Résolution d'équations et procédés itératifs

© Pierre Lantagne
Enseignant retraité du Collège de Maisonneuve

Ce document est une révison de ceux produits en 2001 et 2004. L'objectif principal de ce document est de présenter les procédés itératifs que sont les méthodes de bissection, d'interpolation linéaire et de Newton-Raphson pour approcher les zéros réels d'une fonction réelle. La présentation de ces différents procédés itératifs est l'occasion pour l'élève d'être initier à certains éléments fondamentaux de programmation Maple que sont les structures de contrôle et les procédures.

Bonne lecture à tous !

* Ce document Maple est exécutable avec la version 2020.1

Initialisation

```
> restart;
> with(plots,display,setoptions):
> setoptions(xtickmarks=12,ytickmarks=12,
             labels=["x","y"],
             labelfont=[TIMES,ITALIC,10],
             labeldirections=[horizontal,horizontal],
             axesfont=[TIMES,ROMAN,6],
             size=[300,300]);
> interface(imaginaryunit=i);
```

$$I \tag{1.1}$$

Rappel: macro-commande solve

Il n'y a pas de méthode analytique générale pour la résolution des équations. Il en existe, par contre, pour certains cas d'équations bien particuliers. On a seulement à penser à la formule quadratique pour la résolution d'une équation du deuxième degré $ax^2+bx+c=0$.

```
> Équation:=a*x^2+b*x+c=0;
```

$$Équation := a x^2 + b x + c = 0 \tag{2.1}$$

```
> Racines:=solve(Équation,{x});
```

$$Racines := \left\{x = \frac{-b+\sqrt{-4ac+b^2}}{2a}\right\}, \left\{x = -\frac{b+\sqrt{-4ac+b^2}}{2a}\right\} \tag{2.2}$$

C'est effectivement cette formule que la macro-commande `solve` applique lorsque l'équation à résoudre est reconnue comme une équation du deuxième degré. Dans ce cas, Maple a la capacité de toujours formuler les racines de manière symbolique en utilisant évenuellement la notation racine carrée.

Soit à résoudre l'équation $2x^2+3x-4=0$.

```
> Équation:=2*x^2+3*x+4=0;
```

$$Équation := 2x^2 + 3x + 4 = 0 \quad \textbf{(2.3)}$$

```
> Racines:=solve(Équation,{x});
```

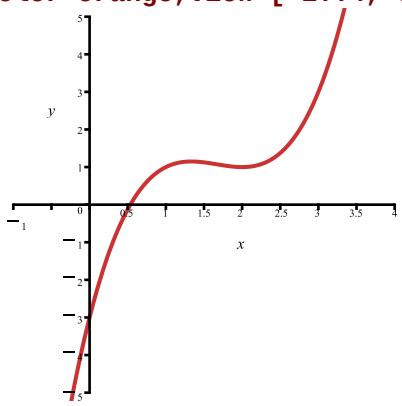
$$Racines := \left\{x = -\frac{3}{4} + \frac{i\sqrt{23}}{4}\right\}, \left\{x = -\frac{3}{4} - \frac{i\sqrt{23}}{4}\right\} \quad \textbf{(2.4)}$$

L'ensemble solution est donc E.S = $\left\{\frac{-3 \pm \sqrt{23}\, i}{4}\right\}$.

Graphiquement, il est facile de se convaincre de l'existence ou de l'inexistence de *racines réelles* d'une équation. Pour cela, il suffit de transformer l'équation à résoudre en une équation équivalente dont le membre de droite de l'égalité est 0, puis de créer une fonction dont la règle de correspondance est le membre de gauche. Alors, graphiquement, à chaque endroit dans le **domaine de la fonction** où la courbe touche l'axe des x correspond un zéro réel de la fonction et donc à une racine réelle de l'équation à résoudre.

Soit à résoudre l'équation $x^3 - 5x^2 + 8x - 3 = 0$. Créons le tracé de la fonction f définie par $f(x) = x^3 - 5x^2 + 8x - 3$.

```
> f:=x->x^3-5*x^2+8*x-3:
  plot([x,f(x),x=-1..4],color=orange,view=[-1..4,-5..5]);
```



Puisque la fonction f est continue sur $\mathbb{R}$ et donc en particulier sur l'intervalle [0,1], il est clair que cette fonction possède (au moins) un zéro réel et que celui-ci se trouve dans l'intervalle [0,1] (on pourrait montrer assez facilement, à l'aide de la règle des signes de Descartes, que la fonction f ne possède aucun autre zéro réel ou encore par l'étude de la monotonie de la fonction f).

En plus de la formule quadratique, il existe aussi une formule générale pour obtenir les racines des équations polynomiales de degré 3 et une formule générale également pour les équations polynomiales de degré 4. Or, en 1829, Évariste Galois a démontré qu'il n'en existe pas pour les équations polynomiales de degré supérieur à 4. Tout de même, il est possible, pour certains cas bien particuliers d'équation polynomiale d'un degré supérieur à 4, d'obtenir analytiquement certaines de ses racines. Mais, évidemment, si cela est possible, c'est sans passer par une formule générale spécifique pour les polynômes de ce degré.

Dans le cas de l'équation polynomiale de degré 3 précédente, on peut donc trouver exactement la racine réelle comprise dans l'intervalle [0,1]. Il suffit d'utiliser la macro-commande `solve` qui applique, bien sûr, la formule générale pour les équations de degré 3.

```
> Équation:=x^3-5*x^2+8*x-3=0;
```

$$Équation := x^3 - 5x^2 + 8x - 3 = 0 \quad \textbf{(2.5)}$$

```
> Racines:=solve(Équation,x);
```

$$Racines := -\frac{\left(116+12\sqrt{93}\right)^{1/3}}{6}-\frac{2}{3\left(116+12\sqrt{93}\right)^{1/3}}+\frac{5}{3},\ \frac{\left(116+12\sqrt{93}\right)^{1/3}}{12}+\frac{1}{3\left(116+12\sqrt{93}\right)^{1/3}}+\frac{5}{3}+\frac{i\sqrt{3}\left(-\frac{\left(116+12\sqrt{93}\right)^{1/3}}{6}+\frac{2}{3\left(116+12\sqrt{93}\right)^{1/3}}\right)}{2},\ \frac{\left(116+12\sqrt{93}\right)^{1/3}}{12}+\frac{1}{3\left(116+12\sqrt{93}\right)^{1/3}}+\frac{5}{3}-\frac{i\sqrt{3}\left(-\frac{\left(116+12\sqrt{93}\right)^{1/3}}{6}+\frac{2}{3\left(116+12\sqrt{93}\right)^{1/3}}\right)}{2} \quad \textbf{(2.6)}$$

Les trois racines précédentes découlent de l'application de la formule générale pour les racines d'un polynôme de degré 3. L'évaluateur a donné trois racines à cette équation car, d'une part, l'évaluateur utilise l'ensemble des nombres complexes comme référentiel et, d'autre part, le théorème de d'Alembert (1746) portant sur le nombre de racines d'un polynôme établi que tout polynôme à coefficients complexes de degré $n > 0$ possède n racines complexes. Lorsqu'une racine est de multiplicité m, elle est compté m fois.

Pour ne faire afficher que les racines réelles, utilisons l'astuce suivante qui consiste à résoudre simultanément l'équation avec l'inégalité $x < \infty$. Cela exclurera évidemment les racines imaginaires.

```
> Racine_réelle:=solve({Équation,x<infinity},x);
```

$$Racine\_réelle := \left\{x = -\frac{\left(116+12\sqrt{93}\right)^{1/3}}{6}-\frac{2}{3\left(116+12\sqrt{93}\right)^{1/3}}+\frac{5}{3}\right\} \quad \textbf{(2.7)}$$

Nous pouvons également utiliser la macro-commande `solve` dans l'environnement `RealDomain`.

```
> use RealDomain in solve(Équation,{x}) end use;
```

$$\left\{x = -\frac{\left(116+12\sqrt{93}\right)^{1/3}}{6}-\frac{2}{3\left(116+12\sqrt{93}\right)^{1/3}}+\frac{5}{3}\right\} \quad \textbf{(2.8)}$$

La macro-commande evalf permet d'obtenir une approximation décimale.

```
> evalf(Racine_réelle);
```

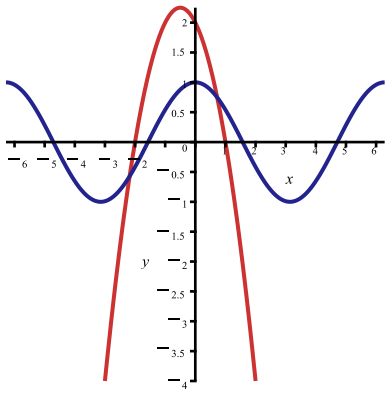
$$\{x = 0.5344287680\} \quad \textbf{(2.9)}$$

En mathématique, les équations à résoudre ne sont pas nécessairement des équations polynomiales. L'exemple suivant est pertinent pour mettre en évidence les limites de la macro-commande `solve` à résoudre (analytiquement) des équations en générale.

Soit à résoudre l'équation $-x^2 - x + 2 = \cos(x)$.

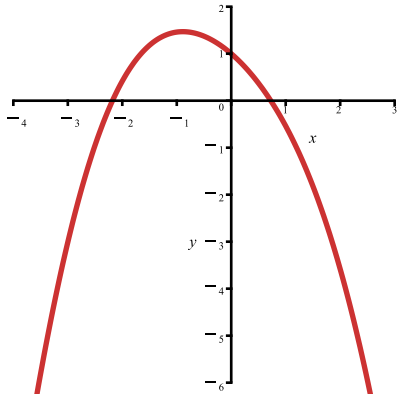
Traçons les graphiques appropriés pour s'assurer de l'existence ou de l'inexistence de racines réelles.

```
> Courbe_1:=plot([x,-x^2-x+2,x=-3..2],color=orange):
  Courbe_2:=plot([x,cos(x),x=-2*Pi..2*Pi],color=navy):
  display([Courbe_1,Courbe_2]);
```



Puisque les fonctions $x \rightarrow -x^2 - x + 2$ et $x \rightarrow \cos(x)$ sont toutes deux continues sur tous les réels $\mathbb{R}$, il est graphiquement évident que l'équation $-x^2 - x + 2 = \cos(x)$ possède exactement deux racines réelles et donc que la fonction f définie par $f(x) = -x^2 - x + 2 - \cos(x)$ possède deux zéros réels. Obtenons le tracé de la fonction f sur l'intervalle [-4, 3].

```
> f:=x->-x^2-x+2-cos(x):
  plot([x,f(x),x=-4..3],color=orange,thickness=2,view=[-4..3,-6..2],
  xtickmarks=6);
```



Puisque la fonction f est continue sur $\mathbb{R}$ et donc en particulier sur l'intervalle [-3,1], on observe donc l'existence de deux zéros réels distincts dans l'intervalle [-3,1]. Résolvons alors simultanément l'équation $f(x) = 0$ avec l'inéquation $x < \infty$.

```
> Racines:=solve({f(x)=0,x<infinity},{x});
```

$$Racines := \{\ \}$$ **(2.10)**

Qu'est-ce qui ce passe ? La macro-commande `solve` n'a pu donner aucune racine réelle. La fonction f étant une fonction continue qui touche (ici coupe) l'axe des *x* deux fois sur l'intervalle [-3,1], elle possède donc au moins deux zéros réels. **Cet exemple montre que si Maple n'affiche aucun résultat, ce n'est pas nécessairement parce qu'il n'en y a pas**. La macro-commande `solve` commande une résolution analytique de l'équation impliquant des formules algébriques et, pour ce type d'équation, il n'existe pas de telles formules. C'est ce qui explique le précédent résultat. En résumé, si la macro-commande `solve` ne donne aucune solution réelle, ce n'est pas nécessairement parce qu'il n'en existe pas. Il faut s'en assurer autrement.

Lorsqu'il existe des formules pouvant être utilisées, les macro-commandes `solve` et `RootOf` les appliquent. Lorsque cela n'est pas possible compte tenue de la nature de l'équation à résoudre, ces macro-commandes s'avèrent donc impuissantes à formuler *exactement* et symboliquement les racines, lorsqu'il en existe, bien sûr. Dans ce cas, il est nécessaire de procéder autrement. Il faut donc faire intervenir des méthodes numériques d'approximations. C'est exactement ce que la macro-commande fsolve fait.

```
> fsolve(f(x)=0);
```

$$0.7254899240 \qquad \textbf{(2.11)}$$

Si on n'utilise aucune des options permises, la macro-commande `fsolve` cherchera à obtenir une seule racine.

Pour être en mesure de comprendre les différentes options que les macro-commandes `RootOf` et `fsolve` autorisent, vous allez être introduit à trois méthodes numériques d'approximations successives d'une racine réelle d'une équation à coefficients réels:

– *Méthode de bissection*
– *Méthode d'interpolation linéaire*
– *Méthode de Newton-Raphson*

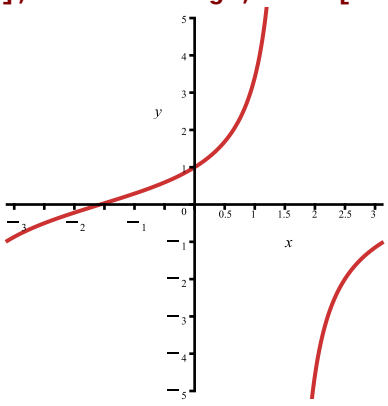
Chacune des trois méthodes ne sont pas, en fait, utilisées telles quelles par l'évaluateur de Maple. La pertinence de les présenter ici réside dans la simplicité de leur procédé itératif. En comparant leur procédé respectif avec différents exemples, vous allez constater qu'une méthode peut être plus efficace qu'une autre. De plus, la transposition de chacune de ces méthodes sur le plan informatique est une excellente occasion d'être introduit aux éléments de programmation que sont les structures de contrôle et les procédures.

Remarque

En plus de l'approche graphique, il ne faut jamais sous-estimé l'importance de s'attarder également sur l'analyse de la continuité d'une fonction sur un intervalle donné pour se convaincre de l'existence d'un zéro réel de cette fonction sur cet intervalle. Si c'est uniquement l'approche graphique qui est utilisée et qu'on néglige d'en faire l'analyse, on peut parfois être leurrer dans la localisation d'un zéro réel.

Soit l'équation $\sec(x) + \tan(x) = 0$. Obtenons le tracé de la fonction f définie par $f(x) = \sec(x) + \tan(x)$ sur l'intervalle $[-\pi, \pi]$.

```
> f:=x->sec(x)+tan(x):
  plot([x,f(x),x=-Pi..Pi], color=orange,view=[-Pi..Pi,-5..5],discont=true);
```



Si on se fit uniquement au tracé de la fonction f, on devrait conclure qu'il existe effectivement un zéro réel dans l'intervalle [-2,-1]. Or, à l'endroit où la fonction f semble toucher l'axe des *x* dans l'intervalle [-2,-1], la

fonction f n'est pas définie. À cet endroit, la fonction f présente une discontinuité réductible, c'est-à-dire un trou.

En effet, il suffit de transformer $f(x)$ en sinus et en cosinus pour en être convaincu.

```
> f(x)=normal(convert(f(x),sincos));
```

$$\sec(x) + \tan(x) = \frac{1 + \sin(x)}{\cos(x)} \qquad \textbf{(2.1.1)}$$

Pour $x = -\frac{\pi}{2} \in [-2,-1]$, la fonction f n'est pas définie et elle ne touche donc pas l'axe des x. C'est bien en cet endroit que la fonction semble couper l'axe des x puisque qu'au voisinage de $x = -\frac{\pi}{2}$, $f(x)$ est voisin de 0. Calculons la limite pour le montrer.

```
> Limit(f(x),x=-Pi/2)=limit(f(x),x=-Pi/2);
```

$$\lim_{x \to -\frac{\pi}{2}} (\sec(x) + \tan(x)) = 0 \qquad \textbf{(2.1.2)}$$

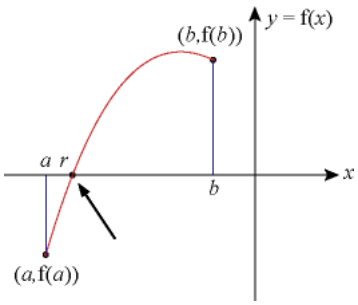
Méthode de bissection

La méthode de bissection est une méthode plutôt élémentaire d'approximations successives d'un zéro réel d'une fonction réelle. Cette méthode repose sur le théorème de la valeur intermédiaire. Le théorème de la valeur intermédiaire établit que si f est une fonction telle que

1) f est continue sur un intervalle [a,b]
2) f(a) et f(b) sont de signes contraires

alors il existe au moins un nombre réel $r \in]a,b[$ tel que $f(r) = 0$.

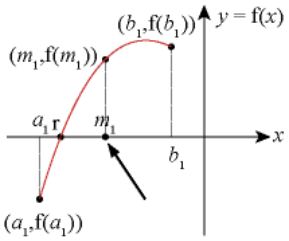
Sans perte de généralité, envisageons la situation où il ne se présente qu'un seul zéro réel $r \in]a,b[$.



Pour les développements qui vont suivre, renommons l'intervalle [a,b] par $[a_1, b_1]$.

Le calcul du milieu, noté m_1, de l'intervalle $[a_1, b_1]$ donne une première valeur approchée du zéro réel r de la fonction.

$$m_1 = \frac{a_1 + b_1}{2}$$

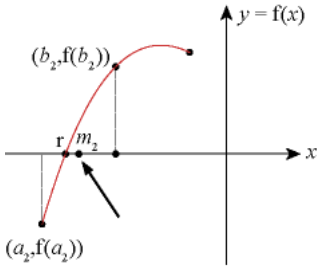


On dit aussi que m_1 est une approximation du zéro réel *r*. Dans le cas où le tracé de la fonction n'est pas une droite, on a que $|r - m_1| < \frac{b_1 - a_1}{2}$. La valeur m_1 réalise une première bissection de l'intervalle $[a_1, b_1]$. Analysons ensuite le signe du produit $f(a_1)\, f(m_1)$:

- si $f(a_1)\, f(m_1) < 0$, alors, en vertu du théorème de la valeur intermédiaire, il y a donc un zéro (réel) dans l'intervalle $]\, a_1, m_1\, [$
- sinon, c'est le produit $f(m_1)\, f(b_1)$ qui doit être négatif et il y aurait donc un zéro (réel) dans l'intervalle $]\, m_1, b_1\, [$

Si le produit $f(a_1)\, f(m_1)$ est nul, dans ce cas, $f(m_1) = 0$ et nous avons alors trouvé un zéro réel exact de la fonction f (une racine de l'équation). Mais, il faudrait être plutôt chanceux. Ne comptons pas trop sur cette chance.

On peut répéter ce processus chaque fois que le théorème de la valeur intermédiaire s'applique. Désignons par $[a_2, b_2]$ l'intervalle où le théorème de la valeur intermédiaire s'applique. En répétant la bissection de l'intervalle $[a_2, b_2]$ avec la valeur du milieu $m_2 = \frac{a_2 + b_2}{2}$, on obtient une deuxième approximation m_2 d'un zéro de la fonction et $|r - m_2| < \frac{b_2 - a_2}{2}$. Et nous avons, bien sûr $|r - m_2| < |r - m_1|$.



En répétant ce processus aussi longtemps que l'on veut, on crée une suite d'approximations successives m_1, m_2, m_3, … et une suite de sous-intervalles successifs $[a_1, b_1]$, $[a_2, b_2]$, $[a_3, b_3]$, …. où chaque sous-intervalle de la suite contient le zéro réel de la fonction f et dont la largeur de chacun d'eux est égale à la moitié de la largeur du sous-intervalle précédent. Dans la pratique, nous appliquerons la bissection des différents sous-intervalles $[a_k, b_k]$ jusqu'à ce que le zéro réel à localiser soit à l'intérieur d'un *n-ième* sous-intervalle dont la demi largeur $d_n = \frac{b_n - a_n}{2}$ sera inférieure à une certaine valeur fixée au départ. On désignera par la lettre *E* cette valeur fixée au départ et on appellera cette valeur "écart maximal toléré". Ce procédé itératif sera donc interrompu à la *n-ième* itération lorsque $d_n =$

$$\frac{b_n - a_n}{2} < E.$$

Algorithme de la méthode de bissection

Soit f une fonction continue sur un intervalle $[a_1, b_1]$ telle que l'on ait $f(a_1)\, f(b_1) < 0$.
Soit E désignant la valeur maximale tolérée de l'écart, c'est-à-dire $E = |r - m_n|$.

Répéter les étapes 1 à 5 pour $k = 1, 2, 3, \ldots, n,$ jusqu'à ce que $d_n < E$

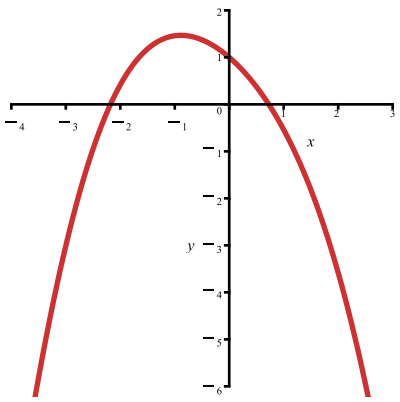
Tant que $d_k < E$

1. Calculer $m_k = \frac{a_k + b_k}{2}$
2. Calculer $f(m_k)$. Si $f(m_k) = 0$, on arrête. Un zéro exact est trouvé.
3. Calculer $d_k = \frac{b_k - a_k}{2}$
4. Si $f(a_k)\, f(m_k) < 0$, alors posons $a_{k+1} = a_k$ et $b_{k+1} = m_k$
5. Si $f(a_k)\, f(m_k) > 0$, alors posons $a_{k+1} = m_k$ et $b_{k+1} = b_k$

La méthode de bissection fonctionne toujours, c'est-à-dire que $\lim_{n \to \infty} m_n = r$. Avec la méthode de bissection, on est donc assuré de la convergence des approximations successives vers un zéro réel. Cette convergence n'est pas très rapide habituellement. À l'aide d'exemples, on le constatera en comparant le nombre d'itérations que mets chacune des trois méthodes avant d'atteindre la valeur maximale tolérée de l'écart .

Comme premier exemple d'application de la méthode de bissection, considérons de nouveau la fonction f définie par $f(x) = -x^2 - x + 2 - \cos(x)$. Appliquons l'algorithme de bissection avec la fonction f.

```
> f:=x->-x^2-x+2-cos(x):
  plot([x,f(x),x=-4..3],color=orange,thickness=2,view=[-4..3,-6..2],
  xtickmarks=6);
```



Puisque la fonction f est continue sur $\mathbb{R}$, elle est donc continue sur l'intervalle [-3,-2]. La fonction f est aussi continue sur l'intervalle [0,1]. Dans les deux cas, le théorème de la valeur intermédiaire s'applique. Obtenons d'abord une approximation du zéro réel localisé l'intervalle [-3,-2].

Pour transposer sur le plan informatique l'algorithme de bissection, il est nécessaire de prendre connaissance des instructions de programmation que sont:

– les boucles
– les tests

Présentons alors brièvement ces différentes structures de contrôles.

Les boucles sont utiles afin d'exécuter un bloc de requêtes que l'on désire répéter. On utilisera la boucle for si le nombre de répétitions du bloc de requêtes est connu sinon on utilisera la boucle while.

La structure d'une boucle **for** est la suivante:

for *indice* **from** *valeur_début* **to** *valeur_fin* **by** *incrément* **do**
bloc de requêtes
od;

Exemple: Soit l'impression des 5 premiers entiers impairs positifs.

```
> for k from 0 to 4 by 1 do
   print(2*k+1);
  od;
```

1
3
5
7
9 **(3.1)**

```
> k;
```

5 **(3.2)**

Le nombre de répétition était connu d'avance. Le bloc de requêtes a effectivement été répété 5 fois.

Libérons la variable *k*.

```
> k:='k':
```

La structure d'une boucle **while** est la suivante:

while *condition vraie* **do**
bloc de requêtes
od;

Exemple: Soit la division par 4 du nombre 256. Poursuivons la division par 4 des quotients successifs aussi longtemps que chaque quotient successif demeure supérieur à 0,0001.

```
> j:=0:
  quotient:=256/4:
  while quotient > 0.0001 do
    j:=j+1:
    quotient:=evalf(quotient/4);
  od;
```

$$j := 1$$

$$quotient := 16.0000000000$$

$$j := 2$$
$$quotient := 4.0000000000$$
$$j := 3$$
$$quotient := 1.0000000000$$
$$j := 4$$
$$quotient := 0.2500000000$$
$$j := 5$$
$$quotient := 0.0625000000$$
$$j := 6$$
$$quotient := 0.0156250000$$
$$j := 7$$
$$quotient := 0.0039062500$$
$$j := 8$$
$$quotient := 0.0009765625$$
$$j := 9$$
$$quotient := 0.0002441406$$
$$j := 10$$
$$quotient := 0.0000610352 \quad \textbf{(3.3)}$$

Le nombre de répétition était inconnu d'avance.

Libérons la variable *j*.

```
> j:='j':
```

Les tests sont utiles lorsqu'il faut, selon un contexte donné, exécuter conditionnellement un bloc de requêtes.

La structure de base d'un test if est la suivante:

if *condition vraie* **then**
  bloc de requêtes
fi;

On peut également exécuter conditionnellement un autre bloc de requêtes selon que la condition soit vraie ou fausse.

if *condition vraie* **then**
  bloc de requêtes A
else
  bloc de requêtes B
fi;

Il est possible également d'utiliser plusieurs tests imbriqués.

if *condition vraie* **then**
  bloc de requêtes A
elif *condition vraie* **then**
  bloc de requêtes B
elif *condition vraie* **then**

bloc de requêtes C
.....
else
bloc de requêtes par défaut
fi;

Exemple simplet.

```
> a:=-8;
  b:=2;
  if a=b then
    print(`a et b sont égaux`)
  elif a<b then
    print(`a est inférieur à b`)
  else
    print(`a est supérieur à b`)
  fi;
```

$$a := -8$$

$$b := 2$$

a est inférieur à b **(3.4)**

Vous pouvez expérimenter les tests imbriqués précédents en modifiant les valeurs de *a* et *b* et en exécutant à nouveau le bloc précédent.

Suggestion: Par exemple, sans changer la valeur de *a*, donnez à *b* la valeur $\sqrt{2}$ ou même π.

– Qu'est-ce qui se passe ?
– Comment pouvez-vous expliquer ce qui arrive ?
– Modifiez les requêtes qu'il faut pour que ça marche.

Revenons maintenant à notre exemple. On pourrait, bien sûr, appliquer étape par étape l'algorithme de bissection, mais, avec la connaissance de ces structures de contrôle, nous sommes maintenant en mesure d'automatiser cet algorithme avec un programme informatique. Un programme informatique est constitué d'un certain nombre de requêtes formant un bloc d'instructions permettant d'automatiser l'exécution de tâches plus ou moins complexes. Cette première transposition de l'algorithme de bissection dans un programme ne testera pas si le théorème de la valeur intermédiaire s'applique sur l'intervalle en cause.
La programmation de l'algorithme supposera que les conditions de son applicabilité sont satisfaites.

Rappelons que $f(-3)\,f(-2) < 0$. En effet,

```
> f:=x->-x^2-x+2-cos(x);
  is(f(-3)*f(-2),negative);
```

$$f := x \mapsto -x^2 - x + 2 - \cos(x)$$

true **(3.5)**

Obtenons la séquence de valeurs m_1, m_2, m_3, … avec $E = 0{,}00001$. Incluons, dans la transposition informatique de l'algorithme de bissection décrit plus haut, le calcul du nombre d'itérations que la boucle mets pour que $d_n = \frac{b_n - a_n}{2} < E$.

Assignons d'abord les valeurs requises.

```
> a:=-3;
  b:=-2;
  E:=0.00001;
  Itération:=0;
```

$$a := -3$$

$$b := -2$$

$$E := 0.0000100000$$

$$Itération := 0 \qquad \textbf{(3.6)}$$

Exécutons ensuite le programme ci-dessous qui transpose sur le plan informatique l'algorithme de bissection.

```
> while evalf(abs(b-a)) > evalf(2*E) do
  m:= (a+b)/2;
  if is(f(m)*f(a),negative) then
    b:=m
  elif is(f(m)*f(a),positive) then
    a:=m
  else
    print(`Le zéro trouvé est exact et vaut `,m);
    return;
  fi;
  Itération:=Itération+1;
  od:
  m:=(a+b)/2:
  Itération:=Itération+1:
  print(`Le zéro réel approximatif est `,evalf(m));
  print(`Cela a nécessité `,Itération,`itérations`);
```

$$\textit{Le zéro réel approximatif est}\ ,\ -2.1799240110$$

$$\textit{Cela a nécessité}\ ,\ 17,\ \textit{itérations} \qquad \textbf{(3.7)}$$

Libérons les variables qui ont été utilisées dans le programme précédent.

```
> a:='a':b:='b':E:='E':Itération:='Itération':
```

On peut donc affirmer que le zéro cherché dans l'intervalle]-3,-2[est $r \approx$ -2,17992 avec une erreur maximale égale à 0,00001.
En fait, nous pouvons affirmer que $r \in$ [2,17991; 2,17993].

Évaluons f(-2,179924011) pour voir.

```
> f(-2.179924011);
```

$$7.5673000000\ 10^{-6} \qquad \textbf{(3.8)}$$

On pourrait fort bien exiger que f(r) soit davantage près de 0. Il suffirait alors de prendre un écart maximal E plus petit dans les calculs.

```
> a:=-3;
  b:=-2;
  E:=0.000000001;
  Itération:=0;
```

$$a := -3$$

$$b := -2$$

$$E := 1.0000000000\,10^{-9}$$

$$Itération := 0 \tag{3.9}$$

```
> while evalf(abs(b-a)) > evalf(2*E) do
  m:= (a+b)/2;
  if is(f(m)*f(a),negative) then
    b:=m
  elif is(f(m)*f(a),positive) then
    a:=m
  else
    print(`Le zéro trouvé est exact et vaut `,m);
    return;
  fi;
  Itération:=Itération+1;
  od:
  m:=(a+b)/2:
  Itération:=Itération+1:
  print(`Le zéro réel approximatif est `,evalf(m));
  print(`Cela a nécessité `,Itération,`itérations`);
```

$$\textit{Le zéro réel approximatif est}\ ,\ -2.1799269910$$

$$\textit{Cela a nécessité}\ ,\ 30,\ \textit{itérations} \tag{3.10}$$

Évaluons f(-2.179926991) pour voir.

```
> f(-2.179926991);
```

$$-6.0000000000\,10^{-10} \tag{3.11}$$

Libérons les variables qui ont été utilisées dans le programme précédent.

```
> a:='a':b:='b':E:='E':Itération:='Itération':
```

ATTENTION: Il faut, bien sûr, tenir compte de la valeur courante de la variable d'environnement Digits. Par exemple, en ne modifiant pas sa valeur par défaut (10 chiffres décimaux), il serait inapproprié de considérer, par exemple, une valeur $E = 1.\,10^{-15}$, car l'affichage du résultat ne se fera, de toutes façons, qu'avec 10 chiffres décimaux.

Pour être en mesure d'utiliser de manière plus convivial ce programme dans d'autres situations, par exemple, avec une valeur E de l'écart toléré différente ou avec un intervalle différent ou encore avec d'autres fonctions même, nous allons donc inclure les instructions composant le programme dans une procédure. Une procédure est en quelque sorte une macro-commande semblable à n'importe quelle autre macro-commande Maple.

La structure d'une procédure commence et se termine avec les mots clefs **proc** …**end proc**.

nom:= **proc**(arg_1, arg_2, …, arg_n)
global var_1, var_2, …, var_k;
local var_1, var_2, …, var_l;
corps de la procédure
end proc;

Voici un exemple simple d'une procédure ayant trois arguments qui donnera comme résultat le maximum de trois *nombres entiers* donnés.

```
> Maximum:=proc(a,b,c)
            local x;
```

```
            if a>b then x:=a else x:=b fi;
            if c>x then c else x fi
          end proc;
```

$$Maximum := \mathbf{proc}(a, b, c) \quad \textbf{(3.12)}$$

local x;

if $b < a$ **then** $x := a$ **else** $x := b$ **end if**; **if** $x < c$ **then** c **else** x **end if**

end proc

Expérimentons cette procédure avec les entiers -12, -25, -6. Ces entiers seront assignés, dans l'ordre, aux arguments formels a, b et c de la procédure *Maximum*.

```
> Maximum(-12,-25,-6);
```

$$-6 \quad \textbf{(3.13)}$$

Une procédure peut donc permettre d'acheminer un contenu à certaines variables du corps de la procédure par l'intermédiaire de ses arguments formels. Au moment d'écrire une procédure, le programmeur doit faire en sorte qu'elle doit tenir compte de toutes les situations qui empêcheraient le corps de la procédure de donner le résultat attendu. Par exemple, voyons comment la procédure *Maximum* retournera le maximum de trois nombres suivants: -1, 3 et $\sqrt{2}$.

```
> Maximum(-1,3,sqrt(2));
Error, (in Maximum) cannot determine if this expression is
true or false: 3 < 2^(1/2)
```

La procédure a planté sur une instruction du corps de la procédure car une requête de celui-ci n'a pas pu être exécutée correctement. On est heureusement ici assez bien informé sur la cause du plantage. Ce n'est pas toujours le cas. Pour réaliser une bonne programmation, il est d'abord nécessaire de contrôler la nature du contenu qui sera assigné ultérieurement à des variables du programme. De cette façon, on gère soi-même les limites qu'on impose à la macro-commande. Il était prévu au départ que chaque nombre qui serait passé à chaque argument formel serait un nombre entier. Alors, à l'aide de l'opérateur de conformité de type :: , testons le type « integer » de chaque nombre qui seront passé à chacun des paramètres formels a, b et c.

```
> Maximum:=proc(a::integer,b::integer,c::integer)
            local x;
            if a>b then x:=a else x:=b fi;
            if c>x then c else x fi
          end proc;
```

$$Maximum := \mathbf{proc}(a::integer, b::integer, c::integer) \quad \textbf{(3.14)}$$

local x;

if $b < a$ **then** $x := a$ **else** $x := b$ **end if**; **if** $x < c$ **then** c **else** x **end if**

end proc

Exécutons de nouveau la macro-commande *Maximum* ainsi modifiée.

```
> Maximum(-1,3,sqrt(2));
Error, invalid input: Maximum expects its 3rd argument, c, to
be of type integer, but received 2^(1/2)
```

La procédure a encore planté mais, cette fois, c'était prévu.

Nous sommes maintenant en mesure de réaliser la transposition informatique de l'algorithme de bissection dans le corps d'une procédure. Nous allons donné le nom **bissection** à cette procédure. Dans l'élaboration de la procédure bissection, on testera, au départ, si le théorème de la valeur intermédiaire s'applique. En fait, si la fonction change effectivement de signe aux bornes de l'intervalle considéré. Si ce n'est pas le cas, on fera en sorte que le résultat de la macro-commande soit un message d'erreur signalant à l'utilisateur que cette condition d'applicabilité de l'algorithme de bissection n'est pas satisfaite.

```
> bissection:=proc(f::procedure,intervalle::anything=range,tolérance::anything=realcons)
   local a,b,Bornes,E,Itération,m;
  Bornes:=lhs(rhs(intervalle)), rhs(rhs(intervalle));
  a:=min(Bornes);
  b:=max(Bornes);
  if not(evalf(f(a)*f(b))<0)  then
   ERROR(`La fonction ne change pas de signes aux bornes de l'intervalle considéré.`)
  fi;
  Itération:=0;
  E:=abs(rhs(tolérance));
  while evalf(abs(b-a)) > evalf(2*E) do
   m:=(a+b)/2;
   if evalf(f(m)*f(a)) < 0 then
    b:=m
   elif evalf(f(m)*f(a)) > 0 then
    a:=m
    else
    print(`Le zéro réel trouvé est exact et vaut `,m);
    return;
   fi;
  Itération:=Itération+1;
  od:
  m:= (a+b)/2;
  Itération:=Itération+1;
  print(`Le zéro réel approximatif est `,evalf(m));
  print(`Cela a nécessité `, Itération,` itérations`)
  end proc:
```

Première version de la macro-commande « bissection »

La macro-commande

bissection(f,intervalle,tolérance)

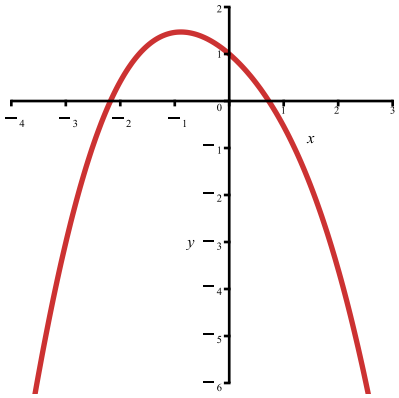
applique la méthode de bissection pour estimer un zéro réel d'une fonction réelle d'une variable réelle sur un intervalle donné.

Cette première version de la macro-commande bissection possède trois arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- intervalle: L'intervalle considéré doit être énoncé dans la forme x = a .. b
- tolérance: La précision de l'estimation doit être énoncée dans la forme E = valeur

Rappelons-nous le tracé de la fonction f.

```
> f:=x->-x^2-x+2-cos(x):
  plot([x,f(x),x=-4..3],color=orange,thickness=2,view=[-4..3,-6..2],
  xtickmarks=6);
```



Utilisons la macro-commande **bissection** pour obtenir une approximation du zéro réel de la fonction f dans l'intervalle [-3,-2] avec un écart maximal $E = 0{,}00001$.

```
> bissection(f,x=-3..-2,E=0.00001);
```

Le zéro réel approximatif est , −2.1799240110

Cela a nécessité , 17, *itérations* **(3.15)**

Obtenons maintenant une approximation de l'autre zéro compris dans l'intervalle [0,1] avec un écart maximal $E = 0{,}00000001$.

```
> bissection(f,x=0..1,E=0.00000001);
```

Le zéro réel approximatif est , 0.7254899219

Cela a nécessité , 27, *itérations* **(3.16)**

On peut donc affirmer que le second zéro est $r = 0{,}72548992$ avec un écart maximal égale à 0,00000001. Donc que $r \in [0{,}72548991 , 0{,}72548993]$.

Sachant que f(x) ne change pas de signes aux bornes de l'intervalle [-2,-1], testons si, dans ce cas, le résultat de la macro-commande **bissection** sur cet intervalle sera bien le messsage d'erreur qui a été prévu.

```
> bissection(f,x=-2..-1,E=0.00001);
Error, (in bissection) La fonction ne change pas de signes aux
bornes de l'intervalle considéré.
```

Avant la présentation de la méthode d'interpolation linéaire, peaufinons la macro-commande `bissection` en ajoutant un quatrième argument formel qui autorisera ou non, selon le cas, l'impression d'un tableau informatif. Un tel tableau présentera le détail de certains calculs jusqu'à ce l'écart maximal toléré soit atteint. De plus, dans cette seconde mouture de la procédure `bissection`, la variable d'environnement `Digits` sera localement initialisée à 40 afin de minimiser les cumuls d'erreurs dans les cas de calculs impliquant plusieurs opérations mathématiques. Compte tenu de ses capacités de mise en forme, c'est la macro-commande `prinf` plutôt que `print` qui sera utilisée. Finalement, la plus plus petite valeur de E qui sera acceptée par la procédure est $E = 1.\,10^{-15}$. (On pourra modifiée cette valeur sans peine dans la procédure mais la présentation du tableau risque d'en souffrir. Je n'ai pas déployé d'efforts de programmation pour cette mise en forme qui pourrait prendre en charge la valeur de E, car cela n'est pas pertinent dans le contexte actuel).

```
> bissection:=proc(f::procedure,intervalle::anything=range,
              tolérance::anything=realcons,tableau::name)
```

```
  local a,b,Bornes,Détail,E,Itération,k,m;
  if nargs<>4 then
 ERROR(`Le nombre d'arguments requis est 4. Vous en avez donné `||nargs||`.`)
  fi;
  if rhs(tolérance)<0.1e-14  then
   ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
  fi;
  Digits:=40;
  Bornes:=lhs(rhs(intervalle)), rhs(rhs(intervalle));
  a:=min(Bornes);
  b:=max(Bornes);
  if not(evalf(f(a)*f(b))<0)  then
   ERROR(`La fonction ne change pas de signes aux bornes l'intervalle considéré.`)
   fi;
  Itération:=0;
  E:=abs(rhs(tolérance));
  Détail:=[];
  while evalf(abs(b-a)) > evalf(2*E) do
   m:=(a+b)/2;
   Itération:=Itération+1;
   Détail:=Détail,[Itération,evalf(abs(b-a)/2),evalf((a+b)/2),evalf(f((a+b)/2))];
   if evalf(f(m)*f(a)) < 0 then
    b:=m
   elif evalf(f(m)*f(a)) > 0 then
    a:=m
   else
    print(`\nLe zéro réel trouvé est exact et vaut `,m);
    return;
   fi;
   od:
   m:=(a+b)/2;
   Itération:=Itération+1;
   Détail:=Détail,[Itération,evalf(abs(b-a)/2),evalf((a+b)/2),evalf(f((a+b)/2))];
   if tableau=non then
    printf(`\n\n%s`,`Avec la méthode de bissection,`);
    printf(`\n%s% .15g`,`le zéro réel approximatif obtenu est r = `,evalf(m));
    printf(`\n%s%d%s% 0.15f`, `Cela a nécessité `, Itération,` itérations avec E = `,E)
   else
    printf(`\n\n%s% .15f`,`Méthode de bissection avec E = `,E);
    printf(`\n%s%a`,`Intervalle initial `,intervalle);
    printf(`\n|   k  |        d[k]       |       m[k]        |       f(m[k])      | \n|======|====================|============
========|====================| \n`);
    seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15f |\n", Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=2..Itération+1)
   fi
  end:
```

Version finale de la macro-commande « bissection »

La macro-commande

bissection(f,intervalle,tolérance,tableau)

applique la méthode de bissection pour estimer un zéro réel d'une fonction réelle d'une variable réelle sur un intervalle donné.

La version finale de la macro-commande bissection possède quatre arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- intervalle: L'intervalle considéré doit être énoncé dans la forme x = a .. b
- tolérance: La précision de l'estimation doit être énoncée dans la forme E = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

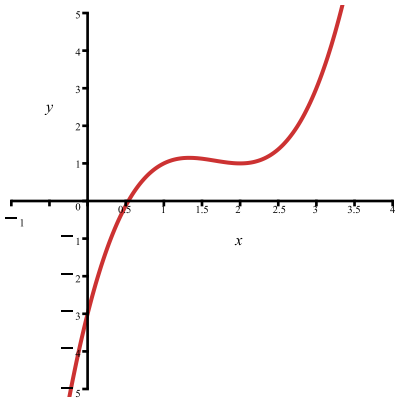
Expérimentons la version finale de la macro-commande **bissection** avec les "valeurs" *non* et *oui* respectivement sur l'intervalle [-2,5 -1,5] avec E = 0,00001.

```
> bissection(f,x=-2.5..-1.5,E=0.00001,non);


Avec la méthode de bissection,
le zéro réel approximatif obtenu est r = -2.17992401123047
Cela a nécessité 17 itérations avec E =  0.000010000000000
```

```
> bissection(f,x=-2.5..-1.5,E=0.00001,oui);


Méthode de bissection avec E =  0.000010000000000
Intervalle initial x = -2.5 .. -1.5
|   k   |         d[k]         |         m[k]         |        f(m[k])       |
|=======|======================|======================|======================|
|    1  |  0.500000000000000 | -2.000000000000000 |  0.416146836547142 |
|    2  |  0.250000000000000 | -2.250000000000000 | -0.184326377277261 |
|    3  |  0.125000000000000 | -2.125000000000000 |  0.135641334704305 |
|    4  |  0.062500000000000 | -2.187500000000000 | -0.019307050663167 |
|    5  |  0.031250000000000 | -2.156250000000000 |  0.059413495824725 |
|    6  |  0.015625000000000 | -2.171875000000000 |  0.020366396519555 |
|    7  |  0.007812500000000 | -2.179687500000000 |  0.000608162776562 |
|    8  |  0.003906250000000 | -2.183593750000000 | -0.009329797051620 |
|    9  |  0.001953125000000 | -2.181640625000000 | -0.004355908463096 |
|   10  |  0.000976562500000 | -2.180664062500000 | -0.001872646056177 |
|   11  |  0.000488281250000 | -2.180175781250000 | -0.000631934990753 |
|   12  |  0.000244140625000 | -2.179931640625000 | -0.000011809450799 |
|   13  |  0.000122070312500 | -2.179809570312500 |  0.000298195826210 |
|   14  |  0.000061035156250 | -2.179870605468750 |  0.000143197978631 |
|   15  |  0.000030517578125 | -2.179901123046875 |  0.000065695461659 |
|   16  |  0.000015258789062 | -2.179916381835938 |  0.000026943304867 |
|   17  |  0.000007629394531 | -2.179924011230469 |  0.000007567001894 |
```

Comme autre exemple, considérons de nouveau l'équation polynomiale du troisième degré $x^3 - 5x^2 + 8x - 3 = 0$ du début.

```
> f:=x->x^3-5*x^2+8*x-3:
  plot([x,f(x),x=-1..4],color=orange,view=[-1..4,-5..5]);
```



Obtenons le tableau des approximations successives à l'aide de la macro-commande `bissection` sur l'intervalle [0,1] et avec une valeur $E = 0{,}000000000000001$.

```
> bissection(f,x=0..1,E=0.1e-14,oui);
```

```
Méthode de bissection avec E =  0.000000000000001
Intervalle initial x = 0 .. 1
```

k	d[k]	m[k]	f(m[k])
1	0.500000000000000	0.500000000000000	-0.125000000000000
2	0.250000000000000	0.750000000000000	0.609375000000000
3	0.125000000000000	0.625000000000000	0.291015625000000
4	0.062500000000000	0.562500000000000	0.095947265625000
5	0.031250000000000	0.531250000000000	-0.011199951171875
6	0.015625000000000	0.546875000000000	0.043193817138672
7	0.007812500000000	0.539062500000000	0.016203403472900
8	0.003906250000000	0.535156250000000	0.002553522586823
9	0.001953125000000	0.533203125000000	-0.004310242831707
10	0.000976562500000	0.534179687500000	-0.000875120051205
11	0.000488281250000	0.534667968750000	0.000840010936372
12	0.000244140625000	0.534423828125000	-0.000017352096620
13	0.000122070312500	0.534545898437500	0.000411380029618
14	0.000061035156250	0.534484863281250	0.000197026619617
15	0.000030517578125	0.534454345703125	0.000089840424863
16	0.000015258789062	0.534439086914062	0.000036244954973
17	0.000007629394531	0.534431457519531	0.000009446626891
18	0.000003814697266	0.534427642822266	-0.000003952685436
19	0.000001907348633	0.534429550170898	0.000002746983085
20	0.000000953674316	0.534428596496582	-0.000000602848086
21	0.000000476837158	0.534429073333740	0.000001072068272
22	0.000000238418579	0.534428834915161	0.000000234610286
23	0.000000119209290	0.534428715705872	-0.000000184118852
24	0.000000059604645	0.534428775310516	0.000000025245729
25	0.000000029802322	0.534428745508194	-0.000000079436558
26	0.000000014901161	0.534428760409355	-0.000000027095414
27	0.000000007450581	0.534428767859936	-0.000000000924842
28	0.000000003725290	0.534428771585226	0.000000012160443
29	0.000000001862645	0.534428769722581	0.000000005617801
30	0.000000000931323	0.534428768791258	0.000000002346479
31	0.000000000465661	0.534428768325597	0.000000000710818
32	0.000000000232831	0.534428768092766	-0.000000000107012
33	0.000000000116415	0.534428768209182	0.000000000301903
34	0.000000000058208	0.534428768150974	0.000000000097446
35	0.000000000029104	0.534428768121870	-0.000000000004783
36	0.000000000014552	0.534428768136422	0.000000000046331
37	0.000000000007276	0.534428768129146	0.000000000020774
38	0.000000000003638	0.534428768125508	0.000000000007995
39	0.000000000001819	0.534428768123689	0.000000000001606
40	0.000000000000909	0.534428768122780	-0.000000000001589
41	0.000000000000455	0.534428768123234	0.000000000000009
42	0.000000000000227	0.534428768123007	-0.000000000000790
43	0.000000000000114	0.534428768123121	-0.000000000000391
44	0.000000000000057	0.534428768123178	-0.000000000000191
45	0.000000000000028	0.534428768123206	-0.000000000000091
46	0.000000000000014	0.534428768123220	-0.000000000000041

```
|  47  |  0.000000000000007 |  0.534428768123227 | -0.000000000000016 |
|  48  |  0.000000000000004 |  0.534428768123231 | -0.000000000000004 |
|  49  |  0.000000000000002 |  0.534428768123233 |  0.000000000000003 |
|  50  |  0.000000000000001 |  0.534428768123232 | -0.000000000000001 |
```

Comparons le résultat de la résolution de l'équation $x^3 - 5x^2 + 8x - 3 = 0$ avec celui que donne la macro-commande **fsolve**.

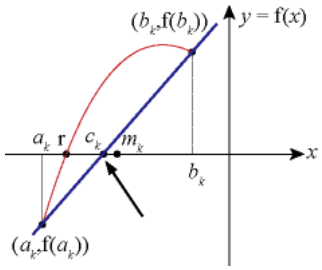
```
> bissection(f,x=0..1,E=0.1e-14,non);

Avec la méthode de bissection,
le zéro réel approximatif obtenu est r =  0.534428768123232
Cela a nécessité 50 itérations avec E =  0.000000000000001
```

```
> Digits:=15:
  fsolve(f(x)=0,{x});
  Digits:=10:
```

$$\{x = 0.5344287681\} \qquad \textbf{(3.17)}$$

Méthode d'interpolation linéaire

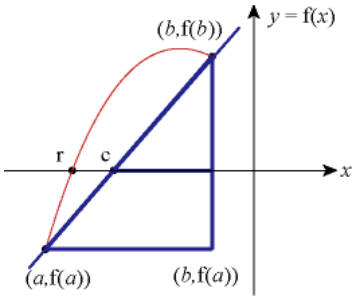
Avec la méthode de bissection, le choix systématique du milieu m_k dans chaque intervale $[a_k, b_k]$ comme valeur approchée du zéro réel de la fonction ne tient pas compte de la concavité de la fonction f au voisinage du zéro qu'on cherche à approximer. La *méthode d'interpolation linéaire* tiendra compte de cette concavité. Cette méthode permet de choisir plus "intelligemment" dans chaque intervalle $[a_k, b_k]$ une valeur approchée du zéro de la fonction. À la place de choisir systématiquement milieu de l'intervalle, choisissons plutôt systématiquement la valeur de l'abscisse du point d'intersection de l'axe des x et de la sécante à la courbe f passant par les points $(a_k, \mathrm{f}(a_k))$ et $(b_k, \mathrm{f}(b_k))$. Notons cette valeur d'abscisse par la lettre c_k.



Déterminons la formule générale pour le calcul de c_k et modifions en conséquence la procédure `bissection`.

Pour alléger la notation, utilisons les lettres a, b et c tout court au lieu de leur notation indicielle.

Soit la construction suivante.



Pour déterminer la formule nous donnant la valeur c, on tiendra compte, dans la construction précédente, des parties proportionnelles qui découlent des deux triangles semblables.

```
> f:='f':
  (b-c)/(b-a)=f(b)/(f(b)-f(a));
```

$$\frac{b-c}{b-a} = \frac{f(b)}{f(b)-f(a)} \tag{4.1}$$

Reste alors à isoler le terme c.

```
> c=solve(%,c);
```

$$c = \frac{f(b)\,a - b f(a)}{f(b)-f(a)} \tag{4.2}$$

Ce développement permettant d'obtenir *c* est appelée méthode d'interpolation linéaire.

Modifions maintenant l'algorithme de `bissection` pour obtenir une nouvelle procédure d'approximations successives qu'on appelera sécante.

```
> sécante:=proc(f::procedure,intervalle::anything=range,
            tolérance::anything=realcons,tableau::name)
   local a,b,Bornes,c,Détail,E,Itération,k;
  if nargs<>4 then
   ERROR(`Le nombre d'arguments requis est 4. Vous en avez donné `||nargs||`.`)
  fi;
  if rhs(tolérance)<0.1e-14  then
    ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
  fi;
  Digits:=40;
  Bornes:=lhs(rhs(intervalle)), rhs(rhs(intervalle));
  a:=min(Bornes);
  b:=max(Bornes);
  if not(evalf(f(a)*f(b))<0)  then
   ERROR(`La fonction ne change pas de signes aux bornes de l'intervalle considéré.`)
  fi;
  if evalf(abs(f(a))) < evalf(abs(f(b))) then
   c:=b;
   b:=a;
   a:=c;
  fi;
  c:=b;
  Itération:=0;
  E:=abs(rhs(tolérance));
```

```
 Détail:=[];
 while evalf(abs(b-a)) > evalf(2*E) do
  c:=evalf((-b*f(a)+f(b)*a)/(f(b)-f(a)));
  if evalf(f(c))=0 then
   print(`\nLe zéro réel trouvé est exact et vaut `,c);
   return
  fi;
  Itération:=Itération+1;
  Détail:=Détail,[Itération,evalf(abs(b-a)/2),c,evalf(f(c))];
  a:=b;
  b:=c
 od:
  c:=evalf((-b*f(a)+f(b)*a)/(f(b)-f(a)));
  Itération:=Itération+1;
  Détail:=Détail,[Itération,evalf(abs(b-a)/2),c,evalf(f(c))];
  if tableau=non then
   printf(`\n\n%s`,`Avec la méthode d'interpolation linéaire,`);
   printf(`\n%s% .15g`,`le zéro réel approximatif obtenu est r =`,evalf(c));
   printf(`\n%s%d%s% 0.15f`, `Cela a nécessité `, Itération,` itérations avec E = `,E)
  else
   printf(`\n\%s%.15f`,`Méthode d'interpolation linéaire avec E = `,E);
   printf(`\n%s%a`,`Intervalle initial `,intervalle);
   printf(`\n|   k  |        d[k]       |        c[k]       |       f(c[k])      | \n|======|====================|================
=====|====================| \n`);
   seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15f |\n", Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=2..Itération+1)
  fi
 end:
```

Description de la macro-commande « sécante »

La macro-commande

sécante(f,intervalle,tolérance,tableau)

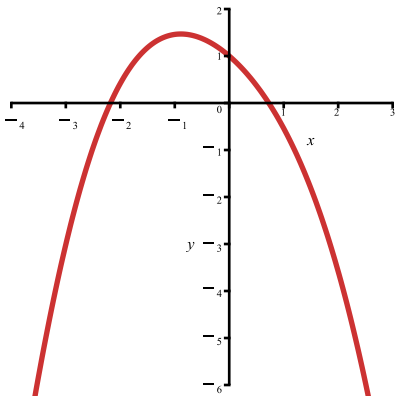
applique la méthode d'interpolation linéaire pour estimer un zéro réel d'une fonction réelle d'une variable réelle sur un intervalle donné.

La macro-commande sécante possède quatre arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- intervalle: L'intervalle considéré doit être énoncé dans la forme $x = a .. b$
- tolérance: La précision de l'estimation doit être énoncée dans la forme E = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Comparons maintenant ces deux méthodes avec l'équation $-x^2 - x + 2 - \cos(x) = 0$. Rappelons-nous encore une fois la fonction f et son tracé.

```
> f:=x->-x^2-x+2-cos(x):
  plot([x,f(x),x=-4..3],color=orange,thickness=2,view=[-4..3,-6..2],
  xtickmarks=6);
```



Comparons maintenant les deux méthodes.

```
> bissection(f,x=0..1,E=0.00001,non);
```

```
Avec la méthode de bissection,
le zéro réel approximatif obtenu est r =  0.725486755371094
Cela a nécessité 17 itérations avec E =  0.000010000000000
```

```
> sécante(f,x=0..1,E=0.00001,non);
```

```
Avec la méthode d'interpolation linéaire,
le zéro réel approximatif obtenu est r = 0.725489923996311
Cela a nécessité 6 itérations avec E =  0.000010000000000
```

L'intelligence qu'on a pensé mettre en ne choisissant pas systématiquement le milieu de l'intervalle semble être assez géniale. Il y a eu 11 itérations de moins avec la méthode d'interpolation linéaire.

Comparons aussi les deux méthodes dans la localisation de l'autre zéro réel dans l'intervalle [-2,2, -2]. Cette fois, affichons le tableau des calculs avec la valeur $E = 0{,}0000001$.

```
> bissection(f,x=-2.2..-2,E=0.0000001,oui);
  sécante(f,x=-2.2..-2,E=0.0000001,oui);
```

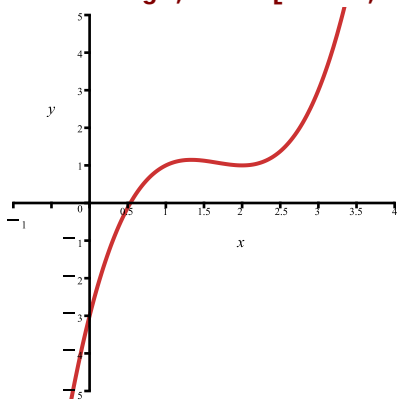
```
Méthode de bissection avec E =  0.000000100000000
Intervalle initial x = -2.2 .. -2
|   k   |          d[k]          |          m[k]          |        f(m[k])        |
|=======|========================|========================|=======================|
|    1  |  0.100000000000000 | -2.100000000000000 |  0.194846104599857 |
|    2  |  0.050000000000000 | -2.150000000000000 |  0.074857665480271 |
|    3  |  0.025000000000000 | -2.175000000000000 |  0.012481915532573 |
|    4  |  0.012500000000000 | -2.187500000000000 | -0.019307050663167 |
|    5  |  0.006250000000000 | -2.181250000000000 | -0.003362309022573 |
|    6  |  0.003125000000000 | -2.178125000000000 |  0.004572355381009 |
|    7  |  0.001562500000000 | -2.179687500000000 |  0.000608162776562 |
|    8  |  0.000781250000000 | -2.180468750000000 | -0.001376288028125 |
|    9  |  0.000390625000000 | -2.180078125000000 | -0.000383866376496 |
|   10  |  0.000195312500000 | -2.179882812500000 |  0.000112197259299 |
|   11  |  0.000097656250000 | -2.179980468750000 | -0.000135822293400 |
|   12  |  0.000048828125000 | -2.179931640625000 | -0.000011809450799 |
|   13  |  0.000024414062500 | -2.179907226562500 |  0.000050194670807 |
|   14  |  0.000012207031250 | -2.179919433593750 |  0.000019192801644 |
|   15  |  0.000006103515625 | -2.179925537109375 |  0.000003691723333 |
|   16  |  0.000003051757812 | -2.179928588867188 | -0.000004058851755 |
|   17  |  0.000001525878906 | -2.179927062988281 | -0.000000183561217 |
|   18  |  0.000000762939453 | -2.179926300048828 |  0.000001754081807 |
|   19  |  0.000000381469727 | -2.179926681518555 |  0.000000785260482 |
|   20  |  0.000000190734863 | -2.179926872253418 |  0.000000300849679 |
|   21  |  0.000000095367432 | -2.179926967620850 |  0.000000058644243 |

Méthode d'interpolation linéaire avec E = 0.000000100000000
```

```
Intervalle initial x = -2.2 .. -2
|   k   |         d[k]         |         c[k]         |        f(c[k])       |
|=======|======================|======================|======================|
|   1   |  0.100000000000000 | -2.177975257499355 |  0.004951934969341 |
|   2   |  0.011012371250323 | -2.179907294925850 |  0.000050021051313 |
|   3   |  0.000966018713247 | -2.179927010193165 | -0.000000049477017 |
|   4   |  0.000009857633657 | -2.179926990711592 |  0.000000000000493 |
|   5   |  0.000000009740786 | -2.179926990711787 |  0.000000000000000 |
```

Encore une fois, la méthode d'interpolation linéaire s'est avérée, de loin, beaucoup plus efficace que celle de bissection.

Comme autre exemple, comparons les résultats de la résolution de l'équation $x^3 - 5x^2 + 8x - 3 = 0$ sur l'intervalle [0,1] avec les deux méthodes avec $E = 0{,}0000000001$.

```
> f:=x->x^3-5*x^2+8*x-3:
  plot([x,f(x),x=-1..4],color=orange,view=[-1..4,-5..5]);
```



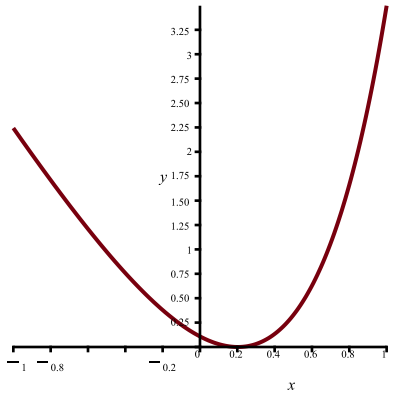

```
> bissection(f,x=0..1,E=0.1e-9,non);
  sécante(f,x=0..1,E=0.1e-9,non);


Avec la méthode de bissection,
le zéro réel approximatif obtenu est r =  0.534428768150974
Cela a nécessité 34 itérations avec E =  0.000000000100000

Avec la méthode d'interpolation linéaire,
le zéro réel approximatif obtenu est r = 0.534428768123232
Cela a nécessité 10 itérations avec E =  0.000000000100000
```

Indépendamment de leur efficacité respective, les méthodes de bissection et d'interpolation linéaire permettent d'obtenir assurément une valeur approchée du zéro d'une fonction avec, en principe, une précision E aussi grande que l'on veut, à la condition, bien sûr, que le théorème de la valeur intermédiaire puisse s'appliquer sur l'intervalle proposé. Dans le cas où le zéro d'une fonction est un extremum relatif, ces deux méthodes d'appoximations successives ne peuvent évidemment être appliquée. Dans de tels cas, ces deux méthodes sont impuissantes.

Par exemple, soit la fonction f définie par $f(x) = e^{2x} - 3x - \dfrac{3\left(1 - \ln\left(\dfrac{3}{2}\right)\right)}{2}$.

Obtenons son tracé.

```
> f:=x->exp(2*x)-3*x-3/2*(1-ln(3/2)):
  plot([x,f(x),x=-1..1],xtickmarks=12,font=[TIMES,ROMAN,8]);
```



Appliquons la méthode d'interpolation linéaire avec l'intervalle [0,199999999; 0,200000001] et avec $E = 0,0000000001$.

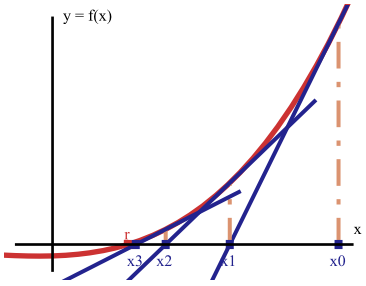
```
> sécante(f,x=0.199999999..0.200000001,E=0.0000000001,non);
Error, (in sécante) La fonction ne change pas de signes aux bornes de l'intervalle considéré.
```

La méthode de Newton-Raphson est une autre méthode d'approximations successives. Cette méthode n'est pas basée sur le théorème de la valeur intermédiaire. Elle est plus efficace et peut être utilisée également dans le cas où le zéro réel à localiser est un extremum relatif. Par contre, la méthode de Newton-Raphson possède certaines "sensibilités" qui peut la mettre en échec.

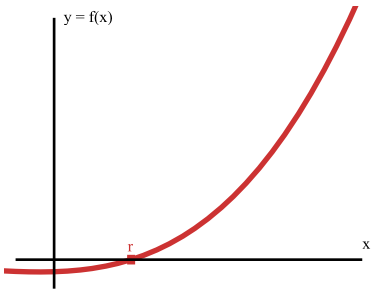
Méthode de Newton-Raphson

Les méthodes de bissection et d'interpolation linéaire propose chacune une manière respective d'obtenir, étape par étape, une valeur approchée d'un zéro réel. La méthode de Newton-Raphson propose, sous certaines conditions, une manière très différente d'obtenir, étape par étape, une valeur approchée d'un zéro réel.

La méthode de Newton-Raphson tient compte également de la concavité de la fonction. Cette méthode est basée sur l'utilisation de la tangente en un point de la courbe d'une fonction f. Plus précisément, le choix d'une première valeur x_0 approchée d'un zéro réel à localiser détermine un premier point $(x_0, \mathrm{f}(x_0))$ sur la courbe qui sera considéré comme un premier point de tangence. Ce nombre x_0 est appelé « amorce » du procédé itératif de Newton-Raphson. L'abscisse x_1 du point d'intersection de la première tangente avec l'axe des x sera considéré comme une deuxième valeur approchée du zéro à localiser. À son tour, cette valeur permettra de considérer un deuxième point de tangence $(x_1, \mathrm{f}(x_1))$. À nouveau, l'abscisse x_2 du point d'intersection de la deuxième tangente avec l'axe des x sera considéré comme une troisième valeur approchée du zéro. En poursuivant ce procédé itérativement, on obtiendra, **sous certaines conditions**, une suite de différentes valeurs $x_0, x_1, x_2, x_3, \ldots$ qui vont se rapprocher de plus en plus d'un zéro réel de la fonction f.



Sélectionner le graphique ci-dessous afin d'activer la barre de menu contextuelle dédiée aux animations. Puis cliquer sur le bouton de départ pour lancer l'animation. Cette animation illustrera dynamiquement le procédé itératif de Newton-Raphson.



Si la suite $x_1, x_2, x_3, \ldots$converge, elle converge vers un zéro réel de la fonction. Le graphique précédent avec son animation montre une situation où effectivement $\lim_{n\to\infty} x_n = r$. Dans une telle situation, pour déterminer successivement les abscisses $x_1, x_2, x_3, \ldots$ des points d'intersection des différentes tangentes avec l'axe des x, nous avons besoin de connaître d'abord l'équation de chaque tangente.

Rappelons que la forme point-pente de l'équation d'une droite passant par un point (a,b) de pente m est donnée par

$$y = \mathrm{m}\,(x - a) + \mathrm{b}$$

Si la fonction f est une fonction dérivable au point (x_k, y_k), la pente de la tangente à la courbe passant par ce point est directement donnée par $\mathrm{f}'(x_k)$. Alors, l'équation de chaque tangente à la courbe d'équation $y = \mathrm{f}(x)$ passant par le point (x_k, y_k) est donnée par

$$\mathrm{y} = \mathrm{f}'(x_k)\ (x - x_k) + y_k$$
$$\mathrm{y} = \mathrm{f}'(x_k)\ (x - x_k) + \mathrm{f}(x_k)$$

L'abscisse du point d'intersection de la *k-ième* tangente avec l'axe des x est évidemment la racine de l'équation $0 = \mathrm{f}'(x_k)\ (x - x_k) + \mathrm{f}(x_k)$. En résolvant, pour x, cette équation, on obtient

$$x = x_k - \frac{\mathrm{f}(x_k)}{\mathrm{f}'(x_k)}$$

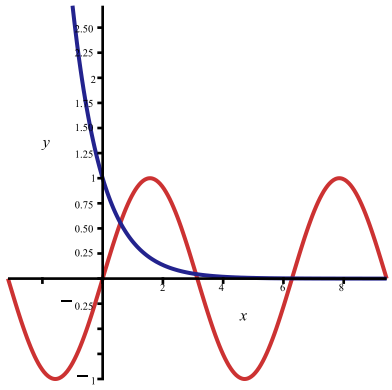
Or, la valeur de x est précisément la prochaine valeur approchée x_{k+1} qui est pris en charge dans ce procédé itératif. On déduit donc que

$$x_{k+1} = x_k - \frac{\mathrm{f}(x_k)}{\mathrm{f}'(x_k)}$$

Cette équation est appelée *équation de récurrence de Newton-Raphson*. Cette manière de procéder itérativement à l'établissement des valeurs $x_1, x_2, x_3, \ldots$est appelée méthode de Newton-Raphson. Raphson a publié cette méthode en 1690, soit presque 50 ans avant que Newton lui-même ne l'ait publiée. Bien que Newton l'ait développée en 1671, il ne l'a, quant à lui, publiée qu'en 1736. Voilà pourquoi cette méthode porte les deux noms.

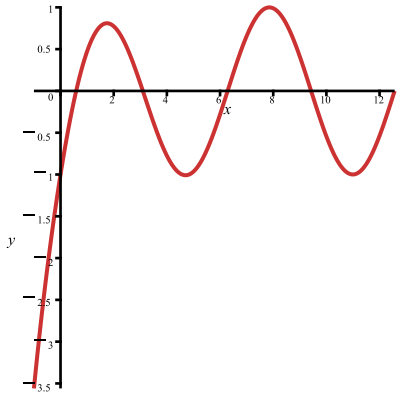
Appliquons cette formule de récurrence à la résolution de l'équation $\sin(x) = \mathrm{e}^{-x}$.

```
> Courbe_1:=plot([x,sin(x),x=-Pi..3*Pi],color=orange):
  Courbe_2:=plot([x,exp(-x),x=-1..3*Pi],color=navy):
  display([Courbe_1,Courbe_2],xtickmarks=6);
```



Puisque les fonctions $x \to \sin(x)$ et $x \to e^{-x}$ sont continues sur tous les réels $\mathbb{R}$, il est graphiquement évident que l'équation $\sin(x) = e^{-x}$ possède une infinité de racines et donc que la fonction f définie par $f(x) = \sin(x) - e^{-x}$ possède une infinité de zéros. Obtenons le tracé de la fonction f sur l'intervalle $[-1, 4\pi]$.

```
> f:=x->sin(x)-exp(-x):
  plot([x,f(x),x=-1..4*Pi],color=orange,xtickmarks=10);
```



Estimons le zéro de la fonction f dans l'intervalle [0,1]. Afin d'appliquer la formule de Newton-Raphson plus efficacement, créons une fonction "newton" définie par

$$newton(x) = x - \frac{f(x)}{f'(x)}$$

De cette manière, il appert que $x_{k+1} = newton(x_k)$.

```
> newton:=x->evalf( x - f(x)/D(f)(x) );
```

$$newton := x \mapsto evalf\left(x - \frac{f(x)}{D(f)(x)}\right) \quad \textbf{(5.1)}$$

Nous avons imbriqué la macro-commande `evalf` dans la fonction `newton` afin que l'image soit un nombre décimal.

Choisissons maintenant l'amorce $x_0 = 1$ et appliquons successivement la fonction `newton` à chaque résultat.

```
> x[0]:=1;
```

$$x_0 := 1 \quad \textbf{(5.2)}$$

```
> x[1]:=newton(x[0]);
```

$$x_1 := 0.4785277891 \quad \textbf{(5.3)}$$

```
> x[2]:=newton(x[1]);
```

$$x_2 := 0.5841570194 \quad \textbf{(5.4)}$$

```
> x[3]:=newton(x[2]);
```

$$x_3 := 0.5885251122 \quad \textbf{(5.5)}$$

```
> x[4]:=newton(x[3]);
```

$$x_4 := 0.5885327439 \quad \textbf{(5.6)}$$

```
> x[5]:=newton(x[4]);
```

$$x_5 := 0.5885327440 \quad \textbf{(5.7)}$$

```
> x[6]:=newton(x[5]);
```

$$x_6 := 0.5885327439 \quad \textbf{(5.8)}$$

On remarque que les nombres x_4 et x_5 ne diffèrent qu'à partir de la 9^e décimale. Cela semble montrer qu'effectivement les différentes valeurs x_k convergent vers un nombre particulier compris dans l'intervalle [0,1] . Et nous en connaissons donc exactement les huit premières décimales de ce nombre.

Au lieu d'exécuter ces dernières requêtes manuellement, automatisons l'exécution de ces requêtes en les incluant dans une boucle **for**.

```
> n:=5;
  x[0]:=1;
  for k from 0 to n do
   x[k+1]:=newton(x[k])
  od;
  n:='n':
```

$$n := 5$$

$$x_0 := 1$$

$$x_1 := 0.4785277891$$

$$x_2 := 0.5841570194$$

$$x_3 := 0.5885251122$$

$$x_4 := 0.5885327439$$

$$x_5 := 0.5885327440$$

$$x_6 := 0.5885327439 \quad \textbf{(5.9)}$$

Montrons que le nombre x_4 = 0,5885327439 est une bonne valeur approchée de ce zéro en évaluant f(x_4).

```
> 'f'(x[4])=f(x[4]);
```

$$f(0.5885327439) = -1.0000000000\,10^{-10} \quad \textbf{(5.10)}$$

Hum! pas si mal... Pour obtenir une meilleur valeur approchant ce zéro (meilleur dans le sens que $f(x_{k+1})$ soit davantage près de zéro), il suffit d'augmenter la valeur de la variable d'environnement Digits et d'augmenter probablement le nombres d'itérations. En initialisant la variable d'environnement `Digits` avec

une valeur plus grande que 10 et en augmentant le nombre d'itérations, nous allons, au fil des calculs itératifs, obtenir un plus grand nombre de chiffres exactes du zéro réel de la fonction f.

```
> Digits:=40:
  n:=7:
> for k from 0 to n do
    x[k+1]:=newton(x[k])
  od;
  n:='n':
```

$$x_1 := 0.4785277890$$

$$x_2 := 0.5841570194$$

$$x_3 := 0.5885251122$$

$$x_4 := 0.5885327440$$

$$x_5 := 0.5885327440$$

$$x_6 := 0.5885327440$$

$$x_7 := 0.5885327440$$

$$x_8 := 0.5885327440 \qquad \textbf{(5.11)}$$

Lorsque la variable `Digits` est initialisé à 40, les nombres x_6 et x_7 ne diffèrent qu'à partir de la 40^e décimale, on a donc obtenu 39 chiffres décimaux exacts du zéro de la fonction f.

```
> 'f'(x[6])=f(x[6]);
```

$$f(0.5885327440) = 1.0000000000\,10^{-40} \qquad \textbf{(5.12)}$$

Redonnons à la variable `Digits` sa valeur par défaut.

```
> Digits:=10;
```

$$Digits := 10 \qquad \textbf{(5.13)}$$

Avec l'équation de récurrence de Newton-Raphson formulée dans une boucle `for`, nous sommes en mesure de créer une procédure Maple permettant d'obtenir, de proche en proche, des valeurs approchant le zéro réel à localiser. Nous appellerons cette procédure `newton`.

La programmation d'un procédé itératif exige la présence d'un mécanisme d'arrêt des calculs, sinon, les calculs pourraient se poursuivre indéfiniment. Avec les macro-commandes `bissection` et `sécante`, il a suffit de contrôler le nombre d'itérations avec le calcul successif de la demi-largeur d_k de chaque intervalle $[a_k, b_k]$ encadrant le zéro réel à localiser. En effet, avec une valeur E donnée, l'inégalité $d_n = \frac{b_n - a_n}{2} < E$ est nécessairement vérifiée après un nombre fini d'étapes.

Avec la méthode de Newton-Raphson, par contre, le contrôle du nombre d'itérations ne peut être fait à l'aide d'un tel calcul car cette méthode ne génère pas une suite d'intervalles. Ce contrôle passera alors tout simplement par le calcul de l'écart $|x_{k+1} - x_k|$ entre deux valeurs approchées successives. Le procédé itératif sera donc interrompu, après n étapes, lorsque $|x_{n+1} - x_n| < E$, à condition qu'il y ait, bien sûr, convergence de

la suite des « approximations successives ».

Algorithme de la méthode de Newton-Raphson

Soit f une fonction dérivable.
Soit x_0 une première valeur proche d'un zéro de la fonction f. (x_0 est appelée amorce du processus de Newton-Raphson.)
Soit E désignant la valeur maximale de l'écart toléré, c'est-à-dire $E = |r - x_n|$.

Répéter l'étape ci-dessous pour $k = 0, 1, 2, \ldots, n,$ jusqu'à ce que $|x_{n+1} - x_n| < E$

1. $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$

Créons une première version de la procédure que nous appelerons `newton`. Modifions la procédure `secante` en conséquence pour tenir compte du critère d'arrêt des itérations de l'algorithme de Newton-Raphson.

```
> newton:=proc(f::procedure,essai::anything=realcons,
              tolérance::anything=realcons,tableau::name)
    local ancien,nouveau,Df,Détail,E,écart,Itération,k;
    if rhs(tolérance)<0.1e-14  then
      ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
     fi;
    Digits:=40;
    nouveau:=rhs(essai);
    Itération:=0;
    Détail:=[];
    Détail:=Détail,[Itération,0,nouveau,evalf(f(nouveau))];
    E:=abs(rhs(tolérance));
    écart:=2*E;
    while evalf(E) < abs(écart) do
     ancien:=nouveau;
     nouveau:=evalf(ancien-f(ancien)/D(f)(nouveau));
     Itération:=Itération+1;
     écart:=ancien-nouveau;
     Détail:=Détail,[Itération,écart,nouveau,evalf(f(nouveau))];
    od;
    if tableau=non then
     printf(`\n%s`,`Avec la méthode de Newton-Raphson,`);
     printf(`\n%s% 0.13f%s`,`le choix de l'amorce x[0] =`,rhs(essai),` a permis de trouver`),
     printf(`\n%s%0.15f%s%0.15f`,`le zéro réel approximatif r =`,evalf(nouveau), ` avec E = `,E);
     printf(`\n%s%d%s`,`Cela a nécessité `, Itération,` itérations`)
    else
     printf(`\n%s`,`Méthode de Newton-Raphson`);
     printf(`\n%s %0.13f`,`L'amorce est x[0] =`,rhs(essai)),
     printf(`\n|   k  |    x[k+1]-x[k]   |       x[k]       |      f(x[k])      | \n|======|==================|============
=======|==================| \n`);
     seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15f |\n", Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=3..Itération+2)
    fi
   end:
```

ATTENTION: En donnant le nom "**newton**" à cette procédure et en l'exécutant, la fonction "**newton**" définie

précédemment n'existe plus.

Première version de la macro-commande « `newton` »

La macro-commande

`newton(f,amorce,tolérance,tableau)`

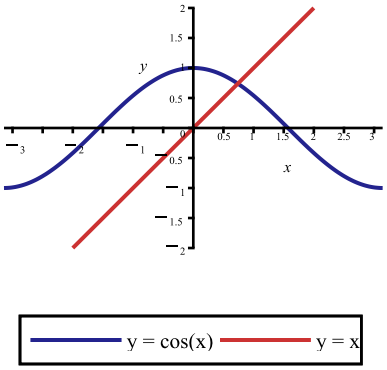
applique la méthode de Newton-Raphson pour estimer les zéros réels d'une fonction réelle d'une variable réelle.

La première version de la macro-commande newton possède quatre arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- amorce: La valeur d'amorce doit être énoncée dans la forme *amorce* = valeur
- tolérance: La précision de l'estimation doit être énoncée dans la forme E = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Appliquons la macro-commande `newton` pour trouver la ou les racines de l'équation $x = \cos(x)$.

```
> Courbe_1:=plot([x,cos(x),x=-Pi..Pi],color=navy,legend="y = cos(x)"):
  Courbe_2:=plot([x,x,x=-2..2],color=orange,legend="y = x"):
  plots[display]([Courbe_1,Courbe_2],scaling=constrained);
```



L'équation à résoudre ne possède qu'une seule racine ($x \to \cos(x)$ et $x \to x$ sont continues sur [0,1]). Utilisons la macro-commande `newton` et considérons l'amorce $x_0 = 0{,}5$ avec une tolérance $E = 0{,}0001$.

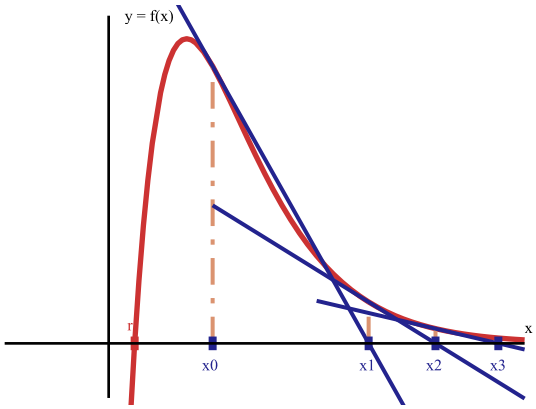
```
> f:=x->x-cos(x):
  newton(f,amorce=0.5,E=0.0001,non);

Avec la méthode de Newton-Raphson,
le choix de l'amorce x[0] = 0.5000000000000 a permis de trouver
le zéro réel approximatif r =0.739085133920807 avec E = 0.000100000000000
Cela a nécessité 3 itérations
```

Affichons le détail des calculs.

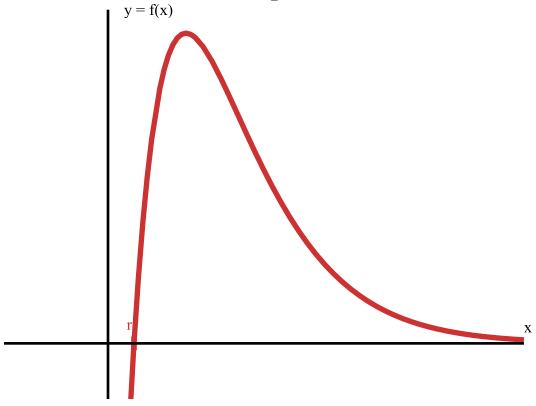
```
> newton(f,amorce=0.5,E=0.0001,oui);

Méthode de Newton-Raphson
L'amorce est x[0] = 0.5000000000000
|   k   |     x[k+1]-x[k]     |         x[k]         |       f(x[k])       |
|=======|=====================|======================|=====================|
|   1   | -0.255222417105636  |  0.755222417105636   |  0.027103311857467  |
|   2   |  0.016080750955757  |  0.739141666149879   |  0.000094615380618  |
|   3   |  0.000056532229072  |  0.739085133920807   |  0.000000001180978  |
```

La méthode de Newton-Raphson **ne garantie aucunement** que l'on va, justement, obtenir une suite de valeurs $x_0, x_1, x_2, x_3, \ldots$qui vont s'approcher aussi près que l'on voudra (selon la valeur E choisie) du zéro à localiser. Il n'y a pas de garantie de convergence. Il est possible qu'au cours du procédé itératif de Newton-Raphson, survienne un point de tangence $(x_k, \mathrm{f}(x_k))$ où la tangente est horizontale. Dans cette éventualité, le procédé itératif sera évidemment mis en échec car, dans ce cas, $\mathrm{f}'(x_k) = 0$ et donc, impossible de calculer $x_{k+1} = x_k - \dfrac{\mathrm{f}(x_k)}{\mathrm{f}'(x_k)}$. Il est donc nécessaire de tenir compte de cette éventualité dans la procédure `newton`.

Ce n'est pas seulement la possibilité d'une tangente horizontale qui peut mettre en échec la méthode de Newton-Raphson. Il y a aussi l'éventualité que la suite de valeurs $x_0, x_1, x_2, x_3, \ldots$ ne converge pas vers un zéro réel de la fonction. Le graphique ci-dessous illustre une telle situation de divergence: dans cet exemple, le choix de l'amorce x_0 amène une divergence vers l'infini.



Cliquez sur le graphique ci-dessous pour activer la barre de menu contextuelle dédiée aux animations puis cliquer sur le bouton de départ pour lancer l'animation qui illustrera cette divergence vers l'infini.



L'animation précédente illustre clairement que la suite de valeurs $x_0, x_1, x_2, x_3, \ldots$ ne converge pas vers un zéro réel de la fonction:

$$\lim_{n \to \infty} x_n = \infty .$$

Il est même possible aussi que la suite de valeurs $x_0, x_1, x_2, x_3, \ldots$ ne converge pas vers le zéro de la fonction sans pour autant qu'elle diverge vers l'infni ou vers moins l'infini. *La suite* $x_0, x_1, x_2, x_3, \ldots$ *pourrait cycler* et donc diverger par oscillation.

Avant de présenter une telle situation de divergence, peaufinons la macro-commande `newton` afin que le

corps de la procédure puisse prévoir ces éventualités. Dans le mécanisme d'arrêt du procédé itératif, il ne faut donc pas uniquement se baser sur le calcul de l'écart $|x_{k+1} - x_k|$ entre deux approximations successives. Il faut prévoir la possibilité de la division par 0 ($f'(x_k) = 0$) et, en plus, limiter le nombre d'itération dans le cas où la séquences de valeurs x_0, x_1, x_2, x_3, … ne convergerait pas vers un zéro réel de la fonction.

```
> newton:=proc(f::procedure,essai::anything=realcons,maxiter::anything=posint,
            tolérance::anything=realcons,tableau::name)
   local ancien,nouveau,Df,Détail,E,écart,Itération,k;
   if nargs<>5 then
    ERROR(`Le nombre d'arguments requis est 5. Vous en avez donné `||nargs||`.`)
   fi;
   if rhs(tolérance)<0.1e-14  then
      ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
     fi;
   Digits:=40;
   Itération:=0;
   Détail:=[];
   nouveau:=evalf(rhs(essai));
   Détail:=Détail,[Itération,0,nouveau,evalf(f(nouveau))];
   E:=abs(rhs(tolérance));
   écart:=E;
   for k to rhs(maxiter) while evalf(E) <= abs(écart) do
    ancien:=evalf(nouveau);
    if evalf(f(nouveau))=0 then
     printf(`\n%s%g`,`Le zéro réel trouvé est exact et vaut `,nouveau);
     return
    fi;
    if evalf(D(f)(nouveau))=0 then
     printf(`\n\n%s%d%s`,`La méthode de Newton-Raphson a échoué à la `,Itération,`ième itération(s).`);
     return
    fi;
    nouveau:=evalf(ancien-f(ancien)/D(f)(nouveau));
    Itération:=Itération+1;
    écart:=ancien-nouveau;
    Détail:=Détail,[Itération,écart,nouveau,evalf(f(nouveau))];
   od;
   if abs(écart) < E and tableau=non then
    printf(`\n\nLa méthode de Newton-Raphson semble donner une convergence.`);
    printf(`\n%s% 0.13f%s`,`Le choix de l'amorce x[0] =`, rhs(essai) ,` a permis de trouver`),
    printf(`\n%s% 0.15f%s% 0.15f`,`le zéro réel approximatif r =`,evalf(nouveau),` avec E = `,E);
    printf(`\n%s%d%s`, `Cela a nécessité `, Itération,` itérations.`)
   elif abs(écart) < E and tableau=oui then
    printf(`\n\n%s`,`Méthode de Newton-Raphson`);
    printf(`\n%s % 0.13f`,`Le choix de l'amorce est x[0] =`,rhs(essai)),
    printf(`\n|   k  |    x[k+1]-x[k]   |       x[k]      |       f(x[k])       | \n|======|====================|==========
=========|======================| \n`);
   seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15E |\n",Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=3..Itération+2);
   printf(`\nLa méthode de Newton-Raphson semble donner un convergence.`);
   printf(`\n%s% 0.15f%s% 0.15f`,`Le zéro réel approximatif est r =`,nouveau,` avec E = `,E);
   elif tableau=non then
    printf(`\n\n%s`,`La méthode de Newton-Raphson semble avoir échoué.`);
```

```
    printf(`\n\n%s%d%s`, `Après `,Itération, ` itération(s), les approximations successives `);
    printf(`\n%s`,`semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.`);
    printf(`\n\n%s`,`Consulter le tableau des calculs pour déterminer s'il est pertinent`):      printf(`\n%s`,`d'essayer une
  autre valeur d'amorce plus près du zéro réel à`):
    printf(`\n%s`,`localiser et/ou d'augmenter le nombre d'itérations.`);
   else
    printf(`\n\n%s`,`La méthode de Newton-Raphson semble avoir échoué.`);
    printf(`\n|   k  |    x[k+1]-x[k]   |       x[k]       |        f(x[k])        | \n|======|==================|===========
  ========|======================| \n`);
    seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15E |\n",Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=3..Itération+2);
   printf(`\n%s%d%s`, `Après `,Itération, ` itération(s), les approximations successives`);
   printf(`\n%s`,`semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.`);
   printf(`\n\n%s`,`Essayer, s'il y a lieu, une autre valeur d'amorce plus près `):
   printf(`\n%s`,`du zéro réel à localiser et/ou augmenter le nombre d'itérations.`);
   fi
  end:
```

```
> newton(f,amorce=0.5,n=10,E=0.00001,oui);
```

```
Méthode de Newton-Raphson
Le choix de l'amorce est x[0] =  0.5000000000000
|   k   |     x[k+1]-x[k]    |         x[k]        |         f(x[k])          |
|======|====================|=====================|==========================|
|   1   | -0.255222417105636 |  0.755222417105636  |  2.710331185746727E-02   |
|   2   |  0.016080750955757 |  0.739141666149879  |  9.461538061775639E-05   |
|   3   |  0.000056532229072 |  0.739085133920807  |  1.180977904622197E-09   |
|   4   |  0.000000000705646 |  0.739085133215161  |  1.840087340814741E-19   |

La méthode de Newton-Raphson semble donner un convergence.
Le zéro réel approximatif est r = 0.739085133215161 avec E =  0.000010000000000
```

Version finale de la macro-commande « newton »

La macro-commande

newton(f,amorce,maxiter,tolérance,tableau)

applique la méthode de Newton-Raphson pour estimer les zéros réels d'une fonction réelle d'une variable réelle.

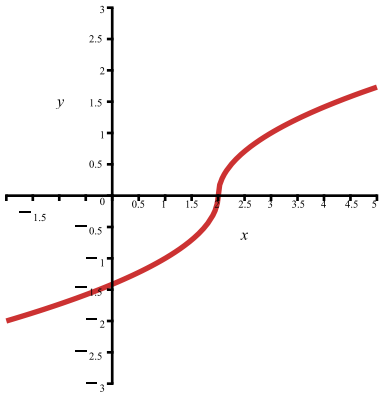
La version finale de la macro-commande newton possède cinq arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- amorce: La valeur d'amorce doit être énoncée dans la forme *amorce* = valeur
- maxiter: Le nombre maximum d'itérations doit être donné dans la forme *n* = nombre
- tolérance: La précision de l'estimation doit être énoncée dans la forme *E* = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Soit la fonction f définie par $f(x) = piecewise\left(x < 2, -\sqrt{2-x}, 2 \le x, \sqrt{x-2}\right)$. La méthode de Newton-Raphson appliquée à la fonction f est mise en échec et ce, quelque soit la valeur d'amorce x_0: la séquence d'approximations successives diverge par oscillation.

```
> f:=x->piecewise(x<2,-sqrt(2-x),x>=2,sqrt(x-2));
  plot([x,f(x),x=-2..5],thickness=2,color=orange,numpoints=80,view=[-2..5,-3.
  .3]);
```

$$f := x \mapsto \begin{cases} -\sqrt{2-x} & x < 2 \\ \sqrt{x-2} & 2 \le x \end{cases}$$

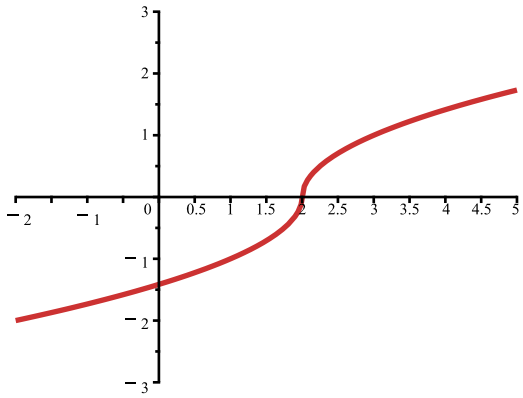



```
> newton(f,amorce=0.5,n=10,E=0.00001,non);


La méthode de Newton-Raphson semble avoir échoué.

Après 10 itération(s), les approximations successives
semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.

Consulter le tableau des calculs pour déterminer s'il est pertinent
d'essayer une autre valeur d'amorce plus près du zéro réel à
localiser et/ou d'augmenter le nombre d'itérations.
```

```
> newton(f,amorce=0.5,n=10,E=0.00001,oui);


La méthode de Newton-Raphson semble avoir échoué.
|   k   |     x[k+1]-x[k]     |         x[k]         |          f(x[k])          |
|=======|======================|======================|===========================|
|    1  | -3.000000000000000 |  3.500000000000000 |  1.224744871391589E+00 |
|    2  |  3.000000000000000 |  0.500000000000000 | -1.224744871391589E+00 |
|    3  | -3.000000000000000 |  3.500000000000000 |  1.224744871391589E+00 |
|    4  |  3.000000000000000 |  0.500000000000000 | -1.224744871391589E+00 |
|    5  | -3.000000000000000 |  3.500000000000000 |  1.224744871391589E+00 |
|    6  |  3.000000000000000 |  0.500000000000000 | -1.224744871391589E+00 |
|    7  | -3.000000000000000 |  3.500000000000000 |  1.224744871391589E+00 |
|    8  |  3.000000000000000 |  0.500000000000000 | -1.224744871391589E+00 |
|    9  | -3.000000000000000 |  3.500000000000000 |  1.224744871391589E+00 |
|   10  |  3.000000000000000 |  0.500000000000000 | -1.224744871391589E+00 |

Après 10 itération(s), les approximations successives
semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.

Essayer, s'il y a lieu, une autre valeur d'amorce plus près
du zéro réel à localiser et/ou augmenter le nombre d'itérations.
```

Dans ce cas-ci, même si on augmentait le nombre d'térations, la méthode de Newton-Raphson serait encore mise en échec.

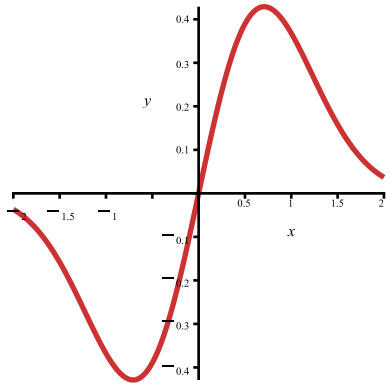
Cliquez sur le graphique ci-dessous pour activer la barre de menu contextuelle dédiée aux animations puis cliquer sur le bouton de départ pour lancer l'animation. On constatera qu'effectivement la méthode de Newton-Raphson diverge par oscillation. La valeur d'amorce utilisée a été $x_0 = 0{,}5$.



Le prochain exemple va montrer la "sensiblité" du procédé itératif de Newton-Raphson face au choix de la valeur d'amorce x_0. Dans cet exemple, on verra qu'en choisissant $x_0 = 0{,}52$, la méthode de Newton-Raphson est mise en échec tandis qu'avec le choix de $x_0 = 0{,}48$, la méthode fonctionne.

```
> f:=x->x*exp(-x^2);
  plot([x,f(x),x=-2..2],color=orange,thickness=2);
```

$$f := x \mapsto x \cdot e^{-x^2}$$



Soit $x_0 = 0.52$ la valeur d'amorce proposée.

```
> newton(f,amorce=0.52,n=20,E=0.00001,non);


La méthode de Newton-Raphson semble avoir échoué.

Après 20 itération(s), les approximations successives
semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.

Consulter le tableau des calculs pour déterminer s'il est pertinent
d'essayer une autre valeur d'amorce plus près du zéro réel à
localiser et/ou d'augmenter le nombre d'itérations.
```

Consultons le tableau des calculs.

```
> newton(f,amorce=0.52,n=20,E=0.00001,oui);


La méthode de Newton-Raphson semble avoir échoué.
|   k   |     x[k+1]-x[k]     |        x[k]         |         f(x[k])         |
|=======|=====================|=====================|=========================|
|   1   |  1.132404181184669  | -0.612404181184669  | -4.208824633100418E-01  |
|   2   | -2.450378912795456  |  1.837974731610787  |  6.269416875604370E-02  |
|   3   | -0.319297816338598  |  2.157272547949385  |  2.054823701772557E-02  |
```

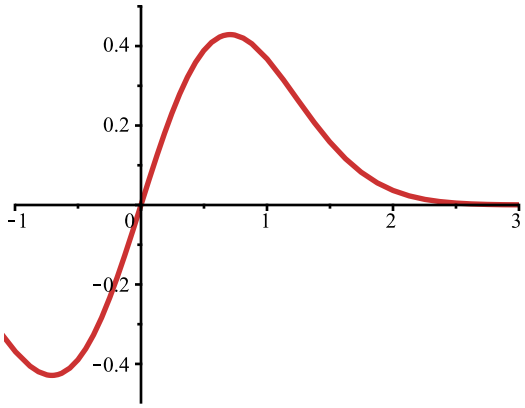
```
|   4  | -0.259673027614070 |  2.416945575563455 |  7.019092582029569E-03 |
|   5  | -0.226236881234015 |  2.643182456797470 |  2.443241023609862E-03 |
|   6  | -0.203747606965644 |  2.846930063763114 |  8.598354322710446E-04 |
|   7  | -0.187174623622555 |  3.034104687385669 |  3.047938629773795E-04 |
|   8  | -0.174257835896659 |  3.208362523282328 |  1.086010598643007E-04 |
|   9  | -0.163799102107172 |  3.372161625389500 |  3.884543244100688E-05 |
|  10  | -0.155092198941101 |  3.527253824330601 |  1.393649071379328E-05 |
|  11  | -0.147688651019029 |  3.674942475349630 |  5.012054146821964E-06 |
|  12  | -0.141287402519622 |  3.816229877869252 |  1.806087799977042E-06 |
|  13  | -0.135677459322946 |  3.951907337192198 |  6.519009073192055E-07 |
|  14  | -0.130705758877762 |  4.082613096069960 |  2.356325011638208E-07 |
|  15  | -0.126258082420783 |  4.208871178490743 |  8.527349871516074E-08 |
|  16  | -0.122247162596666 |  4.331118341087409 |  3.089225602745633E-08 |
|  17  | -0.118604976339413 |  4.449723317426822 |  1.120175154119473E-08 |
|  18  | -0.115277581558855 |  4.565000898985678 |  4.065152835424214E-09 |
|  19  | -0.112221558511077 |  4.677222457496755 |  1.476331599406062E-09 |
|  20  | -0.109401496221224 |  4.786623953717979 |  5.365059182420669E-10 |

Après 20 itération(s), les approximations successives
semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.

Essayer, s'il y a lieu, une autre valeur d'amorce plus près
du zéro réel à localiser et/ou augmenter le nombre d'itérations.
```

Le tableau semble révéler que le choix de $x_0 = 0{,}52$ amène une séquence de valeurs x_k divergentes bien qu'il semble que $f(x) \to 0$. Augmentons le nombre d'itérations "pour voir".

```
> newton(f,amorce=0.52,n=60,E=0.00001,oui);


La méthode de Newton-Raphson semble avoir échoué.
|   k  |     x[k+1]-x[k]    |         x[k]        |          f(x[k])         |
|======|====================|=====================|==========================|
|   1  |  1.132404181184669 | -0.612404181184669 | -4.208824633100418E-01 |
|   2  | -2.450378912795456 |  1.837974731610787 |  6.269416875604370E-02 |
|   3  | -0.319297816338598 |  2.157272547949385 |  2.054823701772557E-02 |
|   4  | -0.259673027614070 |  2.416945575563455 |  7.019092582029569E-03 |
|   5  | -0.226236881234015 |  2.643182456797470 |  2.443241023609862E-03 |
|   6  | -0.203747606965644 |  2.846930063763114 |  8.598354322710446E-04 |
|   7  | -0.187174623622555 |  3.034104687385669 |  3.047938629773795E-04 |
|   8  | -0.174257835896659 |  3.208362523282328 |  1.086010598643007E-04 |
|   9  | -0.163799102107172 |  3.372161625389500 |  3.884543244100688E-05 |
|  10  | -0.155092198941101 |  3.527253824330601 |  1.393649071379328E-05 |
|  11  | -0.147688651019029 |  3.674942475349630 |  5.012054146821964E-06 |
|  12  | -0.141287402519622 |  3.816229877869252 |  1.806087799977042E-06 |
|  13  | -0.135677459322946 |  3.951907337192198 |  6.519009073192055E-07 |
|  14  | -0.130705758877762 |  4.082613096069960 |  2.356325011638208E-07 |
|  15  | -0.126258082420783 |  4.208871178490743 |  8.527349871516074E-08 |
|  16  | -0.122247162596666 |  4.331118341087409 |  3.089225602745633E-08 |
|  17  | -0.118604976339413 |  4.449723317426822 |  1.120175154119473E-08 |
|  18  | -0.115277581558855 |  4.565000898985678 |  4.065152835424214E-09 |
|  19  | -0.112221558511077 |  4.677222457496755 |  1.476331599406062E-09 |
|  20  | -0.109401496221224 |  4.786623953717979 |  5.365059182420669E-10 |
|  21  | -0.106788178580707 |  4.893412132298685 |  1.950836921732766E-10 |
|  22  | -0.104357250373633 |  4.997769382672318 |  7.097402769306198E-11 |
|  23  | -0.102088219628453 |  5.099857602300771 |  2.583386350892081E-11 |
|  24  | -0.099963700198069 |  5.199821302498840 |  9.407467587598235E-12 |
|  25  | -0.097968828887871 |  5.297790131386711 |  3.427155500510715E-12 |
|  26  | -0.096090811379261 |  5.393880942765972 |  1.248989274089403E-12 |
|  27  | -0.094318564526427 |  5.488199507292398 |  4.553392776285739E-13 |
|  28  | -0.092642431689816 |  5.580841938982214 |  1.660550395808329E-13 |
|  29  | -0.091053954068989 |  5.671895893051203 |  6.057588083467174E-14 |
|  30  | -0.089545685433508 |  5.761441578484711 |  2.210392180578242E-14 |
|  31  | -0.088111040819299 |  5.849552619304010 |  8.067758865598231E-15 |
|  32  | -0.086744172051496 |  5.936296791355506 |  2.945393142239037E-15 |
|  33  | -0.085439864635133 |  6.021736655990639 |  1.075558153710196E-15 |
|  34  | -0.084193451800155 |  6.105930107790795 |  3.928428426570700E-16 |
|  35  | -0.083000742419381 |  6.188930850210175 |  1.435134781083932E-16 |
|  36  | -0.081857960222931 |  6.270788810433106 |  5.243852455647148E-17 |
|  37  | -0.080761692270354 |  6.351550502703460 |  1.916406368653396E-17 |
|  38  | -0.079708845055539 |  6.431259347758999 |  7.004867396995561E-18 |
|  39  | -0.078696606940469 |  6.509955954699468 |  2.560846408515092E-18 |
|  40  | -0.077722415864772 |  6.587678370564240 |  9.363427819818240E-19 |
|  41  | -0.076783931475456 |  6.664462302039696 |  3.424132883953012E-19 |
```

```
|  42  | -0.075879010977721 |  6.740341313017417 |  1.252355670837927E-19 |
|  43  | -0.075005688132458 |  6.815347001149875 |  4.581031059328100E-20 |
|  44  | -0.074162154926178 |  6.889509156076053 |  1.675925065379068E-20 |
|  45  | -0.073346745519811 |  6.962855901595864 |  6.131959567870347E-21 |
|  46  | -0.072557922148334 |  7.035413823744198 |  2.243855953915368E-21 |
|  47  | -0.071794262696574 |  7.107208086440772 |  8.211821433989183E-22 |
|  48  | -0.071054449720253 |  7.178262536161025 |  3.005597474419474E-22 |
|  49  | -0.070337260717363 |  7.248599796878388 |  1.100188302574742E-22 |
|  50  | -0.069641559484704 |  7.318241356363092 |  4.027599883464495E-23 |
|  51  | -0.068966288419143 |  7.387207644782235 |  1.474575511225216E-23 |
|  52  | -0.068310461643722 |  7.455518106425957 |  5.399176490910951E-24 |
|  53  | -0.067673158856003 |  7.523191265281959 |  1.977089544270664E-24 |
|  54  | -0.067053519810499 |  7.590244785092459 |  7.240391424659438E-25 |
|  55  | -0.066450739359246 |  7.656695524451705 |  2.651754423890896E-25 |
|  56  | -0.065864062984884 |  7.722559587436589 |  9.712674656646382E-26 |
|  57  | -0.065292782769385 |  7.787852370205974 |  3.557766673118480E-26 |
|  58  | -0.064736233749010 |  7.852588603954984 |  1.303310897234678E-26 |
|  59  | -0.064193790612443 |  7.916782394567427 |  4.774736971311116E-27 |
|  60  | -0.063664864704508 |  7.980447259271935 |  1.749366128853753E-27 |

Après 60 itération(s), les approximations successives
semblent diverger, soit par oscillation ou soit vers + l'infini ou vers - l'infini.

Essayer, s'il y a lieu, une autre valeur d'amorce plus près
du zéro réel à localiser et/ou augmenter le nombre d'itérations.
```

Le procédé de Newton-Raphson est donc vraisemblablement mis en échec dans ce cas. En effet, exécutez l'animation suivante qui montre le comportement des six premières approximations successives avec $x_0 = 0{,}52$.



Choisissons maintenant $x_0 = 0{,}48$ comme valeur d'amorce.

```
> newton(f,amorce=0.48,n=10,E=0.00001,non);


La méthode de Newton-Raphson semble donner une convergence.
Le choix de l'amorce x[0] = 0.4800000000000 a permis de trouver
le zéro réel approximatif r = 0.000000000000000 avec E =  0.000010000000000
Cela a nécessité 6 itérations.
```

Le procédé de Newton-Raphson dans ce cas donne une convergenve. Ces deux derniers développements montrent que la méthode de Newton-Raphson est parfois sensible au choix de la valeur d'amorce x_0.

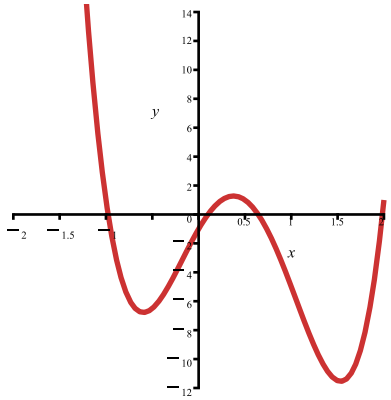
La méthode de Newton-Raphson est plutôt efficace en terme de vitesse de convergence mais il n'y a pas de garantie de convergence. Et, lorsqu'il y a convergence, ce n'est pas nécessairement vers la valeur à laquelle on pense. L'exemple suivant va illustrer une telle situation.

Soit la fonction f définie par $f(x) = 8x^4 - 14x^3 - 9x^2 + 11x - 1$.

```
> f:=x->8*x^4-14*x^3-9*x^2+11*x-1;
  plot([x,f(x),x=-2..2],color=orange,thickness=2,numpoints=80,
```

```
xtickmarks=12,view=[-2..2,-12..14]);
```

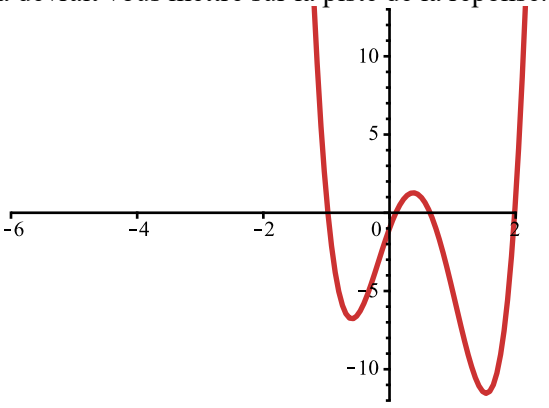
$$f := x \mapsto 8 \cdot x^4 - 14 \cdot x^3 - 9 \cdot x^2 + 11 \cdot x - 1$$



```
> newton(f,amorce=0.37,n=20,E=0.00001,oui);


Méthode de Newton-Raphson
Le choix de l'amorce est x[0] =  0.3700000000000
|   k   |     x[k+1]-x[k]     |         x[k]        |         f(x[k])          |
|======|=====================|=====================|==========================|
|   1   |  6.057390381627316 | -5.687390381627316 |  1.059117580997774E+04 |
|   2   | -1.484999108572100 | -4.202391273055217 |  3.327868431511312E+03 |
|   3   | -1.098325036777342 | -3.104066236277875 |  1.039556420040468E+03 |
|   4   | -0.802823998663243 | -2.301242237614632 |  3.209958920500422E+02 |
|   5   | -0.573234686886415 | -1.728007550728217 |  9.668574828000756E+01 |
|   6   | -0.389197390466775 | -1.338810160261441 |  2.743912280729513E+01 |
|   7   | -0.234576236242642 | -1.104233924018799 |  6.623606187415767E+00 |
|   8   | -0.104437924912860 | -0.999795999105940 |  9.908233303023140E-01 |
|   9   | -0.022034475606762 | -0.977761523499177 |  3.882314129580868E-02 |
|  10   | -0.000936281087436 | -0.976825242411742 |  6.829977424734223E-05 |
|  11   | -0.000001652970842 | -0.976823589440900 |  2.126492995648971E-10 |

La méthode de Newton-Raphson semble donner un convergence.
Le zéro réel approximatif est r =-0.976823589440900 avec E =  0.000010000000000
```

On constate donc que le zéro réel trouvé, soit r $\approx -0{,}97682$, n'est pas l'un des deux situés à proximité de l'amorce $x_0 = 0{,}37$. Pouvez-vous expliquer pourquoi ?

Lancez l'animation suivante. Cela devrait vous mettre sur la piste de la réponse.



Résumé des macro-commandes d'approximations successives

Macro-commande bissection

Initialisation

```
> bissection:=proc(f::procedure,intervalle::anything=range,
                   tolérance::anything=realcons,tableau::name)
    local a,b,Bornes,Détail,E,Itération,k,m;
    if nargs<>4 then
   ERROR(`Le nombre d'arguments requis est 4. Vous en avez donné `||nargs||`.`)
    fi;
    if rhs(tolérance)<0.1e-14  then
     ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
    fi;
    Digits:=40;
    Bornes:=lhs(rhs(intervalle)), rhs(rhs(intervalle));
    a:=min(Bornes);
    b:=max(Bornes);
    if not(evalf(f(a)*f(b))<0)  then
     ERROR(`La fonction ne change pas de signes aux bornes l'intervalle considéré.`)
     fi;
    Itération:=0;
    E:=abs(rhs(tolérance));
    Détail:=[];
    while evalf(abs(b-a)) > evalf(2*E) do
     m:=(a+b)/2;
     Itération:=Itération+1;
     Détail:=Détail,[Itération,evalf(abs(b-a)/2),evalf((a+b)/2),evalf(f((a+b)/2))];
     if evalf(f(m)*f(a)) < 0 then
      b:=m
     elif evalf(f(m)*f(a)) > 0 then
      a:=m
     else
      print(`\nLe zéro réel trouvé est exact et vaut `,m);
      return;
     fi;
     od:
     m:=(a+b)/2;
     Itération:=Itération+1;
     Détail:=Détail,[Itération,evalf(abs(b-a)/2),evalf((a+b)/2),evalf(f((a+b)/2))];
     if tableau=non then
      printf(`\n\n%s`,`Avec la méthode de bissection,`);
      printf(`\n%s% .15g`,`le zéro réel approximatif obtenu est r = `,evalf(m));
      printf(`\n%s%d%s% 0.15f`, `Cela a nécessité `, Itération,` itérations avec E = `,E)
     else
      printf(`\n%s% .15f`,`Méthode de bissection avec E = `,E);
      printf(`\n%s%a`,`Intervalle initial `,intervalle);
      printf(`\n|   k  |        d[k]       |       m[k]        |      f(m[k])      | \n|======|====================|======
==============|====================| \n`);
      seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15f |\n", Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=2..
Itération+1)
     fi
    end:
```

La macro-commande

bissection(f,intervalle,tolérance,tableau)

applique la méthode de bissection pour estimer un zéro réel d'une fonction réelle d'une variable réelle sur un intervalle donné.

La version finale de la macro-commande bissection possède quatre arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- intervalle: L'intervalle considéré doit être énoncé dans la forme x = a .. b
- tolérance: La précision de l'estimation doit être énoncée dans la forme E = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Macro-commande sécante

Initialisation

```
> sécante:=proc(f::procedure,intervalle::anything=range,
              tolérance::anything=realcons,tableau::name)
   local a,b,Bornes,c,Détail,E,Itération,k;
  if nargs<>4 then
   ERROR(`Le nombre d'arguments requis est 4. Vous en avez donné `||nargs||`.`)
  fi;
  if rhs(tolérance)<0.1e-14  then
    ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
  fi;
  Digits:=40;
  Bornes:=lhs(rhs(intervalle)), rhs(rhs(intervalle));
  a:=min(Bornes);
  b:=max(Bornes);
  if not(evalf(f(a)*f(b))<0)  then
   ERROR(`La fonction ne change pas de signes aux bornes de l'intervalle considéré.`)
  fi;
  if evalf(abs(f(a))) < evalf(abs(f(b))) then
   c:=b;
   b:=a;
   a:=c;
  fi;
  c:=b;
  Itération:=0;
  E:=abs(rhs(tolérance));
  Détail:=[];
  while evalf(abs(b-a)) > evalf(2*E) do
   c:=evalf((-b*f(a)+f(b)*a)/(f(b)-f(a)));
   if evalf(f(c))=0 then
    print(`\nLe zéro réel trouvé est exact et vaut `,c);
    return
   fi;
   Itération:=Itération+1;
   Détail:=Détail,[Itération,evalf(abs(b-a)/2),c,evalf(f(c))];
   a:=b;
```

```
    b:=c
   od:
    c:=evalf((-b*f(a)+f(b)*a)/(f(b)-f(a)));
    Itération:=Itération+1;
    Détail:=Détail,[Itération,evalf(abs(b-a)/2),c,evalf(f(c))];
    if tableau=non then
     printf(`\n\n%s`,`Avec la méthode d'interpolation linéaire,`);
     printf(`\n%s% .15g`,`le zéro réel approximatif obtenu est r =`,evalf(c));
     printf(`\n%s%d%s% 0.15f`, `Cela a nécessité `, Itération,` itérations avec E = `,E,`\n`)
    else
     printf(`\n%s%.15f`,`Méthode d'interpolation linéaire avec E = `,E);
     printf(`\n%s%a`,`Intervalle initial `,intervalle);
     printf(`\n|   k  |       d[k]      |       c[k]       |      f(c[k])      | \n|======|====================|=========
==========|====================| \n`);
     seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15f |\n", Détail[k,1],Détail[k,2],Détail[k,3],Détail[k,4]),k=2..
Itération+1)
    fi
   end:
```

La macro-commande

sécante(f,intervalle,tolérance,tableau)

applique la méthode d'interpolation linéaire pour estimer un zéro réel d'une fonction réelle d'une variable réelle sur un intervalle donné.

La macro-commande sécante possède quatre arguments:

- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- intervalle: L'intervalle considéré doit être énoncé dans la forme *x* = a .. b
- tolérance: La précision de l'estimation doit être énoncée dans la forme *E* = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Macro-commande newton

Initialisation

```
> newton:=proc(f::procedure,essai::anything=realcons,maxiter::anything=
  posint,
                  tolérance::anything=realcons,tableau::name)
    local ancien,nouveau,Df,Détail,E,écart,Itération,k;
    if nargs<>5 then
     ERROR(`Le nombre d'arguments requis est 5. Vous en avez donné
  `||nargs||`.`)
    fi;
    if rhs(tolérance)<0.1e-14  then
        ERROR(`La valeur minimale E acceptée est 0.1e-14).`)
       fi;
    Digits:=40;
    Itération:=0;
    Détail:=[];
    nouveau:=evalf(rhs(essai));
```

```
  Détail:=Détail,[Itération,0,nouveau,evalf(f(nouveau))];
  E:=abs(rhs(tolérance));
  écart:=E;
  for k to rhs(maxiter) while evalf(E) <= abs(écart) do
   ancien:=evalf(nouveau);
   if evalf(f(nouveau))=0 then
    printf(`\n%s%g`,`Le zéro réel trouvé est exact et vaut `,nouveau);
    return
   fi;
   if evalf(D(f)(nouveau))=0 then
    printf(`\n\n%s%d%s`,`La méthode de Newton-Raphson a échoué à la `,
Itération,`ième itération(s).`);
    return
   fi;
   nouveau:=evalf(ancien-f(ancien)/D(f)(nouveau));
   Itération:=Itération+1;
   écart:=ancien-nouveau;
   Détail:=Détail,[Itération,écart,nouveau,evalf(f(nouveau))];
  od;
  if abs(écart) < E and tableau=non then
   printf(`\n\nLa méthode de Newton-Raphson semble donner une
convergence.`);
   printf(`\n%s% 0.13f%s`,`L'amorce x[0] =`, rhs(essai) ,` a permis de
trouver`),
   printf(`\n%s% 0.15f%s% 0.15f`,`le zéro réel approximatif r =`,evalf
(nouveau),` avec E = `,E);
   printf(`\n%s%d%s`, `Cela a nécessité `, Itération,` itérations.`)
  elif abs(écart) < E and tableau=oui then
   printf(`\n%s`,`Méthode de Newton-Raphson`);
   printf(`\n%s % 0.13f`,`La valeur de l'amorce est x[0] =`,rhs(essai)
),
   printf(`\n|   k  |     x[k+1]-x[k]    |         x[k]       |
f(x[k])        | \n|======|====================|====================|==
======================| \n`);
  seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15E |\n",Détail[k,1],
Détail[k,2],Détail[k,3],Détail[k,4]),k=3..Itération+2);
  printf(`\nLa méthode de Newton-Raphson semble donner un convergence.
`);
  printf(`\n%s% 0.15f%s% 0.15f`,`Le zéro réel approximatif est r =`,
nouveau,` avec E = `,E);
  elif tableau=non then
   printf(`\n\n%s`,`La méthode de Newton-Raphson semble avoir échoué.`)
;
   printf(`\n\n%s%d%s`, `Après `,Itération, ` itération(s), les
approximations successives `);
   printf(`\n%s`,`semblent diverger, soit par oscillation ou soit vers
+ l'infini ou vers - l'infini.`);
   printf(`\n\n%s`,`Consulter le tableau des calculs pour déterminer
```

```
s'il est pertinent`):      printf(`\n%s`,`d'essayer une autre valeur
d'amorce plus près du zéro réel à`):
   printf(`\n%s`,`localiser et/ou d'augmenter le nombre d'itérations.`)
;
  else
   printf(`\n\n%s`,`La méthode de Newton-Raphson semble avoir échoué.`)
;
   printf(`\n|   k  |     x[k+1]-x[k]    |         x[k]       |
f(x[k])        | \n|======|====================|====================|==
======================| \n`);
  seq(printf("|%4d  | % 0.15f | % 0.15f | % 0.15E |\n",Détail[k,1],
Détail[k,2],Détail[k,3],Détail[k,4]),k=3..Itération+2);
 printf(`\n\n%s%d%s`, `Après `,Itération, ` itération(s), les
approximations successives`);
 printf(`\n%s`,`semblent diverger, soit par oscillation ou soit vers +
l'infini ou vers - l'infini.`);
 printf(`\n\n%s`,`Essayer, s'il y a lieu, une autre valeur d'amorce
plus près `):
 printf(`\n%s`,`du zéro réel à localiser et/ou augmenter le nombre
d'itérations.`);
 fi
 end:
```

La macro-commande

newton(f,amorce,maxiter,tolérance,tableau)

applique la méthode de Newton-Raphson pour estimer les zéros réels d'une fonction réelle d'une variable réelle.

La version finale de la macro-commande newton possède cinq arguments:

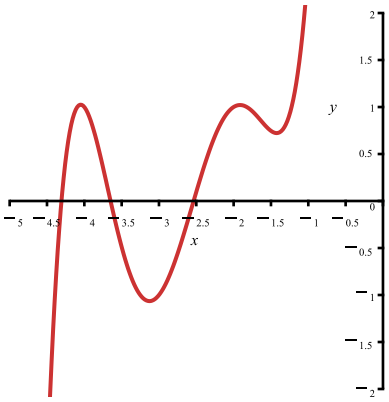
- f: La fonction f doit être une procédure définie avec l'opérateur fonctionnel flèche " -> "
- amorce: La valeur d'amorce doit être énoncée dans la forme *amorce* = valeur
- maxiter: Le nombre maximum d'itérations doit être donné dans la forme *n* = nombre
- tolérance: La précision de l'estimation doit être énoncée dans la forme *E* = valeur
- tableau: Si la valeur de l'argument tableau est *non*, c'est seulement un résumé des résultats des calculs qui sera affiché.
  Sinon, ce sera les détails des différents calculs qui seront affichés.

Exemple

Comparons les trois méthodes sur la base d'une même valeur $E = 0{,}000000000001$.

```
> f:=x->7/8*x^5+23/2*x^4+459/8*x^3+541/4*x^2+151*x+65;
  plot([x,f(x),x=-5..-1],color=orange,view=[-5..0,-2..2]);
```

$$f := x \mapsto \frac{7}{8}\cdot x^5 + \frac{23}{2}\cdot x^4 + \frac{459}{8}\cdot x^3 + \frac{541}{4}\cdot x^2 + 151\cdot x + 65$$



Approximons le zéro réel dans l'intervalle [-3, -2] avec $E = 0{,}00000000000001$.

```
> bissection(f,x=-3..-2,E=0.1e-14,oui);
  sécante(f,x=-3..-2,E=0.1e-14,oui);
  newton(f,amorce=-2.5,n=10,E=0.1e-14,oui);
```

```
Méthode de bissection avec E =  0.000000000000001
Intervalle initial x = -3 .. -2
```

k	d[k]	m[k]	f(m[k])
1	0.500000000000000	-2.500000000000000	0.097656250000000
2	0.250000000000000	-2.750000000000000	-0.558227539062500
3	0.125000000000000	-2.625000000000000	-0.239757537841797
4	0.062500000000000	-2.562500000000000	-0.071174502372742
5	0.031250000000000	-2.531250000000000	0.013487372547388
6	0.015625000000000	-2.546875000000000	-0.028816328034736
7	0.007812500000000	-2.539062500000000	-0.007653347402083
8	0.003906250000000	-2.535156250000000	0.002920333467159
9	0.001953125000000	-2.537109375000000	-0.002365743945877
10	0.000976562500000	-2.536132812500000	0.000277493921833
11	0.000488281250000	-2.536621093750000	-0.001044076272106
12	0.000244140625000	-2.536376953125000	-0.000383278858838
13	0.000122070312500	-2.536254882812500	-0.000052889373015
14	0.000061035156250	-2.536193847656250	0.000112303050333
15	0.000030517578125	-2.536224365234375	0.000029707032383
16	0.000015258789062	-2.536239624023438	-0.000011591121917
17	0.000007629394531	-2.536231994628906	0.000009057967337
18	0.000003814697266	-2.536235809326172	-0.000001266574264
19	0.000001907348633	-2.536233901977539	0.000003895697293
20	0.000000953674316	-2.536234855651855	0.000001314561703
21	0.000000476837158	-2.536235332489014	0.000000023993767
22	0.000000238418579	-2.536235570907593	-0.000000621290237
23	0.000000119209290	-2.536235451698303	-0.000000298648232
24	0.000000059604645	-2.536235392093658	-0.000000137327232
25	0.000000029802322	-2.536235362291336	-0.000000056666732
26	0.000000014901161	-2.536235347390175	-0.000000016336483
27	0.000000007450581	-2.536235339939594	0.000000003828642
28	0.000000003725290	-2.536235343664885	-0.000000006253920
29	0.000000001862645	-2.536235341802239	-0.000000001212639
30	0.000000000931323	-2.536235340870917	0.000000001308002
31	0.000000000465661	-2.536235341336578	0.000000000047681
32	0.000000000232831	-2.536235341569409	-0.000000000582479
33	0.000000000116415	-2.536235341452993	-0.000000000267399
34	0.000000000058208	-2.536235341394786	-0.000000000109859
35	0.000000000029104	-2.536235341365682	-0.000000000031089
36	0.000000000014552	-2.536235341351130	0.000000000008296
37	0.000000000007276	-2.536235341358406	-0.000000000011396
38	0.000000000003638	-2.536235341354768	-0.000000000001550
39	0.000000000001819	-2.536235341352949	0.000000000003373
40	0.000000000000909	-2.536235341353859	0.000000000000912
41	0.000000000000455	-2.536235341354313	-0.000000000000319
42	0.000000000000227	-2.536235341354086	0.000000000000296
43	0.000000000000114	-2.536235341354200	-0.000000000000012
44	0.000000000000057	-2.536235341354143	0.000000000000142
45	0.000000000000028	-2.536235341354171	0.000000000000065
46	0.000000000000014	-2.536235341354185	0.000000000000027

```
|  47  |  0.000000000000007 | -2.536235341354192 |  0.000000000000008 |
|  48  |  0.000000000000004 | -2.536235341354196 | -0.000000000000002 |
|  49  |  0.000000000000002 | -2.536235341354194 |  0.000000000000003 |
|  50  |  0.000000000000001 | -2.536235341354195 |  0.000000000000000 |

Méthode d'interpolation linéaire avec E = 0.000000000000001
Intervalle initial x = -3 .. -2
|   k  |         d[k]       |        c[k]        |       f(c[k])      |
|======|====================|====================|====================|
|   1  |  0.500000000000000 | -2.500000000000000 |  0.097656250000000 |
|   2  |  0.250000000000000 | -2.554112554112554 | -0.048434208509751 |
|   3  |  0.027056277056277 | -2.536172308352372 |  0.000170599078811 |
|   4  |  0.008970122880091 | -2.536235277213935 |  0.000000173596720 |
|   5  |  0.000031484430781 | -2.536235341354506 | -0.000000000000841 |
|   6  |  0.000000032070286 | -2.536235341354195 |  0.000000000000000 |
|   7  |  0.000000000000155 | -2.536235341354195 |  0.000000000000000 |
|   8  |  0.000000000000000 | -2.536235341354195 | -0.000000000000000 |

Méthode de Newton-Raphson
La valeur de l'amorce est x[0] = -2.5000000000000
|   k  |     x[k+1]-x[k]    |         x[k]       |         f(x[k])        |
|======|====================|====================|========================|
|   1  |  0.036443148688047 | -2.536443148688047 | -5.624430814445750E-04 |
|   2  | -0.000207804036573 | -2.536235344651473 | -8.924139592704960E-09 |
|   3  | -0.000000003297278 | -2.536235341354195 | -2.260416645343654E-18 |
|   4  | -0.000000000000000 | -2.536235341354195 |  1.000000000000000E-37 |

La méthode de Newton-Raphson semble donner un convergence.
Le zéro réel approximatif est r =-2.536235341354195 avec E =  0.000000000000001
```