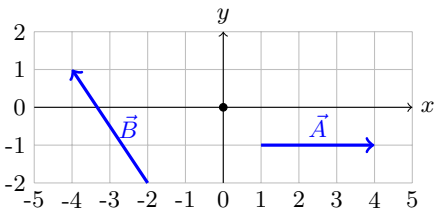
1 Affectation : produit scalaire

Objectif : Mettre en œuvre l'instruction d'affectation.

Syntaxe Python : `variable = expression`

Enoncé : On considère les 2 vecteurs $\vec{A}$ et $\vec{B}$ définis dans $\mathbb{R}^2$ comme le montre la figure ci-dessous.



Proposer une instruction de type « affectation » qui calcule le produit scalaire $\vec{A} \cdot \vec{B}$ de ces deux vecteurs.

Méthode : Il s'agit ici de déterminer le produit scalaire $\vec{V_1} \cdot \vec{V_2}$ de 2 vecteurs $\vec{V_1}$ et $\vec{V_2}$ de $\mathbb{R}^2$ ayant pour composantes respectives (x_1, y_1) et (x_2, y_2). Par définition, ce produit scalaire est un réel p qui a pour expression $p = x_1 \cdot x_2 + y_1 \cdot y_2$.

Connaissant les composantes respectivement (x_1, y_1) et (x_2, y_2) des vecteurs $\vec{V_1}$ et $\vec{V_2}$, on détermine le produit scalaire p par une affectation simple : `p = x1*x2 + y1*y2` .

Résultat : On applique la méthode précédente aux vecteurs $\vec{V_1} = \vec{A}$ et $\vec{V_2} = \vec{B}$. Leurs composantes respectives se lisent directement sur la figure : $x_1 = (4) - (1) = 3$, $y_1 = (-1) - (-1) = 0$, $x_2 = (-4) - (-2) = -2$ et $y_2 = (1) - (-2) = 3$.
Compte-tenu de ces valeurs, le code ci-contre permet de calculer le produit scalaire $\vec{A} \cdot \vec{B}$ demandé.

Listing 1 – **produit scalaire**

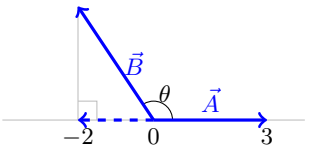
```
1 x1, y1 = (4)-(1), (-1)-(-1)
2 x2, y2 = (-4)-(-2), (1)-(-2)
3 p = x1*x2 + y1*y2
```

Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : Python les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la $3^{ème}$ ligne du code en `p = 3*(-2) + 0*3` : la forme plus abstraite `p = x1*x2 + y1*y2` restera identique pour deux autres vecteurs quelconques de $\mathbb{R}^2$.

Vérification : Une autre manière de déterminer le produit scalaire est donnée par la formule « géométrique » : $\vec{A} \cdot \vec{B} = |\vec{A}| \cdot |\vec{B}| \cdot \cos(\theta)$ où $|\vec{A}|$ et $|\vec{B}|$ représentent les modules des vecteurs $\vec{A}$

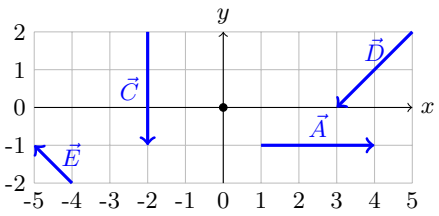
et $\vec{B}$ et θ l'angle formé par ces 2 vecteurs.

$|\vec{B}| \cdot \cos(\theta)$ représente donc la projection de $\vec{B}$ sur $\vec{A}$. Or cette projection se lit très facilement sur la figure puisque $\vec{A}$ est horizontal (parallèle à l'axe des abscisses avec $|\vec{A}| = 3$) : elle a pour valeur (-4) - (-2) = -2, d'où le produit scalaire $p = 3 \cdot (-2) = -6$.



On obtient bien par le calcul analytique le résultat obtenu par la projection géométrique.

Généricité : Pour vérifier la généricité de la méthode précédente, calculer , à l'aide du code PYTHON précédent, les produits scalaires suivants : $\vec{A} \cdot \vec{A}$, $\vec{A} \cdot \vec{C}$, $\vec{A} \cdot \vec{D}$, $\vec{A} \cdot \vec{E}$, $\vec{C} \cdot \vec{C}$, $\vec{C} \cdot \vec{D}$, $\vec{C} \cdot \vec{E}$, $\vec{D} \cdot \vec{D}$, $\vec{D} \cdot \vec{E}$ et $\vec{E} \cdot \vec{E}$. Les vecteurs $\vec{A}$, $\vec{C}$, $\vec{D}$ et $\vec{E}$ sont définis sur la figure ci-dessous.



Vérifier en particulier qu'on obtient bien les résultats attendus pour le produit scalaire d'un vecteur par lui-même et pour le produit scalaire de deux vecteurs perpendiculaires.

Entraînement : Dans le même esprit, utiliser l'affectation pour calculer :

1. le produit vectoriel $\vec{A} \times \vec{B}$ de 2 vecteurs $\vec{A}$ et $\vec{B}$ de $\mathbb{R}^3$ de composantes respectives (a_1, a_2, a_3) et (b_1, b_2, b_3) ;
2. le produit mixte $\vec{A} \cdot (\vec{B} \times \vec{C})$ de 3 vecteurs $\vec{A}$, $\vec{B}$ et $\vec{C}$ de $\mathbb{R}^3$ de composantes respectives (a_1, a_2, a_3), (b_1, b_2, b_3) et (c_1, c_2, c_3).

2 Alternative : graphe d'une fonction

Objectif : Mettre en œuvre l'instruction d'alternative multiple.

Syntaxe Python :

test simple

```
if condition : bloc
```

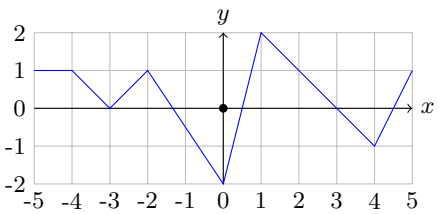
alternative simple

```
if condition : bloc1
else         : bloc2
```

alternative multiple

```
if   condition1 : bloc1
elif condition2 : bloc2
elif condition3 : bloc3
...
else            : blocn
```

Enoncé : On considère dans $\mathbb{R}$ la fonction continue f, affine par morceaux, définie sur $[-5;5]$ par le graphe ci-dessous et $\forall x < -5, f(x) = f(-5)$ et $\forall x > 5, f(x) = f(5)$.



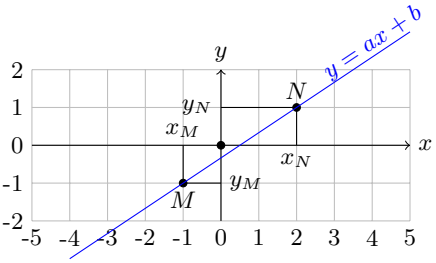
Proposer une instruction de type « alternative multiple » qui calcule la fonction $y = f(x)$ $\forall x \in \mathbb{R}$.

Méthode : Il s'agit de déterminer la valeur $y = f(x)$ d'une fonction continue affine par morceaux sur $\mathbb{R}$. L'axe des réels $]-\infty, x_1, x_2, \ldots, x_n, +\infty[$ est donc vu comme une succession d'intervalles $]-\infty, x_1[$, $[x_1, x_2[$, ..., $[x_{n-1}, x_n[$ et $[x_n, +\infty[$ sur lesquels la fonction f est définie respectivement par les fonctions f_1, f_2, ..., f_n et f_{n+1} :

$$\begin{aligned} y = f(x) &= f_1(x) && \forall x \in]-\infty, x_1[\\ &= f_2(x) && \forall x \in [x_1, x_2[\\ &= f_3(x) && \forall x \in [x_2, x_3[\\ &= \ \ldots \\ &= f_n(x) && \forall x \in [x_{n-1}, x_n[\\ &= f_{n+1}(x) && \forall x \in [x_n, +\infty[\end{aligned}$$

Chacune des fonctions f_i correspond à une droite d'équation $y = a_i x + b_i$ où a_i représente la pente de la droite et b_i son ordonnée à l'origine. Lorsqu'on connaît 2 points $M(x_M, y_M)$ et $N(x_N, y_N)$ d'une droite d'équation $y = ax+b$, les coefficients a (pente de la droite) et b (ordonnée à l'origine) de la droite sont obtenus par résolution du système de 2 équations : $y_M = ax_M + b$ et $y_N = ax_N + b$. On obtient alors a et b :

$$a = \frac{y_N - y_M}{x_N - x_M} \text{ et } b = \frac{y_M x_N - y_N x_M}{x_N - x_M}$$



Pour la droite ci-contre :

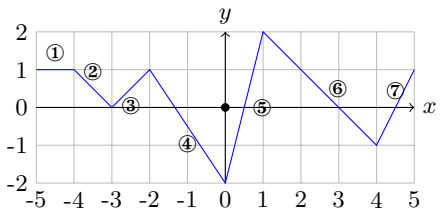
$$a = \frac{1-(-1)}{2-(-1)} = \frac{2}{3} \text{ et } b = \frac{(-1)\cdot 2 - 1\cdot(-1)}{2-(-1)} = -\frac{1}{3}$$

On vérifie graphiquement ces résultats : pour passer de M à N, on se déplace de $\Delta x = 3$ horizontalement puis de $\Delta y = 2$ verticalement (d'où la pente $a = \Delta y/\Delta x = 2/3$), et la droite coupe bien l'axe des ordonnées en $y = -1/3$.

Une fois déterminés les coefficients a_i et b_i de chaque droite, on détermine la valeur de la fonction $y = f(x)$ par une alternative multiple du genre :

```
if x < x1 : y = a1*x + b1
elif x < x2 : y = a2*x + b2
elif x < x3 : y = a3*x + b3
...
elif x < xn : y = an*x + bn
else : y = an+1*x + bn+1
```

Résultat : On applique la méthode précédente à la fonction f de l'énoncé. Il faut donc déterminer les équations de droite correspondant aux différents segments du graphe de la fonction, à savoir :



① $y = 1$
② $y = -x - 3$
③ $y = x + 3$
④ $y = -3x/2 - 2$
⑤ $y = 4x - 2$
⑥ $y = -x + 3$
⑦ $y = 2x - 9$

Compte-tenu de ces équations, le code ci-contre permet de calculer $y = f(x)$, y compris pour $x < -5$ ($y = f(-5) = 1$) et $x > 5$ ($y = f(5) = 1$).
Remarque : on aurait pu simplifier les deux premières lignes de ce code en

```
if x < -4 : y = 1
```

car les instructions associées sont identiques (ie. les fonctions affines sont identiques sur $]-\infty, -5[$ et $[-5, -4[$).

Listing 2 – **graphe d'une fonction**

```
1 if    x < -5 : y = 1
2 elif  x < -4 : y = 1
3 elif  x < -3 : y = -x - 3
4 elif  x < -2 : y = x + 3
5 elif  x <  0 : y = -3*x/2 - 2
6 elif  x <  1 : y = 4*x - 2
7 elif  x <  4 : y = -x + 3
8 elif  x <  5 : y = 2*x - 9
9 else         : y = 1
```

Vérification : Pour tester le résultat précédent, on peut comparer les valeurs obtenues par le calcul avec celles lues directement sur le graphe pour quelques points caractéristiques. Ces points de mesure sont choisis judicieusement : ils ne correspondent pas aux bornes des intervalles déjà prises en compte dans la méthode mais plutôt à des points où la fonction s'annule (exemples :

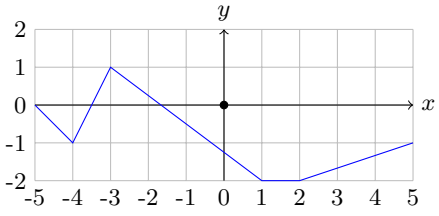
$x = -4/3$, $1/2$, 3 ou $9/2$) ou à des points d'abscisses aux nœuds de la grille de lecture (exemples : $x = -1$ ou $x = 2$). On peut vérifier par exemple pour $x = -1$ ($y = f(-1) = -1/2$) et $x = 3$ ($y = f(3) = 0$).

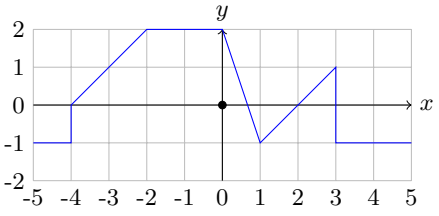
```
>>> x = -1
>>> if x < -4 : y = 1
elif   x < -3 : y = -x - 3
elif   x < -2 : y = x + 3
elif   x <  0 : y = -3*x/2 - 2
elif   x <  1 : y = 4*x - 2
elif   x <  4 : y = -x + 3
elif   x <  5 : y = 2*x - 9
else          : y = 1

>>> y
-0.5
```

```
>>> x = 3
>>> if x < -4 : y = 1
elif   x < -3 : y = -x - 3
elif   x < -2 : y = x + 3
elif   x <  0 : y = -3*x/2 - 2
elif   x <  1 : y = 4*x - 2
elif   x <  4 : y = -x + 3
elif   x <  5 : y = 2*x - 9
else          : y = 1

>>> y
0
```

On obtient bien par le calcul les résultats lus sur la grille.

Généricité : Pour vérifier la généricité de la méthode précédente, proposer une alternative multiple pour chacune des 2 fonctions définies sur $[-5; 5]$ par les graphes ci-dessous et $\forall x < -5, f(x) = f(-5)$ et $\forall x > 5, f(x) = f(5)$.

1. 

2. 

Entraînement : Utiliser une alternative multiple pour déterminer les racines réelles d'un trinôme du second degré $ax^2 + bx + c$ à coefficients a, b et c réels.

3 Boucle : intégration numérique

Objectif : Mettre en œuvre l'instruction d'itération en respectant les 4 étapes de la construction d'une boucle :

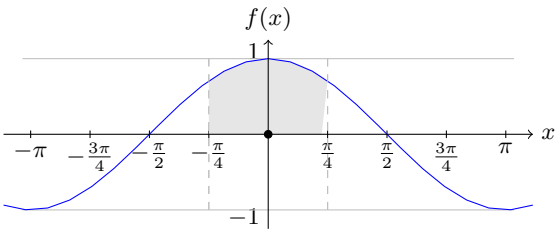
1. **Invariant :** proposer une situation générale décrivant le problème posé.
2. **Condition d'arrêt :** à partir de la situation générale imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors est appelée situation finale. La condition d'arrêt fait sortir de la boucle.
3. **Progression :** se « rapprocher » de la situation finale, tout en faisant le nécessaire pour conserver à chaque étape une situation générale analogue à celle retenue en [1]. La progression conserve l'invariant.
4. **Initialisation :** initialiser les variables introduites dans l'invariant pour que celui-ci soit vérifié avant d'entrer dans la boucle. L'initialisation « instaure » l'invariant.

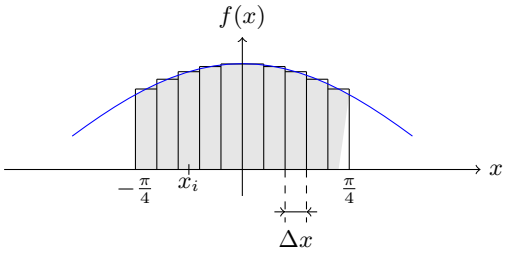
Syntaxe Python : `while condition : bloc`

Enoncé : On considère l'intégrale I de la fonction $x \mapsto \cos(x)$ entre $-\frac{\pi}{4}$ et $\frac{\pi}{4}$:

$$I = \int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} \cos(x)dx$$

Cette intégrale représente classiquement l'aire comprise entre la courbe représentative de la fonction cosinus et les droites d'équations $x = -\pi/4$, $x = \pi/4$ et $y = 0$ (en grisé sur la figure ci-dessous). Pour calculer numériquement cette intégrale, on utilisera la méthode des rectangles selon laquelle l'aire sous la courbe est assimilée à la somme des surfaces de n petits rectangles de base Δx (voir figure de droite ci-dessous) ; ce qui revient à approximer la fonction cosinus par une fonction en escalier.





Proposer une instruction de type « boucle » qui calcule l'intégrale I de la fonction cosinus entre $-\pi/4$ et $\pi/4$ par la méthode des rectangles.

Méthode : Il s'agit ici d'intégrer une fonction continue f de $\mathbb{R} \to \mathbb{R}$ sur un intervalle $[a, b]$. On supposera que f à toutes les bonnes propriétés mathématiques pour être intégrable sur l'intervalle considéré.

On cherche donc à calculer l'intégrale $I = \int_a^b f(x)dx$ qui représente classiquement l'aire comprise entre la courbe représentative de f et les droites d'équations $x = a$, $x = b$ et $y = 0$. Les

méthodes d'intégration numérique consistent essentiellement à trouver une bonne approximation de cette aire.

Dans la méthode des rectangles proposée, on subdivise l'intervalle d'intégration de longueur $b-a$ en n parties égales de longueur $\Delta x = \dfrac{b-a}{n}$. Soient x_1, x_2, ..., x_n les points milieux de ces n intervalles. Les n rectangles formés avec les ordonnées correspondantes ont pour surface $f(x_1)\Delta x$, $f(x_2)\Delta x$, ..., $f(x_n)\Delta x$. L'aire sous la courbe est alors assimilée à la somme des aires de ces rectangles, soit

$$I = \int_a^b f(x)dx \approx \left(f(x_1) + f(x_2) + \cdots + f(x_n)\right)\Delta x = \Delta x \cdot \sum_{i=1}^{i=n} f(x_i)$$

C'est la formule dite des rectangles qui repose sur une approximation par une fonction *en escalier*.

Le calcul attendu revient donc à évaluer d'abord la somme $s = \sum f(x_i)$ puis à la multiplier par la largeur $w = \Delta x$ des petits rectangles. Pour calculer s « de tête », on évalue d'abord $f(x_1)$ que l'on mémorise : $s = f(x_1)$, puis on rajoute $f(x_2)$ et on mémorise cette somme intermédiaire : $s = s + f(x_2)$ $(= f(x_1) + f(x_2))$, puis on rajoute $f(x_3)$ et on mémorise cette nouvelle somme intermédiaire $s = s + f(x_3)$ $(= f(x_1) + f(x_2) + f(x_3))$ et ainsi de suite jusqu'à rajouter enfin $f(x_n)$: $s = s + f(x_n)$ $(= f(x_1) + f(x_2) + f(x_3) + \ldots + f(x_n))$. On a alors la somme recherchée $(s = \sum f(x_i))$ et il ne reste plus qu'à multiplier cette somme s par la largeur w pour obtenir la valeur de l'intégrale I.

Les 4 étapes de construction de la boucle recherchée sont donc les suivantes :

- pour l'invariant, on vérifie qu'à chaque étape k, s_k est toujours égal à la somme des k premiers termes $f(x_k)$: $\forall k, s_k = \sum_{i=1}^{k} f(x_i)$;
- l'initialisation consiste à définir w, à se positionner sur le premier rectangle $x_1 = a + w/2$ et à initialiser s_1 en conséquence : $s_1 = f(x_1)$;
- la progression consiste à se déplacer sur le rectangle suivant : $x_{k+1} = x_k + w$ et à augmenter s_k de la nouvelle valeur $f(x_{k+1})$ afin de conserver l'invariant : $s_{k+1} = s_k + f(x_{k+1})$;
- la condition d'arrêt est obtenue pour $x_k > b - w/2$.

La boucle prend ainsi la forme :

```
« initialisation »
while not « condition d'arrêt » :
    « progression »
```

$w = (b-a)/2$, $x = x_1 = a + w/2$, $s = f(x_1)$
`while not` $x > b - w/2$:
    $x = x_{k+1} = x_k + w$
    $s = s_{k+1} = s_k + f(x_{k+1})$

Résultat : On applique la méthode précédente à la fonction $f : x \mapsto cos(x)$ dans l'intervalle $[a = -\dfrac{\pi}{4}, b = \dfrac{\pi}{4}]$.

Le code PYTHON ci-contre permet de calculer l'intégrale recherchée à condition de ne pas oublier de définir f, a, b et n avant d'exécuter la boucle précédente. On prendra par exemple $n = 100$.

Listing 3 – **intégration numérique**

```
1 w = (b - a)/n
2 x = a + w/2
3 s = f(x)
4 while not x > (b - w/2):
5     x = x + w
6     s = s + f(x)
7 s = w*s
```

Vérification : Pour vérifier le résultat obtenu, on peut le comparer au calcul direct car la primitive de $\cos(x)$ est connue, il s'agit de $\sin(x)$:

$$I = \int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} \cos(x)dx = [\sin(x)]_{-\frac{\pi}{4}}^{\frac{\pi}{4}} = \sin(\frac{\pi}{4}) - \sin(-\frac{\pi}{4}) = 2\sin(\frac{\pi}{4}) = 2\frac{\sqrt{2}}{2} = \sqrt{2}$$

Ainsi, doit-on trouver une valeur proche de $\sqrt{2}$ ($\approx 1.4142135623730951$).

```
>>> from math import cos, pi
>>> f = cos
>>> a, b = -pi/4, pi/4
>>> n = 100
>>> w = (b - a)/n
x = a + w/2
s = f(x)
while not x > (b - w/2):
    x = x + w
    s = s + f(x)
s = w*s
>>> s
1.4142281017781444
```

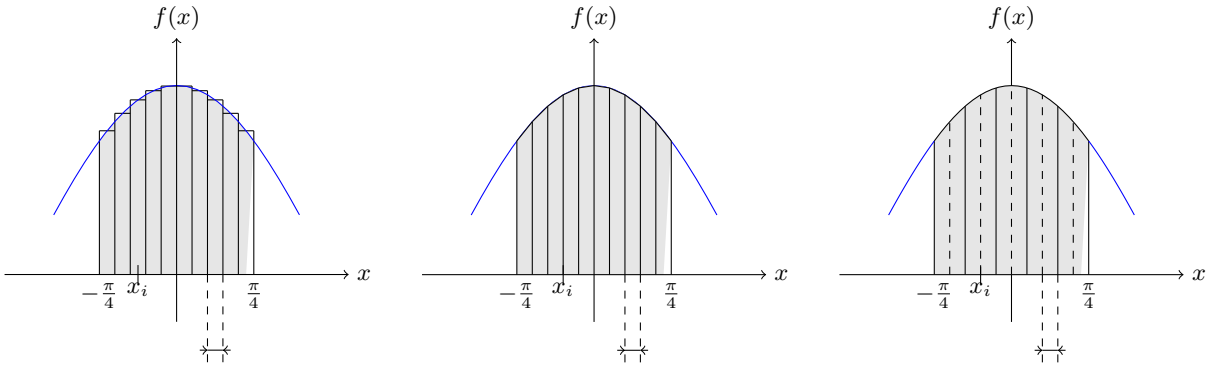
```
>>> from math import cos, pi
>>> f = cos
>>> a, b = -pi/4, pi/4
>>> n = 100000
>>> w = (b - a)/n
x = a + w/2
s = f(x)
while not x > (b - w/2):
    x = x + w
    s = s + f(x)
s = w*s
>>> s
1.4142135623875653
```

On trouve le bon ordre de grandeur quelle que soit la valeur de n mais plus n est grand (ie. les rectangles sont de moins en moins larges) meilleur est le résultat.

Généricité : Pour vérifier la généricité de la méthode précédente, calculer l'intégrale des fonctions suivantes par la méthode des rectangles :

1. $y = \cos(x)$ dans $[0, \pi]$ puis dans $[-\pi/2, \pi/2]$,
2. $y = \sin(x)$ dans $[0, \pi]$ puis dans $[-\pi/2, \pi/2]$,
3. $y = 3x$ dans $[0, 1]$ puis dans $[-1, 1]$.

Entraînement : Déterminer l'intégrale $I = \int_a^b f(x)dx$ selon deux méthodes alternatives : la méthode des trapèzes et la méthode des paraboles (ou méthode de Simpson).



méthode des rectangles — méthode des trapèzes — méthode des paraboles

1. Méthode des trapèzes : on subdivise l'intervalle d'intégration de longueur $b-a$ en n parties égales de longueur $\Delta x = \frac{b-a}{n}$. Les abscisses des points ainsi définis sont a, x_1, x_2, ..., x_{n-1}, b et les trapèzes construits sur ces points et les ordonnées correspondantes ont pour aire

$\frac{\Delta x}{2}(f(a)+f(x_1))$, $\frac{\Delta x}{2}(f(x_1)+f(x_2))$, ..., $\frac{\Delta x}{2}(f(x_{n-1})+f(b))$. L'aire sous la courbe est alors assimilée à la somme des aires de ces trapèzes, soit :

$$I=\int_a^b f(x)dx \approx \left(\frac{f(a)+f(b)}{2}+f(x_1)+f(x_2)+\cdots+f(x_{n-1})\right)\Delta x$$

C'est la formule dite des trapèzes.

2. Méthodes des paraboles : on divise l'intervalle d'intégration $[a,b]$ en un nombre n pair d'intervalles dont la longueur est $\Delta x=\frac{b-a}{n}$. Dans les 2 premiers intervalles d'extrémités a, x_1 et x_2, on approche la courbe représentative de f par une parabole d'équation $y=\alpha x^2+\beta x+\gamma$ passant par les points $A(a,f(a))$, $A_1(x_1,f(x_1))$ et $A_2(x_2,f(x_2))$ de la courbe. Dans les 2 intervalles suivants, on approche la courbe par une autre parabole d'équation similaire, passant par les points A_2, A_3 et A_4, et ainsi de suite. On obtient ainsi une courbe formée de n portions de parabole et l'aire déterminée par ces portions de parabole est une approximation de l'aire I cherchée.
L'intégration de l'équation de la parabole $y=\alpha x^2+\beta x+\gamma$ sur $[-\Delta x,\Delta x]$ donne

$$S=\int_{-\Delta x}^{\Delta x}(\alpha x^2+\beta x+\gamma)dx=\frac{2}{3}\alpha(\Delta x)^3+2\gamma(\Delta x)$$

où les constantes α et γ sont déterminées en écrivant que les points $(-\Delta x,y_0)$, $(0,y_1)$ et $(\Delta x,y_2)$ satisfont l'équation de la parabole. On obtient ainsi :

$$\left|\begin{array}{l} y_0=\alpha(-\Delta x)^2+\beta(-\Delta x)+\gamma \\ y_1=\gamma \\ y_2=\alpha(\Delta x)^2+\beta(\Delta x)+\gamma \end{array}\right. \Rightarrow \left|\begin{array}{l} \alpha=\dfrac{y_0-2y_1+y_2}{2(\Delta x)^2} \\ \beta=\dfrac{y_2-y_0}{2(\Delta x)} \\ \gamma=y_1 \end{array}\right.$$

et $S=\frac{\Delta x}{3}(y_0+4y_1+y_2)$.

Par suite, il vient :

$$\left|\begin{array}{lll} S_1 & = & \dfrac{\Delta x}{3}(y_0+4y_1+y_2) \\ S_2 & = & \dfrac{\Delta x}{3}(y_2+4y_3+y_4) \\ S_3 & = & \dfrac{\Delta x}{3}(y_4+4y_5+y_6) \\ \vdots & = & \\ S_{n/2} & = & \dfrac{\Delta x}{3}(y_{n-2}+4y_{n-1}+y_n) \end{array}\right.$$

d'où

$$I=\int_a^b f(x)dx \approx \frac{\Delta x}{3}\left(f(a)+4\sum_{i=1,3,5...}^{n-1}f(x_i)+2\sum_{i=2,4,6...}^{n-2}f(x_i)+f(b)\right)$$

C'est la formule dite de Simpson qui repose sur une approximation de f par des arcs de parabole.

4 Boucle : développement limité

Objectif : Mettre en œuvre l'instruction d'itération en respectant les 4 étapes de la construction d'une boucle :

1. **Invariant :** proposer une situation générale décrivant le problème posé.
2. **Condition d'arrêt :** à partir de la situation générale imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors est appelée situation finale. La condition d'arrêt fait sortir de la boucle.
3. **Progression :** se « rapprocher » de la situation finale, tout en faisant le nécessaire pour conserver à chaque étape une situation générale analogue à celle retenue en [1]. La progression conserve l'invariant.
4. **Initialisation :** initialiser les variables introduites dans l'invariant pour que celui-ci soit vérifié avant d'entrer dans la boucle. L'initialisation « instaure » l'invariant.

Syntaxe Python : `while condition : bloc`

Enoncé : On considère $\forall x \in]-1;1[$ la fonction $f(x)$ définie par

$$\begin{aligned} y = f(x) = \frac{1}{(1+x)^a} &\approx 1 + \sum_{k=1}^{n} u_k = 1 + \sum_{k=1}^{n} (-1)^k a(a+1)\cdots(a+k-1)\frac{x^k}{k!} \\ &= 1 - ax + a(a+1)\frac{x^2}{2} + \ldots + (-1)^k a(a+1)\cdots(a+k-1)\frac{x^k}{k!} \end{aligned}$$

Ecrire un algorithme qui calcule $y = f(x)$ en respectant les contraintes suivantes :
- les calculs seront arrêtés lorsque la valeur absolue du terme u_k ($|u_k|$) sera inférieure à un certain seuil s (avec $0 < s < 1$) ;
- on n'utilisera ni la fonction *puissance* (x^n) ni la fonction *factorielle* ($n!$) pour effectuer le calcul du développement.

Méthode : On cherche une relation de récurrence g entre u_{k+1} et u_k afin de faciliter le calcul de u_{k+1} sans faire appel aux fonctions *puissance* (x^k) et *factorielle* ($k!$).

On a :

$$u_{k+1} = (-1)^{k+1} a(a+1)\cdots(a+(k+1)-1)\frac{x^{k+1}}{(k+1)!}$$

et on cherche donc à faire apparaître u_k dans l'expression de u_{k+1} :

$$u_{k+1} = (-1)\cdot(-1)^k \cdot a(a+1)\cdots(a+k-1)\cdot(a+k)\cdot\frac{x\cdot x^k}{k!(k+1)}$$

$$u_{k+1} = -x\cdot\frac{(a+k)}{(k+1)}\cdot(-1)^k a(a+1)\cdots(a+k-1)\frac{x^k}{k!} = -x\cdot\frac{(a+k)}{(k+1)}\cdot u_k = g(u_k)$$

Ainsi, dans la pratique, si on mémorise la valeur de u_k, on calculera sans problème la valeur

de u_{k+1} grâce à cette relation de récurrence ($u_{k+1} = g(u_k)$).

$$\begin{array}{lcllcll}
y_1 & = & 1 + u_1 & & & & \\
y_2 & = & 1 + u_1 + u_2 & = & y_1 + u_2 & = & y_1 + g(u_1) \\
y_3 & = & 1 + u_1 + u_2 + u_3 & = & y_2 + u_3 & = & y_2 + g(u_2) \\
\dots & & & & & & \\
y_{m+1} & = & 1 + u_1 + u_2 + \dots + u_m + u_{m+1} & = & y_m + u_{m+1} & = & y_m + g(u_m)
\end{array}$$

Les 4 étapes de construction de l'algorithme recherché sont donc les suivantes :

- pour l'invariant, on vérifie qu'à chaque étape m, y est toujours égal à la somme des m premiers termes u_k : $\forall m, y_m = 1 + \sum_{k=1}^{m} u_k$;
- l'initialisation consiste à décrire le cas $m = 1$: $u_1 = -ax$ et $y = y_1 = 1 + u_1 = 1 - ax$;
- la progression consiste à calculer l'étape suivante à partir de l'étape courante ($m \to m+1$) : $u_{m+1} = g(u_m) = -x \cdot \dfrac{(a+m)}{(m+1)} \cdot u_m$, $y_{m+1} = y_m + g(u_m)$;
- la condition d'arrêt est contrainte par l'énoncé : $|u_m| < s$.

L'algorithme recherché prendra la forme :

```
« initialisation »
while not « condition d'arrêt » :
    « progression »
```

$m = 1$, $u_1 = -ax$, $y_1 = 1 - ax$
`while not` $|u_m| < s$:
    $u_{m+1} = g(u_m)$
    $y_{m+1} = y_m + u_{m+1}$
    $m \to m + 1$

Résultat : En PYTHON, l'algorithme correspondant est présenté ci-contre. Pour le tester, on fixera différentes valeurs des données a, x et s, et pour le vérifier, on comparera la valeur y calculée avec le calcul direct $(1+x)^{-a}$.

```
1 m = 1
2 u = -a*x
3 y = 1 + u
4 while not fabs(u) < s :
5     u = -u*x*(a + m)/(m + 1)
6     y = y + u
7     m = m + 1
```

Vérification : On compare le résultat obtenu par l'algorithme avec le calcul direct : $y - \frac{1}{(1+x)^a}$, pour différentes valeurs des données a, x et s.

```
>>> a, x, s = 1, 0, 1.0e-9
...
>>> y - 1/(1-x)**a
0.0

>>> a, x, s = 1/2, 0.5, 1.0e-9
...
>>> y - 1/(1-x)**a
-2.65160005064e-10

>>> a, x, s = 3, 0.5, 1.0e-9
...
>>> y - 1/(1-x)**a
2.69446243095e-10
>>> a, x, s = 3, 0.5, 1.0e-4
...
>>> y - 1/(1-x)**a
2.31884143971e-05
```

```
>>> a, x, s = 5, 0, 1.0e-9
...
>>> y - 1/(1-x)**a
0.0

>>> a, x, s = 5, 0.85, 1.0e-9
...
>>> y - 1/(1-x)**a
4.27782663459e-10

>>> a, x, s = 5, -0.85, 1.0e-9
...
>>> y - 1/(1-x)**a
-5.8480509324e-09
>>> a, x, s = 5, -0.85, 1.0e-5
...
>>> y - 1/(1-x)**a
-6.00809507887e-05
```

Les différences observées sont toutes voisines de 0 et compatibles avec le seuil de précision s donné.

Généricité : Pour vérifier la généricité de la méthode précédente, calculer les fonctions suivantes à l'aide de leur développement limité.

1. $\cos(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} + \cdots + (-1)^n \frac{x^{2n}}{(2n)!}$ $\quad \forall x \in \mathbb{R}$
2. $\arctan(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)}$ $\quad \forall x \in \mathbb{R}$
3. $\sqrt{1+x} \approx 1 + \frac{x}{2} + \sum_{k=2}^{n} (-1)^{k-1} \frac{1 \times 3 \times \cdots \times (2k-3)}{2^k} \frac{x^k}{k!} =$
   $1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \cdots + (-1)^{n-1} \frac{1 \times 3 \times \cdots \times (2n-3)}{2^n} \frac{x^n}{n!}$ $\quad \forall x \in]-1;1[$

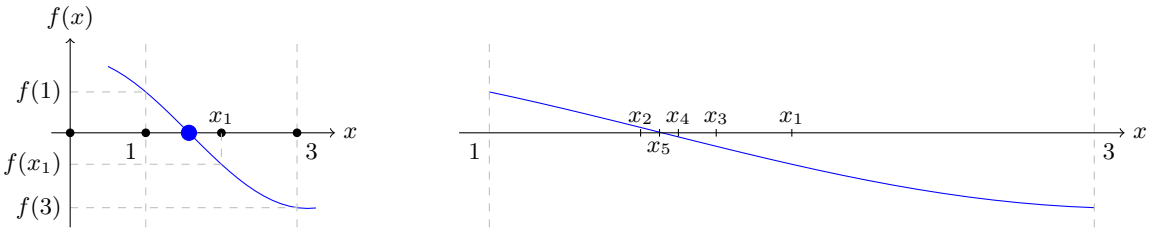
Poursuite : Pour aller un peu plus loin,

5 Boucle : zéro d'une fonction

Objectif : objectif

Syntaxe Python : `python`

Enoncé : Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans l'intervalle $[1,3]$ selon une méthode par dichotomie. On pose $a = 1$, $b = 3$ les extrémités de l'intervalle et $x_1 = (a+b)/2$ le milieu de l'intervalle. Si $f(a).f(x_1) < 0$ (ie. a et x_1 sont situés de part et d'autre du zéro), alors la racine est dans $]a, x_1[$ et on pose $b = x_1$; sinon la racine est dans $]x_1, b[$ et on pose $a = x_1$. Puis on réitère le procédé, la longueur de l'intervalle ayant été divisée par deux. On arrête lorsque a et b sont suffisamment proches.



Méthode : Il s'agit ici de rechercher le zéro d'une fonction f continue sur un intervalle $[a,b]$ telle que $f(a).f(b) < 0$; il existe alors une racine de f dans $]a,b[$ que nous supposerons unique.

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Déterminer le zéro d'une fonction selon les trois méthodes alternatives suivantes.

1. Méthode des tangentes : soit x_n une approximation de la racine c recherchée : $f(c) = f(x_n) + (c - x_n)f'(x_n)$; comme $f(c) = 0$, on a : $c = x_n - f(x_n)/f'(x_n)$. Posons $x_{n+1} = x_n - f(x_n)/f'(x_n)$: on peut considérer que x_{n+1} est une meilleure approximation de c que x_n. On recommence le procédé avec x_{n+1} et ainsi de suite jusqu'à ce que $|x_{n+1} - x_n|$ soit inférieur à un certain seuil s.
2. Méthode des sécantes : reprendre la méthode des tangentes en effectuant l'approximation suivante : $f'(x_n) = (f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})$.
3. Méthode des cordes : reprendre la méthode par dichotomie en prenant pour x le point d'intersection de la corde AB et de l'axe des abscisses : $x = (x_2 f(x_1) - x_1 f(x_2))/(f(x_1) - f(x_2))$, c'est-à-dire le point obtenu par la méthode des sécantes.

6 Boucles imbriquées : produit de matrices

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

7 Spécification : racines du trinôme

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

8 Appels de fonctions :

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

9 Récursivité :

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

10 Recherche :

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

11 Tri :

Objectif : objectif

Syntaxe Python : `python`

Enoncé : énoncé

Méthode : Il s'agit ici de

Résultat : résultat

Vérification : vérification

Généricité : Pour vérifier la généricité de la méthode précédente,

Poursuite : Pour aller un peu plus loin,

12 Projet : résolution de systèmes linéaires