a}{2n}\sum_{k=0}^{n-1}(f(\sigma_k)+f(\sigma_{k+1})) \qquad (3)$$

$$= R_n + \frac{b-a}{2n}(f(b)-f(a)) \qquad (4)$$

FIGURE 5 – Méthode des trapèzes

— Écrire une fonction $def\ trapezes(f,a,b,n)$ prenant en entrée une fonction , deux bornes et un nombre de pas n, et renvoyant l'approximation T_n de l'intégrale de la fonction.
— Tester la fonction précédente pour évaluer des intégrales connues :

1. $t \to t$ sur $[0,1]$

2. $t \to t^{10}$ sur $[0,1]$
3. $t \to \cos t$ sur $[0,\pi/2]$
4. $t \to \exp t$ sur $[-3,3]$

On commencera par définir en Python les 4 fonctions ; ensuite, on évaluera pour chacune de ces fonctions la différence entre la valeur exacte et la valeur approchée, pour différentes valeurs de n : $n \in \{10, 10^3, 10^5\}$. On pourra enfin comparer au résultat fourni par la fonction **quad** de la bibliothèque **scipy.integrate**.

2.3 Méthodes de Simpson

On peut enfin décider d'approcher f (puis son intégrale) en l'interpolant à l'aide de σ_k, σ_{k+1} et $m = \dfrac{\sigma_k + \sigma_{k+1}}{2}$ par un polynôme quadratique :

$$P_k(x) = f(\sigma_k)\frac{(x-m)(x-\sigma_{k+1})}{(\sigma_k-\sigma_{k+1})(\sigma_k-m)} + f(m)\frac{(x-\sigma_k)(x-\sigma_{k+1})}{(m-\sigma_k)(m-\sigma_{k+1})} + f(\sigma_{k+1})\frac{(x-m)(x-\sigma_k)}{(\sigma_{k+1}-\sigma_k)(\sigma_{k+1}-m)} \quad (5)$$

Alors

$$P_n = \frac{b-a}{6n}\sum_{k=0}^{n-1}\left(f(\sigma_k) + 4f(\frac{\sigma_k+\sigma_{k+1}}{2}) + f(\sigma_{k+1})\right) \quad (6)$$

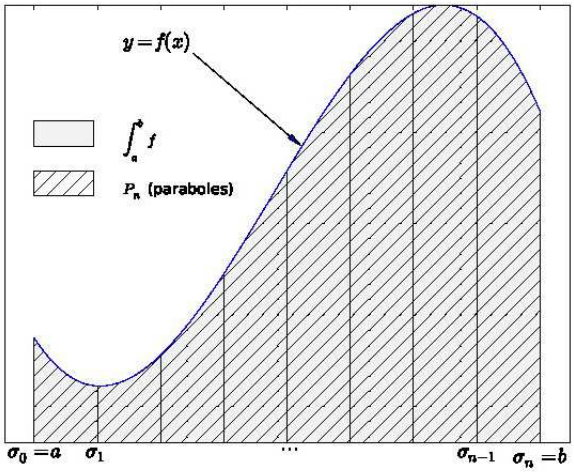


FIGURE 6 – Méthode de Simpson

— Écrire une fonction *def Simpson*(f, a, b, n) prenant en entrée une fonction , deux bornes et un nombre de pas n, et renvoyant l'approximation P_n de l'intégrale de la fonction.
— Tester la fonction précédente pour évaluer des intégrales connues :
1. $t \to t$ sur $[0,1]$
2. $t \to t^{10}$ sur $[0,1]$
3. $t \to \cos t$ sur $[0,\pi/2]$
4. $t \to \exp t$ sur $[-3,3]$

On commencera par définir en Python les 4 fonctions ; ensuite, on évaluera pour chacune de ces fonctions la différence entre la valeur exacte et la valeur approchée, pour différentes valeurs de n : $n \in \{10, 10^3, 10^5\}$. On pourra enfin comparer au résultat fourni par la fonction **quad** de la bibliothèque **scipy.integrate**.
— Donner le nombre d'évaluations de f pour calculer R_n, T_n, P_n.
— Résumer les qualités d'approximation qu'on chiffrera par $R_n - 1$, $T_n - 1$ et $P_n - 1$ et temps de calcul nécessaires à l'évaluation de l'intégrale $\int_0^{\pi/2} \cos t\, dt = 1$ pour les différentes méthodes et différentes valeurs de n : $n \in \{10^2, 10^4, 10^6\}$.
Indication : pour évaluer le temps de calcul placer $t_1 = time()$ avant le calcul et $t_2 = time()$ après le calcul. Le temps de calcul vaut alors $t_2 - t_1$.

3 Résolution d'équation différentielle par la méthode d'Euler

Les différentes bibliothèques utilisées seront :
— numpy pour array, linspace importée sous l'alias np par import numpy as np
— matplotlib.pyplot pour plot, show importée sous l'alias plt par import matplotlib.pyplot as plt
— scipy.optimize pour fsolve

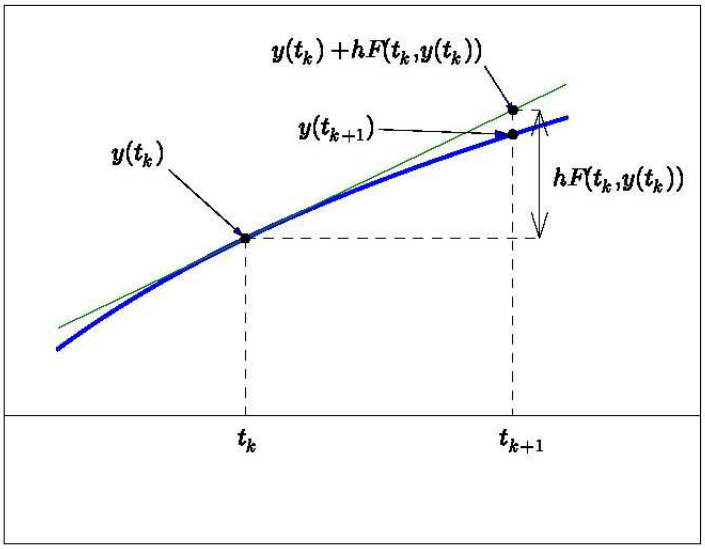
3.1 Méthode d'Euler

Des théorèmes assurent que sous des conditions raisonnables, il existe une unique application y de classe $\mathcal{C}^1$ sur $[a,b]$ dont la valeur est imposée en a et qui vérifie une équation de la forme $y'(t) = F(t, y(t))$ pour tout $t \in [a,b]$.

L'objet des schémas numériques est d'obtenir des approximations de ces solutions dont la théorie donne l'existence de façon non constructive. En pratique, on tente en général d'approcher y en un certain nombre de points répartis sur l'intervalle $[a,b]$.

Plus précisément, on veut calculer une approximation y_k des $y(t_k)$, avec $t_k = a + k.h$ où $h = \dfrac{b-a}{n}$ est un pas qu'il conviendra d'ajuster. De façon très simple, si on écrit :

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(u)\,du = \int_{t_k}^{t_{k+1}} F(u, y(u))\,du \simeq hF(t_k, y(t_k)) \tag{7}$$



alors on obtient la méthode d'Euler : les approximations sont calculées de proche en proche via la formule suivante : $y_{k+1} = y_k + hF(t_k, y_k)$. On initialise bien entendu avec $y_0 = y(a)$.

FIGURE 7 – Méthode d'Euler

— Écrire une fonction $def\ euler(F, t_0, y_0, t_1, n)$ calculant les valeurs approchées d'une solution d'équation différentielle dans le cadre précédent.
La fonction devra renvoyer le tableau des $n+1$ valeurs approchées de y (solution de $y'(t) = F(t, y(t))$ avec $y(t_0) = y0$) aux $n+1$ temps $t_k = t_0 + k.\dfrac{t_1 - t_0}{n}$
Indication :il suffit de calculer les termes de proche en proche. On peut soit créer la liste immédiatement à la bonne taille, soit ajouter chaque nouvel élément en bout de liste via la méthode append.
— Tester la fonction précédente sur $[0,1]$ avec l'équation $y' = y$ et la condition initiale $y(0) = 1$ pour $n = 10$ puis $n = 100$. On notera $y10$ et $y100$ les valeurs approchées de la fonction exponentielle aux temps t_k pour $n = 10$ et $n = 100$.
Tracer sur un même graphe $y10$, $y100$ et $exp(t)$ qui est la solution exacte.
— L'équation du deuxième ordre $y'' = -y$ se ramène à la théorie du premier ordre en considérant

$Y(t) = [y(t), y^{"}(t)]$, qui vérifie $Y^{'}(t) = F(t, Y(t))$. On a

$$\begin{pmatrix} y^{'}(t) \\ y^{"}(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} y(t) \\ y^{'}(t) \end{pmatrix} \tag{8}$$

On a donc $F(t, [a, b] = [b, -a])$. A l'aide de ces considérations, calculer une approximation de la solution de $y^{"} = -y$ sur $[0, 4]$, avec $y(0) = 0$ et $y^{'}(0) = 1$ (la solution de ce problème de Cauchy est la fonction sinus). On découpera l'intervalle en 100, 1000 et 10000.
Tracer sur un même graphe le portrait de phase des solutions obtenues avec les trois différentes découpes de l'intervalle ainsi que celui de la fonction sinus.
Indication : On va passer par des array de numpy pour que l'addition des objets se passe comme dans $\mathbb{R}^2$ lors du calcul de $y_k + h * F(t_0 + k * h, y_k)$. On commencera par définir la fonction F.

3.2 Méthode d'Euler implicite ou rétrograde

Le schéma d'Euler implicite, ou rétrograde consiste à remplacer la relation $y_{k+1} = y_k + hF(t_k, y_k)$ par $y_{k+1} = y_k + hF(t_{k+1}, y_{k+1})$.

Bien entendu, cette relation ne donne pas explicitement y_{k+1} en fonction de y_k (d'où son nom) et nécessite donc à chaque étape une résolution d'équation de la forme $\Phi(y_{k+1}) = 0$. Puisque y_{k+1} est censé être proche de y_k, on dispose d'une bonne première approximation.

— Programmer la méthode d'Euler implicite.
Indication : pour demander à Python une (approximation d'une) solution de l'équation implicite $y_{k+1} = y_k + hF(t_{k+1}, y_{k+1})$, on pourra exécuter : fsolve(lambda $y : y - y_k - h * F(t_{k+1}, y), y)$, avec fsolve issue de la bibliothèque scipy.optimize
— Tester cette méthode sur l'équation $y^{"} = -y$ sur $[0, 3]$, avec la condition initiale $y(0) = 1$. Tester des pas entre $0, 1$ et 2.
— Comparer sur ce même exemple avec la méthode d'Euler usuelle (explicite)