Annexe B

Travaux pratiques et langage Python

B.1 TP nº 1 : Algorithme de recherche d'un zéro par balayage

Le balayage est une méthode pour trouver une valeur approchée de la solution d'une équation de la forme $f(x) = 0$. Elle repose sur l'idée suivante (théorème admis) :

> Si f est une fonction **continue** croissante sur un intervalle $[a; b]$. Si $f(a)$ et $f(b)$ sont de signes contraires, alors il existe un nombre unique x dans $[a; b]$ tel que $f(x) = 0$.

L'algorithme est le suivant : pour un encadrement à 10^{-3} près de la solution d'une équation $f(x) = 0$ sur l'intervalle $[0; 10]$. On effectue les opérations suivantes :

① on commence par balayer l'intervalle $[0; 10]$ avec un pas de 1.

C'est-à-dire qu'on calcule $f(0)$, $f(1)$, $f(2)$, …
On s'arrête dès qu'on a trouvé deux entiers consécutifs n et $n+1$ pour lesquels $f(n)$ et $f(n+1)$ sont de signes opposés. On sait alors que $f(x) = 0$ admet une solution dans l'intervalle $[n; n+1]$.

② on balaie ensuite l'intervalle $[n, n+1]$ avec un pas de $0,1$. On calcule donc $f(n)$, $f(n+0{,}1)$,$f(n+0{,}2)$, …et on s'arrête dès qu'on a trouvé p de sorte que $f(n + \frac{p}{10})$ et $f(n + \frac{p+1}{10})$ sont de signes opposés.

③ on continue en balayant l'intervalle $[n + \frac{p}{10}, n + \frac{p+1}{10}]$ avec un pas de $0,01$, puis avec un pas de $0,001$ et ainsi de suite.

Préliminaires : faire tourner l'algorithme à la main On souhaite résoudre l'équation $x^3 = 18$. Soit la fonction f définie sur $\mathbb{R}$ par $f(x) = x^3 - 18$. On cherche la solution de $f(x) = 0$ sur $[0; 10]$.

1) Compléter le tableau de valeur suivant :

x	0	1	2	3	4	5	6	7	8	9	10
$f(x)$											

L'équation $f(x) = 0$ admet une solution x^* sur l' intervalle .....

2) Remplir le tableau suivant et donner un encadrement de $18^{\frac{1}{3}}$ à 10^{-1} près.

x	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9
$f(x)$										

encadrement de x^* à 10^{-2} près : .....

3) Remplir le tableau suivant et donner un encadrement de $18^{\frac{1}{3}}$ à 10^{-2} près.

x										
$f(x)$										

Donner un encadrement de x^* à 10^{-3} près : .....

Partie A : l'algorithme en Python

1) Le script ci-dessous exécute la première étape. Choisir les bon éléments pour compléter les pointillés :

```
1  def f(x):            # définition de la fonction f
2      return x**3-18
3  binf , bsup= 0 , 10  # encadrement initial [0,10]
4  precision = 0
5
6  x = binf
7  pas = 10**(-precision)
8  for i in range(3) :
9      precision = i
10     pas = 10**(-i)
11     while ..............................  :
12         x = x + pas
13         binf , bsup= ..............  # nouveau encadrement
14
15 print(f"{binf}<18**(1/3)<{bsup}")
```

pour la ligne 11

- ❑ `f(binf)*f(x) < 0`
- ❑ `f(binf)*f(x) > 0`
- ❑ `f(binf)*f(x) < 0 and x <= bsup`
- ❑ `f(binf)*f(x) > 0 and x <= bsup`

pour la ligne 13

- ❑ `x , x+pas`
- ❑ `x-pas , x`
- ❑ `x , x-pas`
- ❑ `x-pas, x+pas`

2) Comment modifier l'algorithme pour un encadrement à 10^{-6} près ? .....

Partie B Mise en situation

Modifier l'algorithme Python pour obtenir un encadrement à 10^{-5} de la solution de l'équation $x^3 + 2x^2 + 10x + 5 = 0$ d'inconnue x.

B.2 TP nº 2 Algorithme de recherche d'un zéro par dichotomie

Dichotomie signifie : division en deux parts égales. En mathématiques elle désigne un algorithme de recherche d'un zéro d'une fonction qui consiste à répéter des partages d'un intervalle en deux parties puis à sélectionner le sous-intervalle dans lequel existe un zéro de la fonction.

Problème 1 — Méthode de dichotomie. Pour cet exercice, nous allons écrire en langage Python un programme qui vise à trouver une valeur approchée de la solution de l'équation : $x^3 + 2x^2 + 10x - 20 = 0$. Soit la fonction f définie sur $\mathbb{R}$ avec $f(x) = x^3 + 2x^2 + 10x - 20$.

Préliminaire : variation et nombre de zéros

1) Représenter la fonction f à l'aide de votre calculatrice et remplir le tableau de variation suivant

x	-3		1		2		3
$f(x)$							

2) Préciser le nombre de solutions de l'équation $f(x) = 0$ sur l'intervalle $[-3; 3]$ et donner un encadrement pour chaque.

.....

Partie A : faire tourner l'algorithme à la main On note la solution x^* la solution à l'équation $f(x) = 0$ sur $[1; 2]$.

1) $I_0 = [1; 2]$. Comparer les signes de $f(1) \dots 0$, $f(1.5) \dots 0$ et $f(2) \dots 0$.
   Entourer le bon encadrement de x^* : $1 < x^* < 1.5$

   $1.5 < x^* < 2$.
2) $I_1 = [1; 1.5]$. Comparer les signes de $f(1) \dots 0$, $f(1.25) \dots 0$ et $f(1.5) \dots 0$.
   Entourer le bon encadrement de x^* : $1 < x^* < 1.25$

   $1.25 < x^* < 1.5$.
3) $I_2 = [1.25; 1.5]$.
   a) Quel est le centre m de l'intervalle I_2 ? .....
   b) Comparer les signes de $f(1.25 \dots 0)$, $f(m) \dots 0$ et $f(1.5) \dots 0$
   c) En déduire un encadrement de x^* par un intervalle de largeur 0.125 :

   $$< x^* <$$
4) En poursuivant une dernière fois, donner un encadrement de x^* à 0.0625 près.

   $$< x^* <$$
5) Combien de fois doit-on répéter la procédure pour avoir un encadrement de x^* à 10^{-3} près ? .....

Partie B : l'algorithme en Python

Si l' intervalle de départ est $[a;b]$, l'encadrement obtenu de x^* après n itérations est de largeur $\dfrac{b-a}{2^n}$.

1) Compléter les lignes 1 et 2 pour définir la fonction f.
2) Tester l'algorithme pour un nombre différent d'itérations et donner un encadrement à 10^{-6} de x^*.

```
1  def f(x) :
2      return ........
3  a,b=1,2
4  erreur = 10**-6
5  while b-a > erreur :
6      m = (a+b)/2          # m milieu de l' intervalle [a,b]
7      if f(m)*f(a)<0 :         # f(m) et f(a) de signes contraires
8          b = m            # x* ∈ [a,m], nouveau encadrement
9      else :
10         a = m            # x* ∈ [m,b]
11     print(a,'<x*<',b)    # nouveau encadrement de x*
```

B.3 TP Algorithme de Babylone pour la racine entière

Préliminaire

1) Créer un script python nommé `racineentiere.py` et y saisir le script ci-dessous :

```
1 def encadrement(n) :
2     racine = 1
3 while racine**2 < n:
4     racine = racine + 1
5     return racine - 1, racine # encadrement
```

2) a) Exécuter le script puis dans la console exécuter l'instruction `encadrement(18)`.
   b) Que vérifie la variable `racine` lors de la sortie de la boucle infinie ?
   c) Interpréter le résultat.
3) Exécuter l'instruction `encadrement(1664)` et interpréter le résultat.
4) Que fait cette fonction ?
5) Quel est le temps d'exécution de l'instruction `encadrement(1e10)` ?
6) Quelles sont les limites de cet algorithme ?

Algorithme de Babylone Pour trouver plus rapidement un encadrement de $\sqrt{1664}$ nous allons utiliser l'algorithme de Babylone. On cherchera des paires de nombres entiers a et b tels que :

- $a \times b \approx 1664$.
- l'écart $|a - b|$ est de plus en plus petit.

Si $b < \sqrt{1664} < a$, avec $ab \approx 1664$

À l'étape suivante, on prendra :

- $a' \approx \dfrac{a+b}{2}$, la moyenne de a et b.
- $b' \approx \dfrac{735}{a'}$, afin que $a' \times b' \approx 1664$.

On a $b' < \sqrt{1664} < a'$.

Si l'écart l'écart $|a' - b'|$ est supérieur à 1, on recommence.

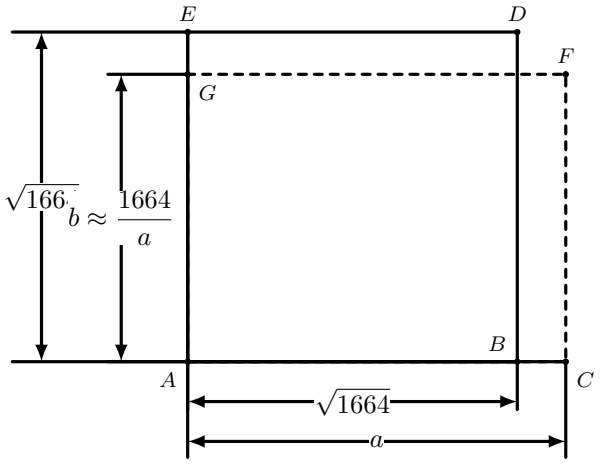


Figure B.1 – Algorithme de Babylone et illustration graphique. Le rectangle $ACFG$ est de même aire que le carré $ABDE$. On a l'encadrement $b < \sqrt{1664} < a$. La moyenne arithmétique $a' = \dfrac{a+b}{2}$ est encore plus proche de $\sqrt{1664}$, idem pour $b' \approx \dfrac{1664}{a'}$

Étape 0 $a_0 = 1664$, et $b_0 = \dfrac{1664}{a_0} = 1$. On a $b_0 < \sqrt{1664} < a_0$ et $a_0 b_0 = 1664$.

Étape 1 $a_1 = \dfrac{1664+1}{2} \approx 832$ et $b_1 = \dfrac{1664}{a_1} \approx 2$. $|a_1 - b_1| > 1$, on continue.
À vous poursuivre et compléter le tableau ci-dessous.

Étape	a	b
$i = 0$	$a_0 = 1664$	$b_0 = 1$
$i = 1$	$a_1 = 832$	$b_1 = 2$
$i = 2$	$a_2 =$	$b_2 =$
$i = 3$	$a_3 =$	$b_3 =$
$i = 4$	$a_4 =$	$b_4 =$
$i = 5$	$a_5 =$	$b_5 =$
$i = 6$	$a_6 =$	$b_6 =$
$i = 7$	$a_7 =$	$b_7 =$

- Dans Python, le texte à droite de `#` est ignoré par l'interpréteur. Il sert de commentaire.
- Depuis Python v3, la division `/` retourne un nombre de type `float` (virgule flottante).
- La division entière est donnée par l'instruction `//`.
- Pour calculer la valeur absolue d'une variable `x`, on utilise l'instruction `abs{x}`

L'algorithme en Python

```
1 def babylone(n) :
2     a , b = n , 1            # démarrage : double affectation
3 while ...                  # poursuivre jusqu'à |a − b| ⩽ 1
4     a = round(......)      # moyenne arithmétique des anciens a, b
5     b = round(......)      # il faut garder a × b ≈ n
6     print(a , b )          # pour vérifier
7     return a , b           #
```

a) Compléter le script de la fonction `babylone()` afin qu'elle retourne deux entiers a et b, d'écart inférieur ou égal à 1. L'une de ses valeurs sera la racine entière de `n`.

b) Rentrer le script sur votre pythonette. Inutile de rentrer les commentaires.

c) Exécuter le script et vérifier que l'instruction `approximation(1664)` retourne `40` et `41`. Vérifier les étapes intermédiaires avec les résultats obtenus dans le tableau précédent.

Une fois qu'on a vérifié le déroulement du script, on peut effacer ou commenter la ligne 6.

B.4 TP Approximation de racines carrées par des rationnels

Prélminaires Exprimer les epxresions suivantes sous forme d'une fraction irréductible. Montrer les étapes de calcul.

$A = 1 + \dfrac{1}{1+\frac{1}{1}}$ $\quad$ $B = 1 + \dfrac{1}{1+\frac{3}{2}}$ $\quad$ $C = 1 + \dfrac{1}{1+\frac{7}{5}}$

$=$ $\quad$ $=$ $\quad$ $=$

$=$ $\quad$ $=$ $\quad$ $=$

Compléter les pointillés par $<$ ou $>$:

$\left|\sqrt{2} - A\right| \ldots 10^{-3}$ $\quad$ $\left|\sqrt{2} - B\right| \ldots 10^{-3}$ $\quad$ $\left|\sqrt{2} - C\right| \ldots 10^{-3}$

Exercice 1 — Rappel.

a) Simplifier $\dfrac{1}{\sqrt{2}+1}$

b) En déduire que $\sqrt{2} = 1 + \dfrac{1}{1+\sqrt{2}}$.

Algorithme L'algorithme suivant vise à obtenir des nombres rationnels $\dfrac{a}{b}$ (a et $b \in \mathbb{N}$), de plus en plus proches de $\sqrt{2}$.

Si à l'étape on a l'approximation $\sqrt{2} \approx \dfrac{a}{b}$. À l'étape suivante on calcule $1 + \dfrac{1}{1+\frac{a}{b}} = \dfrac{a'}{b'}$.

Exercice 2 Mettre sous forme d'une unique fraction simplifiée :

$$1 + \frac{a}{b} =$$

$$1 + \frac{1}{1+\frac{a}{b}} =$$

Si $\sqrt{2} \approx \dfrac{a}{b}$, à l'étape suivante $\sqrt{2} \approx \dfrac{a'}{b'}$ avec $\begin{cases} a' &= a + 2b \\ b' &= a + b \end{cases}$.

Étape 0 Valeurs de départ : $\begin{cases} a_0 &= 1 \\ b_0 &= 1 \end{cases}$ $\quad$ On a $\left|\sqrt{2} - \dfrac{1}{1}\right| \approx 0{,}41 < \dfrac{1}{2b_0^2}$.

Étape 1 $\begin{cases} a_1 &= a_0 + 2b_0 = 1 + 2 = 3 \\ b_1 &= a_0 + b_0 = 1 + 1 = 2 \end{cases}$ $\quad$ On a $\left|\sqrt{2} - \dfrac{3}{2}\right| \approx$

Étape 2 $\begin{cases} a_2 &= a_1 + 2b_1 = \\ b_2 &= a_1 + b_1 = \end{cases}$ $\quad$ On a $\left|\sqrt{2} - \text{——}\right| \approx$

Étape 3 $\begin{cases} a_3 &= a_2 + 2b_2 = \\ b_3 &= a_2 + b_2 = \end{cases}$ $\quad$ On a $\left|\sqrt{2} - \text{——}\right| \approx$

Étape 4 $\begin{cases} a_4 & = a_3 + 2b_3 = \\ b_4 & = a_3 + b_3 = \end{cases}$ On a $\left|\sqrt{2} - \frac{\quad}{\quad}\right| \approx$

L'algorithme en Python

```
1 def approximation(n) :        # n est le nombre d'étapes à faire
2     a , b = 1 , 1             # démarrage : double affectation
3 for i in range(n) :       # i prend les valeurs ...
4     a = a + 2*b
5     b = a + b
6     return a, b
```

a) Montrer que l'instruction `approximation(1)` retourne `a=3` et `b=4`.
Quelle est l'erreur de cet algorithme ?

b) Tester la version modifiée sur votre pythonette et vérifier que l'instruction `approximation(5)` retourne `a=99` et `b=70`.

c) Que retourne `approximation(6)` ? `a=` et `b=`

```
1 def approximation(n) :        # n est le nombre d'étapes à faire
2     a , b = 1 , 1             # démarrage : double affectation
3 for i in range(n) :
4     a , b = a + 2*b  ,  a + b
5     return a, b
6
```

$\frac{99}{70}$ est une approximation rationnelle de $\sqrt{2}$ à 10^{-4} près.
Pour faire mieux, il faut un dénominateur au moins égal à 169 : il s'agit du rationnel $\frac{239}{169}$.

B.5 TP Algorithmes de test de primalité

- l'instruction `a//b` retourne le quotient de la division entière de a par b.
- l'instruction `a%b` retourne le reste de la division entière de a par b.

L'objectif est de comprendre le fonctionnement des différents scripts par une lecture attentive. Les programmer sur une calculatrice peut servir de vérification, mais n'est ni nécessaire ni suffisant.

Exercice 3

```
1  def fonction(n) :
2      x = 0
3      for i in range(1,n):
4          if n%i == 0 :
5              x = x+1
6      return x
```

```
7  def estpremier(n) :
8      p = fonction(n)
9      if ....
10         return True
11     else :
12         return False
```

1) On utilise l'instruction `fonction(12)`. Faire tourner le script à la main, et vérifier que la valeur retournée est 5. Préciser :
   a) le nombre de répétitions de la boucle `for`.
   b) pour chaque répétition `i`, la valeur de `x` à la fin.
2) Lancer à la main l'instruction `fonction(5)` et `fonction(6)`, et préciser la valeur retournée.
3) On suppose que `fonction(n)` est un entier $\geqslant 2$. Corriger la ligne 3 afin que l'instruction `fonction(n)` retourne le nombre de diviseurs de `n`.
4) Comment utiliser l'instruction `fonction(n)` pour déterminer si `n` est premier ?
5) Compléter la ligne 9 afin que l'instruction `estpremier(n)` retourne `True` si `n` est premier, et `False` dans le cas contraire.

Exercice 4

```
1  def fonction(n) :
2      i = 2
3      while n%i !=0 :
4          i = i + 1
5      return i
6
```

```
7  def estpremier(n) :
8      p = fonction(n)
9      if ......
10         return True
11     else :
12         return False
```

1) Faire tourner le script à la main, et vérifier que l'instruction `fonction(7)` retourne `7`.
2) Que retourne l'instruction `fonction(12)` ? `fonction(35)` ?
3) Que retourne l'instruction `fonction(n)` lorsque `n` est un nombre entier supérieur ou égal à 2 ?
4) Comment utiliser l'instruction `fonction(n)` pour déterminer si `n` est premier ?
5) Compléter la ligne 9 afin que l'instruction `estpremier(n)` retourne `True` si `n` est premier, et `False` dans le cas contraire.

Exercice 5

1) Expliquer pourquoi `10101` n'est pas premier.
2) Comparer le nombre de boucles nécessaire à chacun des algorithmes des exercices 3 et 4 pour identifier que `10101` n'est pas premier. Quel algorithme est le plus rapide ?
3) Le script ci dessous `estpremier()` ci-dessous est supposé retourner `True` si la variable en entrée `n` est un nombre premier et `False` sinon.

```
1 def estpremier(n) :          # n est un entier supérieur ou égal à 1
2     x = 2
3 while x**2 < n :         # on cherche des diviseurs inférieurs à √n
4     if n%x== 0 :
5         return ...
6     else :
7         x = x + 1
8         return ...
```

On rappelle que l'exécution de la fonction s'arrête à l'exécution d'une instruction `return`.

a) Faire tourner le script avec l'instruction `estpremier(15)`.
Combien de fois la boucle `while` est exécutée ?
Que vaut la variable `x` à l'arrêt de la fonction ?
b) Mêmes questions, cette fois on utilise l'instruction `estpremier(15)`.
c) Compléter les lignes 5 et 8 par `True` ou `False`.

4) Expliquer pourquoi il n'est pas nécessaire de chercher des diviseurs au delà de $\sqrt{n}$.
5) Faire tourner (à la main) le scipt avec l'instruction `estpremier(9)` puis `estpremier(16)`. Que constatez vous ?
6) Corriger la ligne 3, afin de remédier à l'erreur de la question précédente.

Exercice 6 — Réfuter une conjecture à l'aide d'un algorithme.

Pierre de Fermat (* 1605, † 1665) pensait que tous les entiers $F_n = 2^{(2^n)} + 1$ étaient des nombres premiers. Effectivement $F_0 = 3$, $F_1 = 5$ et $F_2 = 2^{(2^2)} + 1 = 17$ sont des nombres premiers.

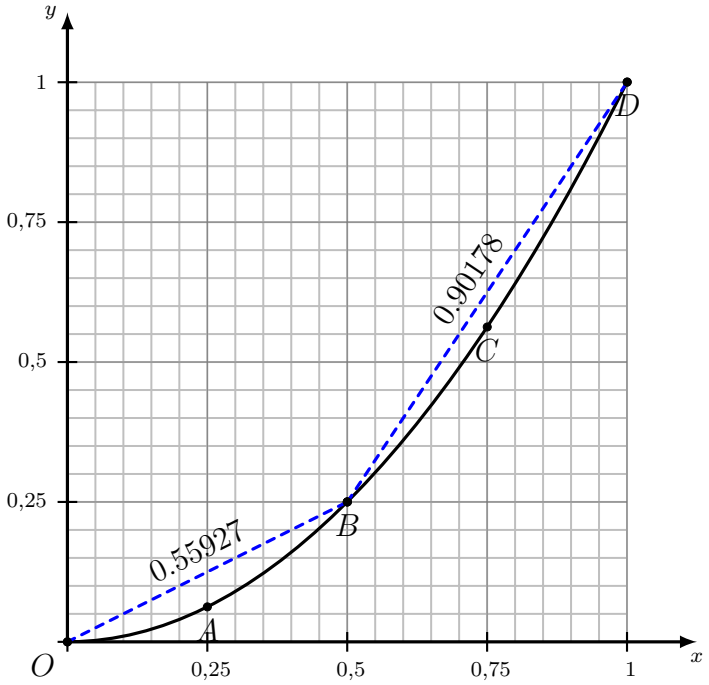
À l'aide de l'algorithme final de l'exercice 5 , et en complétant le script ci-dessous, trouve le plus petit entier F_n qui n'est pas premier.

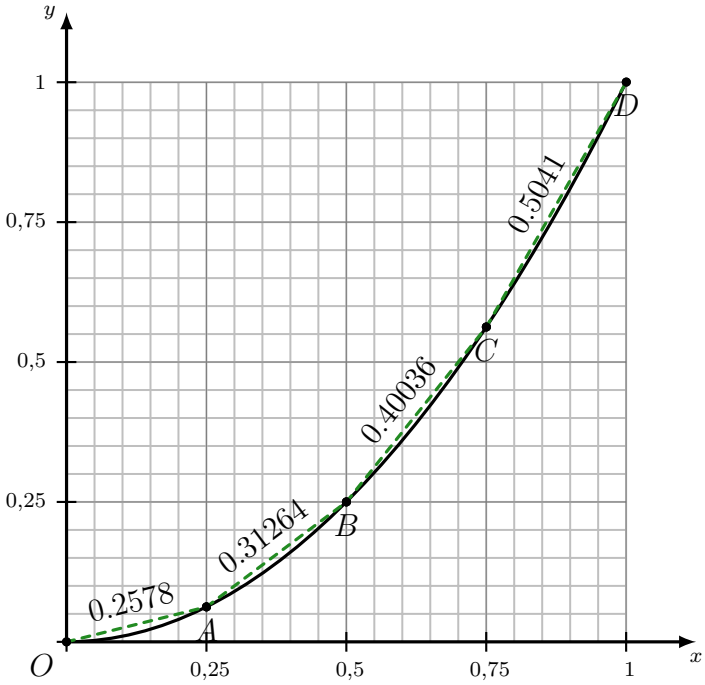
```
10 n , F = 0 , 3
11     while ....
12     n = n + 1
13     F =
14     print(n)
```

B.6 TP Algorithme de calcul de la longueur d'une courbe

Soit la fonction carré $f : [0; 1] \to \mathbb{R}$ représentée par la courbe $\mathscr{C}_f$ dans un repère orthonormé d'origine O.
$x \mapsto x^2$

L'objectif du problème est d'estimer la longueur du tracé. Pour cela on approche la courbe par une ligne brisée.





1) Avec 2 segments :
   a) Quelles sont les coordonnées des points O, B et D ?

   b) Calculer la distance OB de façon exacte et comparer avec l'affichage du logiciel.

   c) Estimer la longueur de la ligne brisée OBD obtenue, tracée en bleu sur la figure.

2) Avec 4 segments :
   a) Quelles sont les coordonnées des points A et C ?

   b) Calculer la distance AB de façon exacte et comparer avec l'affichage du logiciel.

   c) Estimer à l'aide des valeurs données par le logiciel la longueur de la ligne brisée $OABCD$ obtenue, tracée en vert sur la figure.

3) On souhaite automatiser le calcul pour pouvoir prendre plus de points et mieux approcher la courbe.
   a) P et Q sont les points de $\mathscr{C}_f$ d'abscisses a et b. Donner les coordonnées de P et Q.

b) Exprimer la longueur PQ en fonction de a et b.

c) Compléter la ligne 4 du script ci-dessous afin que la fonction d'appel `d(a,b)` retourne la longueur du segment droit PQ, ou P et Q sont les points de $\mathscr{C}_f$ d'abscisses a et b.

d) On exécute l'instruction `long(4)`. Compléter le tableau en indiquant des valeurs approchées des contenus successifs des variables `L` et `x`...

L	0						
x	0						
$x+1/n$	0						
$x+1/n \leqslant 1$	Vraie						

4) Saisir le script sur sa Pythonette ou sur `https://basthon.fr/`. Que retournent les instructions :

a) `long(4)=`

b) `long(1000)=`

c) `long(10000)=`

```
1  from math import sqrt
2  def d(a,b):
3      distance = ...
4      return distance
5
6  def long(n):
7      L = 0
8      x = 0
9      while x + 1/n <= 1 :
10         L = L + d(x,x+1/n)
11         x = x + 1/n
12     return L
```