Résolution numérique d'équations

1 Présentation du problème

Il est fréquent, en sciences, que l'on ait à trouver le ou les réel(s) x_0 vérifiant $f(x_0) = 0$ où f est une fonction quelconque [1]

En mécanique, par exemple, dans un problème à un seul degré de liberté, une position d'équilibre du système correspond à une annulation de la dérivée de l'énergie potentielle par rapport à la variable de position. En chimie, l'équilibre d'une réaction peut être déterminé par la valeur de l'avancement solution d'une équation faisant intervenir la constante d'équilibre de la réaction. En mathématiques, on peut chercher des candidats pour la limite d'une suite $u_{n+1} = g(u_n)$ en résolvant l'équation $g(x) = x$ soit $g(x) - x = 0$. Les exemples sont donc très nombreux.

Parfois, ces équations peuvent être résolues explicitement (dans le cas d'un polynôme de faible degré, par exemple). Mais bien souvent, il n'existe pas de méthode mathématique pour déterminer les solutions de l'équation, et on en est réduit, comme pour le problème de l'intégration, à utiliser des méthodes numériques.

Dans ce chapitre, nous examinerons donc différentes méthodes capable de trouver un x_0 qui vérifierait $f(x_0) = 0$. Nous nous restreindrons au cas de fonctions f réelles, définies sur un intervale de $\mathbb{R}$ et à valeurs dans $\mathbb{R}$. D'autres hypothèses de continuité ou de dérivabilité seront faites sur la fonction f, en fonction des besoins. Il est bien évident que pour certaines fonctions, il n'existe aucun algorithme capable d'identifier x_0, comme pour la fonction

$$f \quad \begin{cases} \mathbb{R} \mapsto \mathbb{R} \\ \\ \begin{cases} \sqrt{5} & \mapsto 0 \\ \\ x \neq \sqrt{5} & \mapsto 1 \end{cases} \end{cases}$$

Ou à tout le moins un flottant aussi proche que possible de x_0, car rien ne dit que x_0 est représentable par un flottant. Ni d'ailleurs que le calcul de $f(x_0)$, même si x_0 est représentable par un flottant, soit égal à 0, compte tenu des imprécisions liées au calcul numérique flottant. Ou inversement, si le calcul numérique de $f(x)$ donne zéro, rien ne dit que $f(x)$ soit réellement rigoureusement nul !

1. Cela couvre tous les problèmes de résolution d'une équation ne faisant intervenir qu'une seule variable, car si d'aventure aucun des deux membres de l'équation ne fait apparaître de zéro, on peut s'y ramener simplement en passant tout du même côté du signe égal !

2 Recherche par dichotomie

2.1 Principe

Supposons dans un premier temps que f est définie et continue sur un intervale $[a, b]$, et que $f(a) \geqslant 0$ et $f(b) \leqslant 0$, ou bien que $f(a) \leqslant 0$ et $f(b) \geqslant 0$ (ces deux cas pouvant être résumés, de façon équivalente, par l'expression $f(a) \times f(b) \leqslant 0$). Le théorème des valeurs intermédiaires nous affirme qu'il existe un $c \in [a, b]$ (pas nécessairement unique) vérifiant $f(c) = 0$.

Pour s'approcher de ce c, nous allons procéder par dichotomie, de façon assez similaire à la recherche d'un élément dans une liste triée que nous avons déjà étudiée.

On s'intéresse donc à ce qui se passe au milieu de l'intervale considéré, soit à $f\left(\frac{a+b}{2}\right)$.

Puisque $f(a) \times f(b) \leqslant 0$, on a nécessairement au moins l'une des propriétés suivantes (les deux pouvant être vérifiés, notamment dans le cas où $f\left(\frac{a+b}{2}\right) = 0$) :

- $f(a) \times f\left(\frac{a+b}{2}\right) \leqslant 0$
- $f\left(\frac{a+b}{2}\right) \times f(b) \leqslant 0$.

Dans le premier cas, cela signifie qu'il existe un $c \in \left[a, \frac{a+b}{2}\right]$ vérifiant $f(c) = 0$.

Dans le second cas, on pourra trouver un tel c dans $\left[\frac{a+b}{2}, b\right]$

Puis on recommence en examinant la valeur de la fonction au milieu de ce nouvel intervale. À chaque étape, on divise ainsi par deux la taille de l'intervale dont on sait qu'il contient une racine. Pour obtenir une approximation à ε près d'une racine, il suffit de réduire cet intervale à une taille inférieure à 2ε.

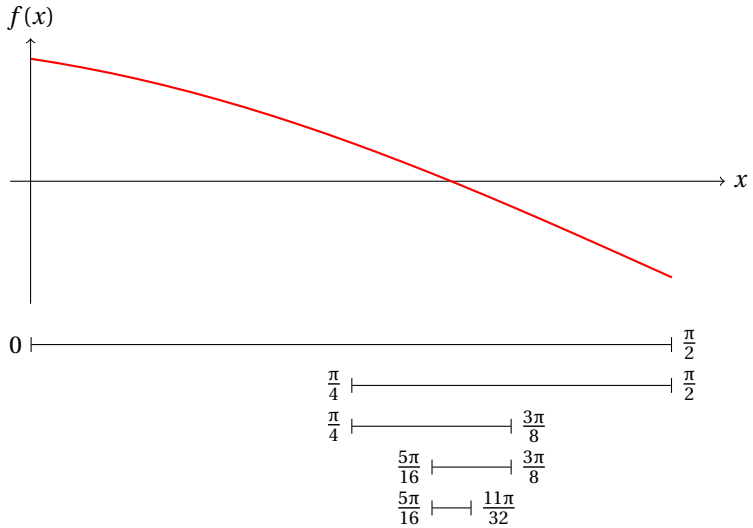
2.2 Exemple d'utilisation

Illustrons cet algorithme sur un exemple. Considérons la fonction

$$f \quad \begin{cases} \mathbb{R} \mapsto \mathbb{R} \\ \\ x \mapsto \cos(x) - \dfrac{x}{2} \end{cases}$$

C'est une fonction continue sur $\mathbb{R}$, et on a $f(0) = 1$ et $f(\pi/2) = -\pi/4$. Ce qui implique qu'il existe un $x_0 \in]0, \pi/2[$ vérifiant $f(x_0) = 0$. Une telle équation ne peut pas être résolue analytiquement, et on doit donc se résoudre à utiliser des méthodes de résolution numérique.

La courbe f, et les quatres premières étapes de l'algorithme de recherche de racine par dichotomie (en partant de l'intervale $]0, \pi/2[$) sont représentées ci-dessous :



2.3 Implémentation

L'algorithme de recherche de racine par dichotomie pourra s'écrire ainsi :

```
def Dichotomie(f, a, b, eps) :
    fa, fb = f(a), f(b)

    # On vérifie les conditions initiales
    if fa * fb > 0 :
        raise ValueError("f(a) et f(b) ne sont pas de signes opposés.")

    # Tant que la largeur de l'intervale excède 2 epsilon
    while (b-a) > 2*eps :
        m = (a+b)/2
        fm = f(m)              # On calcule f( (a+b)/2 )

        if fa * fm <= 0 :
            b, fb = m, fm      # On poursuit avec la moitié gauche
        else :
            a, fa = m, fm      # On poursuit avec la moitié droite

    return (a+b)/2             # On retourne le milieu de l'intervale
```

Cet algorithme permet ainsi d'obtenir une approximation, par exemple à à 10^{-12} près, de la racine (après avoir défini la fonction f avec l'expression $f(x) = \cos(x) - x/2$),

```
In []: Dichotomie(f, 0, pi/2, 1e-12)
Out[]: 1.0298665293225906
```

Dans l'algorithme précédent, on peut choisir pour invariant de boucle la propriété `f(a)*f(b) <= 0`. En effet, l'algorithme garantit [2] que cette propriété reste vraie à chaque itération de la boucle **`while`**.

Alternativement, on peut utiliser l'affirmation « il y a au moins une racine dans l'intervale $[a, b]$ », qui là aussi reste vraie à chaque itération de la boucle.

On remarquera par ailleurs que la largeur de l'intervale est divisée en deux à chaque itération de la boucle **`while`**, il est donc logique d'en déduire qu'il finira par être plus petit que la grandeur `2*eps`, sous réserve que l'on ait choisi pour `eps` une valeur strictement positive.

La fonction retourne le milieu de l'intervale $[a, b]$, dont la largeur est inférieure à `2*eps`, donc un point à une distance d'au plus `eps` de n'importe quel point de cet intervale $[a, b]$. Puisque celui-ci contient au moins une racine c, le résultat obtenu est distant d'au plus `eps` d'une racine.

2.4 Problèmes d'implémentation dûs aux flottants

L'algorithme précédent est parfaitement correct d'un point de vue mathématique, mais il peut dans de très rares cas poser des problèmes, à cause de l'utilisation des nombres flottants. En général, on ne s'en préoccupera pas, mais c'est l'occasion de montrer comment des choses qui semblent naturelles d'un point de vue mathématique peuvent poser des problèmes une fois converties en algorithme.

Tout d'abord, si l'on demande un très petit ε [3], `a` et `b` pourraient être tellement proches que $\frac{a+b}{2}$ puisse être égal à a ou b !

```
In []: (0.9999999999999999 + 1.0)/2.0
Out[]: 1.0

In []: (1.0 + 1.0000000000000001)/2.0
Out[]: 1.0
```

Ce n'est en soit pas bien grave si l'on ne prend pas le milieu de l'intervale, mais il convient de garantir que sa taille décroit strictement à chaque itération, sinon on ne parviendra jamais à avoir un intervale plus petit que 2ε. On n'a pas cette garantie avec l'algorithme précédemment proposé.

2. Ou semble garantir, nous y reviendrons juste après.
3. On ne devrait en principe pas demander un ε plus petit que $x_0 \times 10^-15$ environ, mais l'utilisateur pourrait le faire par erreur, surtout si x_0 est grand...

Un autre problème peut éventuellement survenir : il est possible d'avoir $f(a) > 0$ et $f(b) > 0$ et pourtant $f(a) \times f(b) \leqslant 0$ (si $f(a)$ et $f(b)$ sont suffisamment petits) !

```
In []: exp(-550)
Out[]: 1.374152566130957e-239

In []: exp(-500)
Out[]: 7.124576406741286e-218

In []: exp(-550)*exp(-500) <= 0.0
Out[]: True
```

On peut donc vouloir comparer les signes plus explicitement que par le signe du produit. Une version plus « sûre » de l'algorithme [4] serait donc :

```
def SignesOpposes(y1, y2) :
    return (y1 <= 0.0 and y2 >= 0.0) or (y1 >= 0.0 and y2 <= 0.0)

def Dichotomie(f, a, b, eps) :
    largeur, fa, fb = b-a, f(a), f(b)

    if eps <= 0.0 :
        raise ValueError("epsilon doit être strictement positif")

    if not SignesOpposes(fa, fb) :
        raise ValueError("f(a) et f(b) ne sont pas de signes opposés.")

    # On travaille avec la largeur de l'intervale
    # pour garantir sa décroissance
    while largeur > 2*eps :
        largeur = largeur / 2.0

        m = a + largeur
        fm = f(m)

        if not SignesOpposes(fa, fm) :
            a, fa = m, fm

    return a + largeur / 2.0
```

4. La première version est amplement suffisante dans le cadre des concours

2.5 Vitesse de convergence

Par construction, l'algorithme précédent convergera nécessairement vers une solution, quelle que soit la fonction f tant qu'elle est continue sur l'intervale considéré. Une préoccupation supplémentaire, cependant, peut être de savoir si cette convergence est rapide ou non. Dans bien des situations pratiques, on a besoin d'obtenir une solution aussi précise que possible, mais on a également besoin de l'obtenir rapidement !

Combien d'itérations sont ici nécessaires pour obtenir le résultat ? On constate aisément qu'à chaque étape de l'algorithme, la taille de l'intervale de recherche est divisée par deux. Ainsi, si l'intervale initial est $[a, b]$, après n étapes, une racine au moins se trouve dans un intervale de taille $(b-a)/2^n$.

La condition d'arrêt s'écrit donc $(b-a)/2^n \leqslant 2\varepsilon$, soit

$$n \geqslant \frac{\log\left(\frac{b-a}{\varepsilon}\right)}{\log(2)} - 1$$

L'algorithme effectuera donc $\lceil \log_2\left(\frac{b-a}{\varepsilon}\right) \rceil - 1$ itérations.

2.6 Avantages et limitations

Le principal avantage de cette méthode est qu'elle garantit que l'on aura systématiquement un résultat, quelle que soit la fonction f, du moment que les hypothèses soient vérifiées.

En revanche, cette méthode converge (relativement) lentement, par rapport à d'autres méthodes que nous allons présenter dans la suite. En effet, la division par deux de l'intervalle fait que l'algorithme fournit, en gros, un bit du résultat à chaque itération [5]. On obtient donc une décimale à peu près toutes les trois itérations [6].

Par ailleurs, cette méthode suppose que l'on connaisse un encadrement du résultat recherché, mais également que la fonction f change de signe au voisinage de ce résultat. En effet, il n'est par exemple pas possible de trouver une solution à l'équation $(x-1)^2 = 0$ avec la méthode dichotomique présentée ici, car elle est positive de chaque côté de la racine $x_0 = 1$.

2.7 Utilisation de `scipy`

Bien évidemment, la méthode de recherche de racine par dichotomie est disponible dans l'un des modules de Python, le module `scipy.optimize`, qui regroupe notamment

5. les premières itérations peuvant ne fournir que des 0 non significatifs, cependant
6. Si les bornes initiales de l'intervale a et b sont du même ordre de grandeur, cela signifie qu'on obtiendra la précision maximale pour un flottant en une cinquantaine d'itérations, mais cela peut être bien plus, par exemple si a est nul et la solution très petite devant b

diverses fonctions de recherche de racine ou de minimum. La fonction de recherche de racine par dichotomie s'appelle `bisect`. On l'importera avec la commande

```
In []: from scipy.optimize import bisect
```

La fonction requiert trois paramètres, la fonction f dont on cherche une racine, et les bornes a et b de l'intervale dans lequel on recherche la racine, qui doivent vérifier la condition $f(a) \times f(b) \leqslant 0$:

```
In []: bisect(f, 0, pi/2, 1e-12)
Out[]: 1.0298665293225906

In []: bisect(f, 0, pi/4)
ValueError: f(a) and f(b) must have different signs
```

La fonction permet également de choisir sa précision, soit une précision absolue (l'erreur commise lors de l'estimation de la racine est inférieure à `xtol`), similaire au paramètre `eps` de notre fonction, soit une précision *relative* (l'erreur commise lors de l'estimation de la racine est inférieure à `rtol` multiplié par la racine) :

```
In []: bisect(f, 0, pi/2, xtol = 1e-4)
Out[]: 1.0297804776674795

In []: bisect(f, 0, pi/2, rtol = 1e-4)
Out[]: 1.0297804776674795
```

Comme la solution est proche de 1, ici, les deux jouent un rôle similaire, mais ce n'est pas toujours le cas, comme lorsque l'on cherche la solution positive de l'équation du second degré $x^2 - 2 \times 10^{10} = 0$ (qui vaut bien évidemment $\sqrt{2} \times 10^5$) :

```
In []: def f(x) :
  ...:     return x**2 - 2e10

In []: bisect(f, 0, 2e5, rtol = 1e-4)
Out[]: 141418.45703125

In []: bisect(f, 0, 2e5, xtol = 1e-4)
Out[]: 141421.35614529252

In []: sqrt(2e10)
Out[]: 141421.35623730952
```

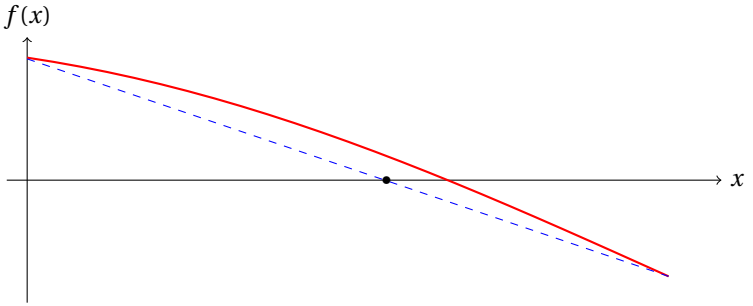
On voit ici que la première méthode donne essentiellement quatre chiffres significatifs (précision relative) tandis que la seconde donne des chiffres corrects jusqu'au quatrième chiffre après la virgule.

3 Méthode de la « fausse position »

3.1 Principe

On peut se demander si découper chaque intervalle en deux intervales de même largeur est la solution la plus intéressante. Après tout, si $|f(a)|$ est beaucoup plus grande que $|f(b)|$, on peut s'attendre à ce que, dans la majorité des situations, la solution se trouve plus proche de b que de a.

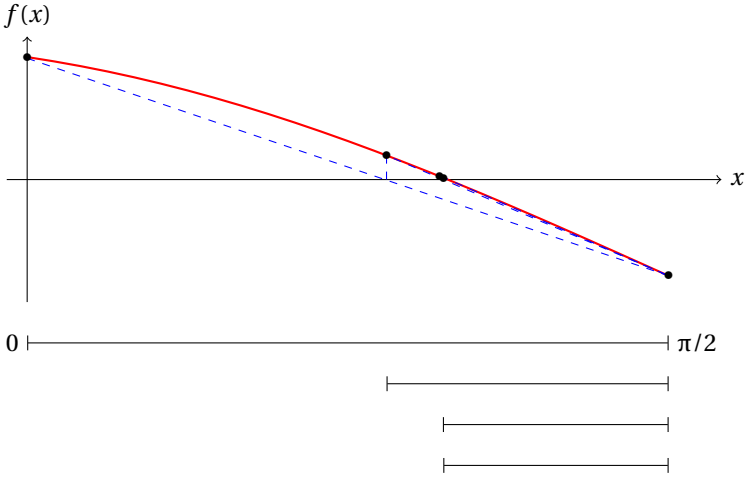
La méthode de la fausse position (ou méthode *regula falsi*) propose de couper l'intervale au niveau de l'intersection entre l'axe des abscisses et la corde reliant les points de coordonnées $(a, f(a))$ et $(b, f(b))$, marquée d'un point sur l'exemple ci-dessous.



L'équation de la corde est $y(x) = f(a) + \dfrac{f(b) - f(a)}{b - a} \times (x - a)$.

On coupe cet intervale à l'abscisse x_m où $y(x)$ s'annule, soit $x_m = \dfrac{a \times f(b) - b \times f(a)}{f(b) - f(a)}$.

On pourra ensuite conserver, comme précédemment, la partie de l'intervale sur laquelle la fonction f dont on cherche une racine, et reitérer l'opération sur l'intervale ainsi réduit :



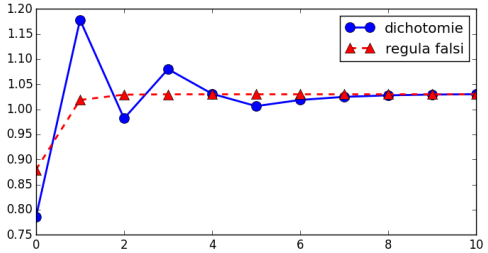
3.2 Implémentation

Seul changement par rapport à la dichotomie, le point où l'on coupe l'intervale :

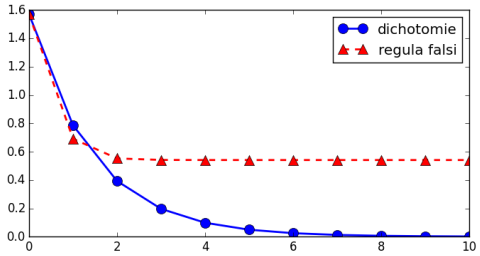
```
def RegulaFalsi(f, a, b, eps) :
    # Vérification des arguments
    if eps <= 0 :
        raise ValueError("epsilon doit être strictement positif")

    if not SignesOpposes(fa, fb) :
        raise ValueError("f(a) et f(b) ne sont pas de signes opposés.")

    # Recherche de la racine
    fa, fb = f(a), f(b)

    while (b-a) > 2*eps :
        # Point pour le "découpage"
        m = (a*fb-b*fa)/(fb-fa)
        fm = f(m)

        # Réduction de l'intervale
        if SignesOpposes(fa, fm) :
            b, fb = m, fm
        else :
            a, fa = m, fm

    return (a*fb-b*fa)/(fb-fa)
```

3.3 Convergence

Si l'on s'intéresse à l'estimation fournie par chacune des deux méthodes à chaque étape (le milieu des intervalles dans le cas de la méthode dichotomique, l'intersection de la corde et de l'axe des abscisses pour la méthode de la fausse position), on peut constater que, sur notre exemple, la méthode de la fausse position converge plus rapidement :



En revanche, un problème apparaît : la largeur de l'intervale de recherche ne tend plus toujours vers 0, comme le montre ce graphique comparant l'évolution de la largeur de l'intervale à chaque étape pour chacune des deux méthodes dans le cas de notre exemple :



Cela peut s'expliquer par le fait que, souvent, l'une des deux bornes de l'intervale « stagne », la racine finissant par tomber systématiquement du même côté de la subdivision.

En effet, si la fonction est concave sur $[a,b]$, alors la corde se situe toujours sous la courbe de la fonction, et la valeur de la fonction au niveau de l'intersection entre la corde et l'axe des abscisses sera donc toujours positive. La borne de l'intervalle où f est négative n'évolue donc plus. Inversement, si f est convexe, la corde se trouve au-dessus de la courbe de la fonction, c'est l'autre extrémité qui n'évolue plus.

Dans notre cas, la fonction est concave sur $[0,\pi/2]$, et par conséquent la borne droite de l'intervale qui ne change jamais.

La condition d'arrêt sur la largeur de l'intervale ne convient donc plus (l'algorithme proposé précédemment ne termine jamais dans le cas de notre exemple !), et il nous en faut une autre. Plusieurs critères sont envisageables :

- limiter le nombre d'itérations de l'algorithme ;
- arrêter l'algorithme lorsque $|f(x_m)|$ est très faible ;
- arrêter l'algorithme lorsque $f(x_m-\varepsilon)$ et $f(x_m+\varepsilon)$ sont de signes différents, ce qui garantit l'existence d'une racine dans un voisinage de taille ε de x_m ;
- arrêter l'algorithme lorsque deux valeurs consécutives de x_m sont très proches.

Aucun de ces critères n'est cependant idéal :

- il est délicat de choisir un nombre d'itérations maximal (s'il est trop faible, on ne sera pas assez près de la racine, et s'il est trop grand, on fera des calculs inutiles) ;
- que $|f(x_m)|$ soit faible ne garantit pas la proximité d'une racine ;
- rien ne dit que $f(x_m-\varepsilon)$ et $f(x_m+\varepsilon)$ seront de signes différents lorsque l'on sera à proximité de la racine ;
- enfin, l'évolution de x_m peut être très lente, et la faible variation de x_m entre une étape et la suivante ne garantit pas que l'on est très proche de la racine.

Souvent, on utilise une combinaison de ces conditions d'arrêt. Par exemple, on choisit ici deux de ces critères : on s'arrêtera lorsque la différence (en valeur absolue) entre deux valeurs successives de l'abscisse où l'on coupe l'intervale sera inférieure à une limite eps,

fournie en paramètre [7], ou bien lorsque l'on aura effectué `maxiter` itérations (afin d'arrêter l'algorithme s'il ne converge pas) :

```
def RegulaFalsi(f, a, b, eps, maxiter) :
    fa, fb, m = f(a), f(b), a

    if eps <= 0 :
        raise ValueError("epsilon doit être strictement positif")

    if not SignesOpposes(fa, fb) :
        raise ValueError("f(a) et f(b) ne sont pas de signes opposés.")

    for _ in range(maxiter) :
        # Point pour le "découpage"
        prec_m, m = m, (a*fb-b*fa)/(fb-fa)
        fm = f(m)

        # Réduction de l'intervale
        if SignesOpposes(fa, fm) :
            b, fb = m, fm
        else :
            a, fa = m, fm

        # Arrêt car on coupe toujours quasiment au même endroit
        if abs(m-prec_m) < eps :
            return (a*fb-b*fa)/(fb-fa)

    # Arrêt car on a atteint le nombre maximal d'itérations
    return (a*fb-b*fa)/(fb-fa)
```

Attention cependant. Bien que l'algorithme renvoie toujours un résultat, rien ne dit que ce résultat soit proche d'une racine de la fonction f. Il peut également s'arrêter car la convergence est trop lente, et que l'on atteint le nombre maximal d'itérations, ou que l'abscisse où s'effectue la découpe n'évolue pas assez rapidement.

En particulier, le `eps` fourni n'est plus ici un majorant de la distance à laquelle se trouve une des racines de f, mais un simple critère d'arrêt.

La méthode de la fausse position n'étant pas au programme, nous ne nous étendrons pas davantage sur les propriétés de sa convergence, mais elle nous sera utile pour bien comprendre la méthode suivante.

7. Le nom `prec_m` mémorisera la valeur précédente de `m` ; on l'initialise à `a` pour permettre la première comparaison.

4 Méthode de la sécante

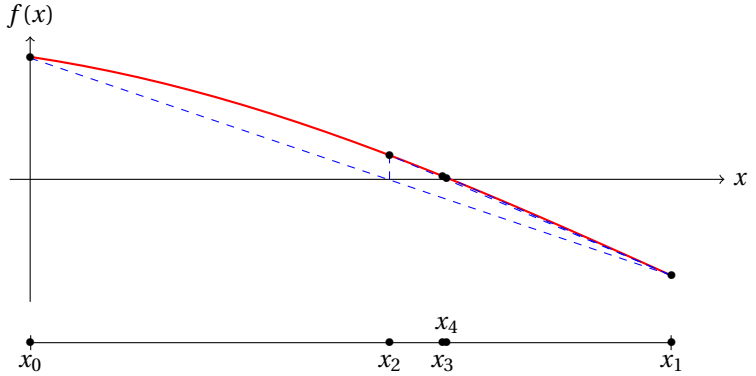
4.1 Principe

Ce que l'on peut retenir de la méthode précédente, c'est qu'elle nous fournit une suite $(x_n)_{n\in\mathbb{N}}$ de valeurs qui converge vers une racine de la fonction f.

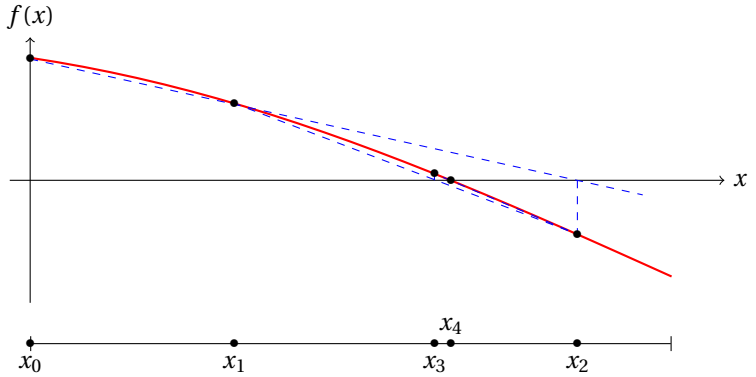
Il en est de même pour la méthode de la sécante. On suppose disposer au début de deux valeurs distinctes x_0 et x_1 si possible proches de la racine recherchées. Puis on définit la suite

$$x_{n+1} = \frac{x_n \times f(x_{n-1}) - x_{n-1} \times f(x_n)}{f(x_{n-1}) - f(x_n)}$$

Il s'agit de la même expression que dans le cas de la méthode de la fausse position : la nouvelle abscisse x_{n+1} correspond à l'intersection de la corde reliant les points $\big(x_{n-1}, f(x_{n-1})\big)$ et $\big(x_n, f(x_n)\big)$.



Notons qu'il n'est pas indispensable que x_0 et x_1 se trouvent de part et d'autre de la racine, comme en témoigne l'exemple suivant :



4.2 Implémentation

L'implémentation de cette méthode est simple. `xp` et `xn` désignent les deux dernières valeurs de la suite [8] (initialement les deux abscisses x_0 et x_1 fournies par l'utilisateur), et on utilise la relation de récurrence pour calculer, itérativement, les valeurs de (x_n).

Il reste le problème de la condition d'arrêt. Comme précédemment, on s'arrêtera lorsque l'incrément $|x_{n+1} - x_n|$ sera inférieur à une limite `eps`, fournie en paramètre, ou bien lorsque l'on aura effectué `maxiter` itérations (afin d'arrêter l'algorithme s'il ne converge pas) :

```
def Secante(f, xp, xn, eps, maxiter) :
    if eps <= 0 :
        raise ValueError("epsilon doit être strictement positif")

    if xn == xp :
        raise ValueError("x0 et x1 doivent être distincts")

    fp, fn = f(xp), f(xn)

    for _ in range(maxiter) :
        # Détermination du terme suivant dans la suite
        xp, fp, xn = xn, fn, (xp*fn-xn*fp)/(fn-fp)
        fn = f(xn)

        # Arrêt car la suite n'évolue plus
        if abs(xn-xp) < eps :
            return xn

    # Arrêt car on a atteint le nombre maximal d'itérations
    return xn
```

Cette fonction trouve rapidement [9] une approximation de la racine de notre problème :

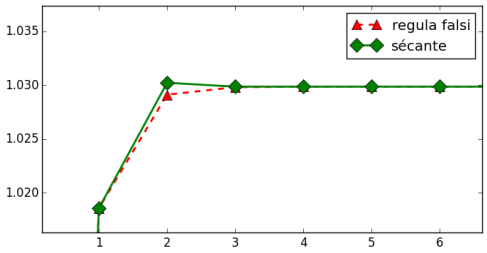
```
In []: Secante(f, 0, pi/2, 1e-12, 100)
Out[]: 1.0298665293222589

In []: Secante(f, 0, 0.5, 1e-12, 100)
Out[]: 1.0298665293222589
```

8. Plus précisément, lorsque l'on entre dans la boucle, `xp` désigne x_{n-1} tandis que `xn` correspond à x_n. `fp` et `fn` conservent les valeurs de $f(x_{n-1})$ et $f(x_n)$. À la sortie de la boucle, `xp` désigne x_n et `xn`, x_{n+1}, et `fp` et `fn` ont été mis à jour pour correspondre à $f(x_n)$ et $f(x_{n+1})$.

9. Une étude plus approfondie montre que la convergence est obtenue en sept itérations pour chacun des deux tests, soit nettement moins que les quarante-et-une étapes nécessaires à la dichotome pour garantir une approximation à 1×10^{-12} près ; tout en gardant à l'esprit que si c'est le cas ici, le résultat obtenu par la méthode de la sécante ne dit pas avec certitude à quel point on s'est approché d'une racine.

4.3 Convergence

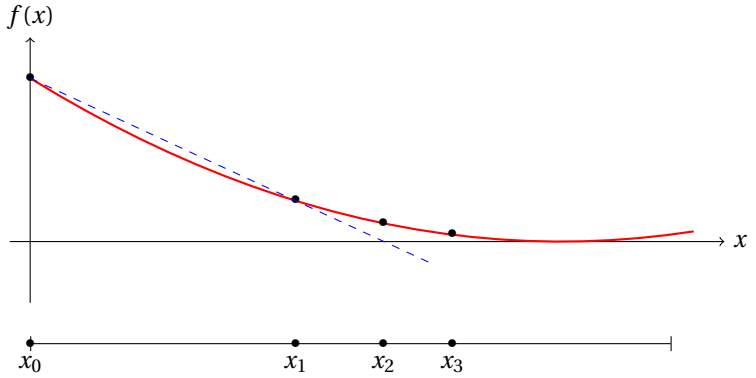
Nous le verrons, lorsque tout se passe bien, elle converge plus rapidement que la méthode dichotomique (et un peu plus vite que la méthode de la fausse position).



Par ailleurs, la méthode de la sécante ne nécessite pas d'avoir un encadrement d'une racine sous la forme de deux valeurs de x pour lesquels les signes de $f(x)$ sont différents. Cela permet de rechercher des racines de fonctions qui ne changent pas de signe, chose impossible avec les deux premières méthodes.

Par exemple, on peut, avec la méthode de la sécante, trouver [10] une approximation [11] de la racine de la fonction $x \mapsto f(x) = (x-2)^2$:

```
In []: def g(x) :
     :     return (x-2)**2

In []: Secante(g, 0, 1, 1e-10, 100)
Out[]: 1.9999999998410967
```



En revanche, puisque l'on ne dispose plus d'encadrements, il n'est pas certain que la suite $(x_n)_{n\in\mathbb{N}}$ converge ! En particulier, x_{n+1} peut ne pas être dans l'intervale de définition

10. On notera que la convergence est moins rapide lorsque la courbe est tangeante avec l'axe des abscisses, comme sur ce dernier exemple.

11. Remarquons au passage que l'erreur sur le résultat obtenu est supérieure ici au `eps` fournie en paramètre.

de la fonction f, et si $f(x_{n-1}) = f(x_n)$, il n'est même pas possible de déterminer x_{n+1} (division par zéro).

5 Méthode de Newton-Raphson

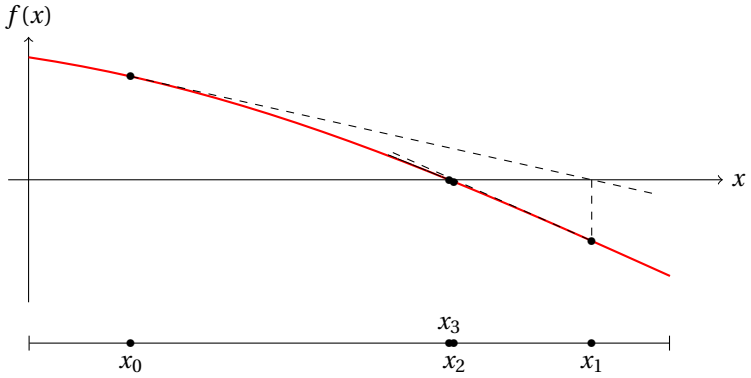
5.1 Présentation

Dans la méthode de la sécante, plus on s'approche de la racine, plus la corde permettant d'obtenir l'abscisse x_n suivante de la suite se rapproche de la tangente de la courbe.

En fait, si f est dérivable, et que l'on connaît sa dérivée, la méthode de Newton propose justement de chercher une racine de f en utilisant une suite $(x_n)_{n\in\mathbb{N}}$ où x_{n+1} est déterminé en calculant l'intersection entre l'axe des abscisse et la tangente à la courbe au point $(x_n, f(x_n))$, dont l'équation est $f'(x_n)\times(x-x_n) + f(x_n)$. On a donc la relation de récurrence suivante :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

En dehors de cette relation de récurrence un peu différente (qui nécessite la connaissance de f' mais qui, en contrepartie, n'a pas besoin de x_{n-1} pour déterminer x_{n+1}), tout se passe comme pour la méthode de la sécante, et on peut utiliser les mêmes conditions d'arrêt, avec les mêmes limitations que précédemment.



Avant même que la méthode soit connue, elle a été utilisée pour le calcul des racines carrées. En effet, Héron d'Alexandrie avait proposé, pour calculer la racine d'un réel a, d'étudier la limite de la suite

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

ce qui se trouve être exactement la suite fournie par la méthode de Newton-Raphson pour l'équation $x^2 - a = 0$!

5.2 Implémentation

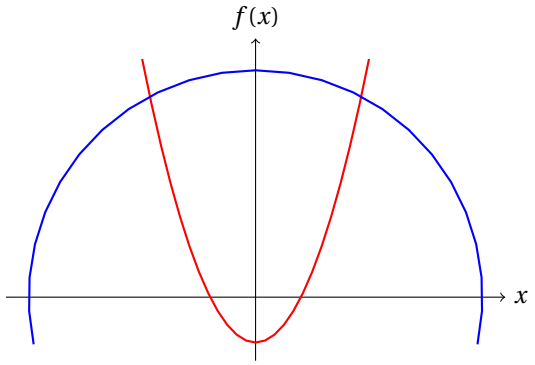
L'implémentation de la méthode de Newton-Raphson est des plus simple :

```
def Newton(f, fprime, x, eps, maxiter) :
    if eps <= 0 :
        raise ValueError("epsilon doit être strictement positif")

    for _ in range(maxiter) :
        # Détermination du terme suivant dans la suite
        nx = x - f(x) / fprime(x)

        # Arrêt car la suite n'évolue plus
        if abs(nx-x) < eps :
            return nx

        # Mise à jour de x
        x = nx

    # Arrêt car on a atteint le nombre maximal d'itérations
    return x
```

5.3 Convergence

Nous verrons que la méthode de Newton-Raphson offre la convergence la plus rapide de toutes les méthodes étudiées dans ce chapitre. Toutefois, elle est parfois très sensible au choix de la première estimation, x_0.

Prenons un exemple concret : on souhaite déterminer les points d'intersection entre le cercle de rayon 5 centré sur l'origine et la parabole d'équation $f : x \mapsto g(x) = x^2 - 1$:

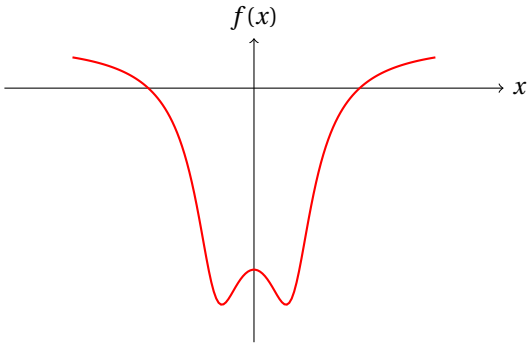


Pour trouver les abscisses des intersections, on cherche les x vérifiant $x^2 + g(x)^2 = 5^2$, ce

qui conduit à chercher les racines de la fonction :

$$f : x \mapsto f(x) = 1 - \frac{5}{\sqrt{x^2 + (x^2-1)^2}}$$

En voici une représentation graphique :



On peut aisément, pour les besoins de l'algorithme, calculer sa dérivée :

$$f' : x \mapsto f'(x) = \frac{5\,(2x^2-1)\,x}{(x^4 - x^2 + 1)^{3/2}}$$

Il se trouve qu'il est ici possible de déterminer analytiquement les racines de ce problème, puisque l'on a une équation bicarrée, qui peut être résolue avec le changement de variable $X = x^2$„ ce qui conduit [12] aux solutions

$$\pm\sqrt{0.5\left(1+\sqrt{97}\right)} \simeq \pm 2{,}329\,040\,339\,045$$

Dans la suite, cependant, nous essaierons de retrouver numériquement ces racines.

Si l'on part d'une estimation proche de la racine positive, par exemple $x_0 = 2$, les choses se passent de façon tout à fait satisfaisante [13] :

```
In []: Newton(f, fprime, 2.0, 1e-10, 25)
 x_0 = 2.0
 x_1 = 2.258969048842402
 x_2 = 2.3257002405632394
 x_3 = 2.3290326605303693
 x_4 = 2.329040339004226
Out[]: 2.329040339044829
```

12. Ou bien pourra-t-on utiliser le site `www.wolframalpha.com/input` qui les fournit obligeamment lorsque l'on entre l'expression de la fonction f.
13. Le programme a ici été modifié afin d'afficher les valeurs de x_n à chaque étape.

Mais regardons ce qui se passe pour $x_0 = 0.8$, $x_0 = 0.79$ et $x_0 = 0.78$, tous trois situés dans la partie croissante de la fonction juste après son second minimum, mais avant le point d'inflexion, dans la partie convexe.

Les choses peuvent se passer tout aussi bien :

```
In []: Newton(f, fprime, 0.8, 1e-10, 25)
 x_0 = 0.8
 x_1 = 3.6329055104033756
 x_2 = 0.9193684032463021
 x_3 = 2.033289727634883
 x_4 = 2.2720388051846196
 x_5 = 2.326824898537034
 x_6 = 2.32903696024019
 x_7 = 2.329040339036967
Out[]: 2.329040339044829
```

Mais il n'en faut pas beaucoup pour que la suite diverge :

```
In []: Newton(f, fprime, 0.79, 1e-10, 25)
 x_0 = 0.79
 x_1 = 4.010531016058355
 x_2 = -0.1244748665655937
 x_3 = 6.419383442477584
 x_4 = -16.241223371359197
 x_5 = 402.444478364447
 x_6 = -6517409.568919649
 x_7 = 2.7683757845243564e+19
 x_8 = -2.1216567589356818e+57
OverflowError: (34, 'Result too large')
```

Ou bien que l'on converge vers une autre racine (ici la racine négative, même si l'on est parti d'une estimation plus proche de la racine positive) :

```
In []: Newton(f, fprime, 0.78, 1e-10, 25)
 x_0 = 0.78
 x_1 = 4.4983149880022335
 x_2 = -1.9868945747055395
 x_3 = -2.253485103929354
 x_4 = -2.3251607721487493
 x_5 = -2.3290299808343544
 x_6 = -2.3290403389709406
Out[]: -2.329040339044829
```

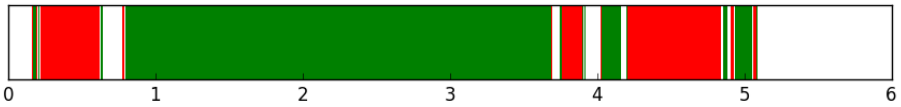
Certaines valeurs initiales peuvent conduire à des comportements oscillants [14], voire

14. Quoique fréquemment, les arrondis dûs aux calculs flottants finissent par briser le cycle.

chaotiques :

```
In []: Newton(f, fprime, 0.7985153381095748, 1e-10, 10)
 x_0 = 0.7985153381095748
 x_1 = 3.6834209129562647
 x_2 = 0.7985153381095764
 x_3 = 3.6834209129562074...
```

Le graphique ci-dessous indique, en fonction de la valeur de x_0, si la méthode de Newton converge vers la racine positive, la racine négative, ou ne converge pas (zones blanches). On remarque une zone de convergence autour de la racine positive, mais en-dehors ce cette zone, les trois comportements sont observés :



On constate une très forte sensibilité aux conditions initiales [15], une caractéristique qui a été utilisée pour construire des courbes fractales.

5.4 Utilisation de `scipy`

La méthode de Newton-Raphson est disponible dans le module `scipy.optimize`, via la fonction `newton`, dont voici une partie de la documentation :

```
newton(func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None)
    Find a zero using the Newton-Raphson or secant method.

    Find a zero of the function `func` given a nearby starting point `x0`.

    func : function
        The function whose zero is wanted. It must be a function of a
        single variable of the form f(x,a,b,c...), where a,b,c... are extra
        arguments that can be passed in the `args` parameter.
    x0 : float
        An initial estimate of the zero that should be somewhere near the
        actual zero.
    fprime : function, optional
        The derivative of the function when available and convenient. If it
        is None (default), then the secant method is used.
    maxiter : int, optional
        Maximum number of iterations.
```

On remarque notamment dans cette documentation que si l'on ne fournit pas de fonction dérivée à l'algorithme, il utilise la méthode de la sécante. On pourrait envisager d'utiliser une dérivation numérique lorsque l'on n'a pas d'expression pour f', mais les difficultés soulevées précédemment quant au choix du pas pour ce calcul auraient rendu la chose très délicate.

En dehors d'un ordre légèrement différent pour les paramètres, les résultats obtenus sont les mêmes [16] :

```
In []: newton(f, 0.8, fprime)
Out[]: 2.3290403390044829

In []: newton(f, 0.78, fprime)
Out[]: -2.3290403390044829

In []: newton(f, 0.79, fprime)

OverflowError: (34, 'Result too large')
```

6 Étude plus poussée de la convergence

6.1 Ordre de convergence

Chacune des méthodes que nous avons étudiées au cours de ce chapitre est itérative : dans chaque cas, on dispose d'une suite de réels $(x_n)_{n\in\mathbb{N}}$ qui, si tout se passe bien, converge vers une racine.

On peut en effet prendre pour x_n le milieu des intervales successifs dans le cas de la méthode dichotomique, et l'intersection des cordes et de l'axe des abscisses dans le cas de la méthode de la fausse position.

Si la suite $(x_n)_{n\in\mathbb{N}}$ converge vers une limite c, on dit que la convergence est d'ordre $r \geqslant 1$ lorsqu'il existe une suite $(e_n)_{n\in\mathbb{N}}$ vérifiant, pour tout $n \in \mathbb{N}$,

$$|x_n - c| \leqslant e_n \quad \text{et} \quad \lim_{n\to\infty} \frac{e_{n+1}}{e_n^r} = \mu > 0$$

L'ordre r permet de quantifier la vitesse de convergence de la suite : plus l'ordre r est grand, plus la convergence sera rapide.

Si une convergence est d'ordre 1, le nombre de décimales exactes est augmenté d'au moins $-\log_{10}(\mu)$ à chaque itération (précisons qu'il faut $\mu < 1$ dans ce cas pour que la majoration garantisse la convergence de la suite).

Si une convergence est d'ordre $r > 1$, le nombre de décimales exactes est au moins multiplié par r à chaque itération, ce qui est bien plus rapide.

15. Il est possible d'observer les trois comportements sur des intervales bien plus petits que $[0.78, 0.8]$.

16. Le lecteur curieux pourra aller vérifier, dans le fichier `zeros.py` du module `scipy`, que la fonction est écrite de façon quasi-identique à celle proposée ici, nonobstant quelques subtilités supplémentaires pour en rendre l'usage plus général.

Une méthode itérative de détermination de racine donnant **généralement** des suites ayant une convergence d'ordre r seront également qualifiées de méthode d'ordre r. Notons que cela ne garantit pas une convergence d'ordre r dans *tous* les cas cependant. En effet, nous avons déjà vu que la plupart des méthodes itératives ne peuvent garantir une convergence, sans même parler d'une convergence rapide !

6.2 Cas de la méthode dichotomique

De façon évidente, dans le cas de la méthode dichotomique,

$$|x_n - c| \leqslant \frac{b-a}{2^n}$$

Par ailleurs, si l'on pose $e_n = \dfrac{b-a}{2^n}$, alors $\lim\limits_{n\to\infty} \dfrac{e_{n+1}}{e_n} = \dfrac{1}{2}$.

La méthode dichotomique est donc une méthode d'ordre 1. On retrouve le fait que l'on gagne environ 0.3 décimale à chaque itération ($-\log_{10}(1/2) \simeq 0.3$).

6.3 Cas de la méthode de Newton-Raphson

Si f est $\mathscr{C}^2$ au voisinage d'un point c pour lequel $f(c) = 0$ et $f'(c) \neq 0$, alors il existe un voisinage $\mathscr{V}$ de c tel que, quel que soit $x_0 \in \mathscr{V}$, la suite de Newton-Raphson $(x_n)_{n\in\mathbb{N}}$ converge vers c avec une convergence d'ordre 2.

Preuve

Supposons que $f'(c) > 0$ (si ce n'est pas le cas, il suffira d'appliquer le raisonnement à la fonction $-f$).

On peut trouver un réel $m_1 > 0$ ainsi qu'un intervale $[a,b]$ contenant c, sur lequel $f'(x) > m_1$, et on a $f(a) < 0 < f(b)$.

Posons la fonction ϕ, $\mathscr{C}^1$ sur $[a,b]$, définie par $x \mapsto \phi(x) = x - \dfrac{f(x)}{f'(x)}$.

$$\phi(x) - c = x - c - \frac{f(x)}{f'(x)} = x - c - \frac{f(x)-f(c)}{f'(x)} = \frac{f(c) - \big(f(x) + (c-x)f'(x)\big)}{f'(x)}$$

Si M_2 est un majorant de $|f''(x)|$ sur $[a,b]$, alors on peut écrire

$$\big|f(c) - \big(f(x) + (c-x)f'(x)\big)\big| \leqslant \frac{M2}{2}(c-x)^2 \qquad \text{donc} \qquad \big|\phi(x) - c\big| \leqslant \frac{M2}{2m_1}(c-x)^2$$

Posons à présent $K = \dfrac{M2}{2m_1}$.

On peut trouver un réel $\eta > 0$ vérifiant $K\eta < 1$ et $[c-\eta, c+\eta] \subset [a,b]$.

Pour tout $x \in [c-\eta, c+\eta]$, $\big|\phi(x) - c\big| \leqslant \eta$, donc $\phi(x) \in [c-\eta, c+\eta]$.

La suite $(x_n)_{n\in\mathbb{N}}$ de la méthode de Newton, définie par $x_{n+1} = \phi(x_n)$, est donc bien définie pour n'importe quel valeur x_0 choisie dans l'intervale $[c-\eta, c+\eta]$.

Par ailleurs, $\dfrac{\big|\phi(x) - c\big|}{(x-c)^2} \leqslant K$. On a donc $|x_n - c| \leqslant \dfrac{1}{K} \times \big(K(x_0 - c)\big)^{2^n}$.

Cette dernière expression garantit la convergence de la suite vers c pour n'importe quel valeur x_0 choisie dans l'intervale $[c-\eta, c+\eta]$, puisque $K|x_0 - c| \leqslant K\eta < 1$.

Enfin, en posant $e_n = \dfrac{1}{K} \times \big(K(x_0 - c)\big)^{2^n}$, on a $|x_n - c| \leq e_n$ avec $\lim_{n\to\infty} \frac{e_{n+1}}{e_n^2} = K > 0$, ce qui signifie que la convergence est quadratique.

Avec des informations supplémentaires sur la fonction, il est possible d'étendre le résultat précédent à un intervale plus large.

Ainsi, dans le cas d'une fonction convexe, on peut montrer que la suite $(x_n)_{n\in\mathbb{N}}$ converge toujours, quelle que soit la valeur initiale x_0, pourvu que $f(x_0)$ soit positif. Il en est de même pour le cas d'une fonction concave en partant de n'importe quelle valeur initiale x_0 vérifiant $f(x_0) < 0$. Ces convergences sont quadratiques puisque l'on finira par entrer dans l'intervale discuté précédemment.

À l'inverse, si la fonction est tangente à l'axe des abscisses, la convergence est plus lente (on retombe dans une convergence linéaire). Il est toutefois parfois possible de retrouver une convergence quadratique en modifiant légèrement la méthode. Ainsi, si l'on s'intéresse à une racine d'ordre p d'un polynôme, il faudra utiliser la relation de récurrence suivante [17] :

$$x_{n+1} = x_n - p \times \frac{f(x_n)}{f'(x_n)}$$

6.4 Cas des autres méthodes

On peut montrer que la méthode de la fausse position est d'ordre 1, ce qui la rend peu intéressante en pratique.

La méthode de la sécante, elle, est d'ordre $\dfrac{1+\sqrt{5}}{2} \simeq 1.618$.

Bien qu'elle converge généralement un peu moins vite que la méthode de Newton, elle présente cependant l'avantage, comme on l'a dit, de ne pas nécessiter la connaissance de f', et elle est généralement un peu plus stable.

Par exemple, les conditions initiales $x_0 = 0.78$, $x_0 = 0.79$, et $x_0 = 0.8$ convergent toutes vers la racine positive de notre problème lorsque l'on utilise la méthode de la sécante [18]

17. Attention, cette relation donnera une suite divergente avec une racine qui ne serait pas d'ordre p !
18. On rappelle que la fonction `newton` de `scipy` utilise la méthode de la *sécante* lorsque f' n'est pas fournie.

proposée par scipy :

```
In []: newton(f, 0.80)
Out[]: 2.3290403390448287

In []: newton(f, 0.78)
Out[]: 2.329040339044829

In []: newton(f, 0.79)
Out[]: 2.32904033904483
```