Stage Laboratoire Mathématiques

Lycée Jeanne d'Abret

Participant-e-s : Christine RUFFAS, Véronique PASCAUD, Gaetan AUBRY, Alain SEBAOUN.
Intervenant Université d'Evry Nicolas MEUNIER

Compte rendu Activité Laboratoire Mathématiques Jeanne d'Albret : Stage Nicolas MEUNIER
Dans le cadre du Laboratoire de Mathématiques du Lycée Jeanne d'Albret, un stage sur trois jours, les lundis 13 mai, 20 mai et 3 juin, dont le thème initial était

Mathématiques-Biologie-Algorithmique-Python Liaison Enseignement-Recherche

a été organisé . L'intervenant était Monsieur Nicolas MEUNIER de l'Université d'Evry. Le groupe concerné par ce stage était formé de quatre professeurs du lycée.

Déroulement des journées de stage :
La première séance (lundi 13 mai) a permis la prise en main rapide de Python et de ses principes généraux sur la base d'un document (fourni par Mr MEUNIER) qui reprend les thèmes usuels que l'on rencontre dans l'enseignement secondaire, à savoir :

- Variables, Affectations
- Instructions conditionnelles
- Opérations mathématiques de base (+ , − , ∗ , / , puissances, artithmétique))
- Modules diers, (math, random, mathplotlib , etc ...)
- Boucles itératives
- Fonctions en code Python
- Graphiques

Chacun des participants pouvait organiser sa séance en fonction de son niveau personnel et de ses compétences propres et le choix des exercices à traiter donnés dans le document était libre. Très vite, les participants ont montré une très grande aisance dans la manipulation de Python ce qui a permis d'aborder ensuite des thèmes d'un niveau plus élevé et au contenu mathématique plus affirmé correspondant au thème initial du stage.

Exemples de thèmes traités relevant du secondaire :

- Nombres parfaits, Nombres Aimables et Suites Aliquotes :
  Sur la base d'un document (Article de Jean-Paul DELAHAYE, Pour la science) , étude des nombres aimables et suites aliquotes, mise en place algorithmique, mise en place sous Python.
- Probabilité : Simulations sur la base de la loi uniforme
- Suites de Syracuse
- Méthode de Monté-Carlo.
- Arithmétique.
- Suites, Sommation, Recherche de seuil.
- Calcul matriciel

Exemples de thèmes traités au-dessus du secondaire

- Modélisation.
- Equations différentielles aux dérivées partielles.
- Méthodes de résolutions numériques.
- Marches aléatoires.

Quelques activités sont exposées dans la suite de ce document. Il s'agit d'une liste non exhaustive qui est là pour indiquer que la connaissance de Python et son usage dans un cadre d'enseignement secondaire est assez bien maîtrisé.

Compte Rendu de Mr MEUNIER

Depuis plusieurs années la modélisation mathématique et numérique de problèmes issus des sciences du vivant est une thématique en plein essor. Les équations aux dérivées partielles sont utilisées pour construire des modèles pouvant expliquer (et prédire) des phénomènes intéressants des biologistes et des médecins.
Les trois projets proposés lors de ce laboratoire de mathématiques ont pour objet la construction et l'étude de modèles mathématiques pour décrire la polarisation et la migration cellulaire, et le fonctionnement du poumon. En particulier, ils visent à rendre compte de différents comportements observés expérimentalement. La polarisation et la migration cellulaire sont des phénomènes physiologiques à la fois courants et fondamentaux. Des dysfonctionnements dans la capacité des cellules à se polariser ou à migrer peuvent provoquer ou entretenir des phénomènes physiologiques pathologiques : lors d'un cancer (métastase) ou de maladies inflammatoires (comme par exemple l'athérosclérose) ainsi que dans la réponse immunitaire (extravasation des leucocytes). Comprendre les facteurs-clés de ces phénomènes est donc un enjeu crucial.

- Dans le premier projet l'accent est mis sur la description mathématique et numérique de la dynamique de protéines spécifiques à la polarisation en utilisant une équation aux dérivées partielles en dimension 1 de type convection-diffusion non linéaire. L'originalité de ce modèle est sa capacité à décrire la boucle positive entre la présence de ces protéines sur la membrane de la cellule et la polymérisation de l'actine et donc l'afflux de nouvelles protéines sur la membrane. D'un point de vue mathématique la polarisation a lieu lorsque la solution d'une équation différentielle ordinaire devient infinie en temps fini.

- Dans le second projet on s'intéresse à la formation d'agrégats de globules rouges lorsque ceux-ci interagissent avec des forces dérivant de potentiels. D'un point de vue mathématique il s'agit d'étudier mathématiquement et numériquement les trajectoires de particules supposées ponctuelles à l'aide d'un système d'équations différentielles ordinaires couplées. L'agrégation a lieu lorsque les trajectoires des globules se concentrent autour d'une position centrale.

- Dans un troisième projet il s'agit d'étudier mathématiquement et numériquement le transport de l'air dans l'arbre bronchique. L'arbre bronchique est modélisé par un arbre dyadique ayant 19 générations. D'un point de vue mathématique il s'agit d'étudier certaines caractéristiques d'un arbre dyadique (distance, limite continue lorsque le nombre de générations tend vers l'infini)

Compte rendu du stage Python avec Nicolas Meunier

Le stage qui s'est tenu sur trois journées au lycée dans le cadre du Laboratoire de matématique fut une initiative des plus intéressantes.

Le thème choisi, le langage Python, était parfaitement en adéquation avec les nouvelles orientations des programmes du secondaire. Il y avait là une bonne occasion de consolider nos connaissances dans la programmation.

La première fut justement consacrée à assurer un niveau homogène des connaissances de Python parmi les participants, ce qui fut grandement facilité par la qualité des documents et des explications fournis par Nicolas Meunier. La progression des exemples à traiter ainsi que le contenu de plus exigeant en mathématique auguraient bien des deux autres journées, à savoir traiter des problèmes utilisant des connaissances de mathématique post-bac sur des sujets aux nombreux intérêts.

La deuxième journée fut consacrée à la programmation, en Python, de l'état de l'évolution d'un mélange gaz-liquide afin de voir apparaître ou non des convergences vers des états d'équilibre. Le liquide, incompréssible, figurant les hématies et le gaz, compréssible, figurant le liquide dans lesquelles elles se déplacent, nous avions là l'occasion de voir si des regroupements, interprétés comme des caillots, se formaient à tout coup ou non. Bien que la simplification soit importante face à la complexité de la réalité, nous pouvions toucher tout à la fois des mathématiques, une application de ces dernières ainsi que de la programmation Python.

Il fut d'ailleurs intéressant de voir qu'un programme qui semblait être plus facilement abordé à l'aide de listes se révéla finalement plus simple à concevoir en recourant aux matrices. Nous apprenions les limites et les possibilités offertes par Python.

La troisième et dernière journée permit de poursuivre et achever le travail débuté lors de la deuxième. Nicolas Meunier a su nous présenter des sujets d'étude nombreux et variés qui nous ont permis de réaliser l'actualité de l'application des mathématiques en biologie et en médecine. Bien qu'il ait réduit la complexité de la réalité de son travail pour la mettre à notre portée, il nous a permis d'approcher des territoires encore inexplorés des mathématiques, des domaines dont la richesse est telle qu'il n'y existe pas encore de théorèmes généraux permettant de répondre rapidement à de nombreuses questions.

Nicolas Meunier, par son ouverture d'esprit, sa gentillesse et son enthousiasme, nous a permis d'aborder, faire et apprécier des mathématiques venant du Supérieur et de la Recherche.

Cette expérience est sans aucun doute à renouveler.

Mesdames Christine RUFAS et Véronique PASCAUD

Laboratoire de mathématique du lycée Jeanne d'Albret

Projet "agglomérats d'hématies"

Après une première journée permettant de revoir les principales notions de Python dont nous aurions besoin par la suite, Monsieur Meunier nous a proposé plusieurs projets, liant les mathématiques, la SVT et Python. Nous (Mme Pascaud et Mme Rufas) avons choisi celui portant sur les agglomérats d'hématies.

1) Description rapide du projet

Il s'agit de modéliser de façon simplifiée la formation d'agglomérats d'hématies , en considérant une situation monodimensionnelle .

On considère donc une population de $N+2$ hématies sur l'intervalle $[0\,;1]$, chacune subissant l'influence de ses deux voisines, sauf la première et la dernière, qui seront donc supposées fixes en 0 et 1 respectivement.
On s'intéresse alors à la position du $i^{ème}$ globule rouge $x_i(t)$ à l'instant t.
La modélisation consiste à trouver une fonction "simple" qui corresponde à la situation souhaitée (signe, variations, limites).
On nous a donc proposé le système dynamique suivant pour décrire le comportement des hématies :

$$\ddot{x}_{_i} = \varphi(x_{i+1} - x_i) - \varphi(x_i - x_{i-1}) - \lambda \dot{x}_{_i} \qquad \forall i \in \{1, \ldots, N\}$$

avec la fonction $\varphi : d \longmapsto \dfrac{\gamma}{d} \ln\left(\dfrac{d}{a}\right)$ où $\varphi(x_{i+1} - x_i)$ représente la force exercée par la particule $i+1$ sur la particule i (et l'opposé de la force exercée par i sur $i+1$)

2) Vérifications mathématiques

Afin de nous "approprier " la situation et les notations, nous avons pris le temps de vérifier deux égalités indiquées dans le document de travail fourni par Monsieur Meunier (en utilisant notamment une primitive Φ de φ et des développements de Taylor à l'ordre 2).
Ce qui permet alors de valider le schéma suivant écrit sous forme vectorielle :

Soit $x^k = (x_1^k, \ldots, x_n^k)$ une approximation du vecteur des positions au temps t_k

$$x^{k+1} = 2x^k - x^{k-1} + h^2\Phi(x^k) - (x^k - x^{k-1})$$

avec $k \geq 1$ et les vecteurs de conditions initiales x^0 et x^1 donnés
($t_k = kh$ avec $h = \dfrac{T}{K}$ ainsi $t_{k+1} - t_k = h$)

3) Application

On applique le schéma numérique ci-dessus à la situation de $N = 49$ particules libres disposées initialement de façon aléatoire en prenant comme vecteur initial $X = (X_1, \ldots, X_N)$ défini par

$$X_i = (i + 0.1\rho_i)(\delta x)$$

avec $\delta x = 50$, ρ_i réalisation d'une variable aléatoire de loi uniforme sur $[-1; 1]$, $K = 1500$, $h = 0,002$,
$T = Kh = 3,0$, $\gamma = 0,5$, $\lambda = 10$ et $a = 0,004$.

4) Utilisation de Python

Dans un premier temps nous avons cherché à écrire la relation encadrée sous forme matricielle.

Nous avons donc découvert et utilisé la bibliothèque Numpy qui permet de gérer des tableaux de nombres :

- Avec le package numpy, on peut alors utiliser le type array qui est similaire à une liste avec la condition supplémentaire que tous les éléments sont du même type.
- La fonction np.eye(n) permet de créer I_n matrice carrée identité de taille n

Nous avons donc commencé à définir les matrices entrant en jeu sous la forme suivante :

```
from random import*
import numpy as np
def X0(i) :
    return (i + 0.1*uniform(-1, 1))/50
X0 =np.array([X0(i) for i in range(1, 51)]
h = 0.002
ld = 10
A = (2 - ld * h)*np.eye(50)
B = (1 - ld * h)*np.eye(50)
```

Mais ensuite, Monsieur Meunier nous a rapidement orientées vers l'utilisation de la fonction linspace.
En effet, il n'était pas indispensable de faire intervenir des matrices. Il suffisait donc d'utiliser des vecteurs.
Nous avons alors découvert et utilisé la fonction linspace (disponible elle aussi avec le package numpy).
linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments.
Il restait ensuite à tenir compte du fait que la première et la dernière hématie seront toujours fixes, nous avons utilisé la fonction "concatenate" .

5) Bilan

Certes, nous n'avons pas eu le temps d'aboutir totalement après deux journées consacrées à ce projet (changements de méthode, plus quelques erreurs en cours de route) .
Mais nous avons vraiment apprécié de pouvoir nous plonger dans le début de ce travail de recherche en mathématiques en lien avec les SVT proposé par Monsieur Meunier.
En effet, tout d'abord, nous avons pu prendre le temps d'essayer de comprendre la modélisation en un système dynamique du problème posé dans ce travail de recherche. Cela a été l'occasion d'échanges particulièrement enrichissants avec M. Meunier.
Ensuite, la vérification des différentes étapes du début de résolution du système d'équations aux dérivées partielles nous a pris un certain temps de relecture mais nous a aussi permis de revoir des notions universitaires et de retrouver le goût de l'effort dans un problème mathématique sans solution immédiate.
Pour finir, nous avons utilisé et renforcé nos compétences en Python et avons pris conscience que c'est un véritable outil au service des mathématiques.

Somme des diviseurs propres, Nombres parfaits, Suites aliquotes

Stage Laboratoire Mathématiques, Mai-Juin 2019

Alain SEBAOUN , Lycée Jeanne d'Albret

Somme des diviseurs propres, Nombres Parfaits
Pour $n \in \mathbb{N}^*$, notons $\mathcal{D}(n)$ l'ensemble des diviseurs propres de n et $S(n)$ la somme des diviseurs propres de n.

$$\forall n \in \mathbb{N}, \quad S(n) = \sum_{d \in \mathcal{D}(n)} d \quad (1)$$

Par exemple, $S(1) = 0$, $S(2) = 1$, $S(4) = 3$, $S(5) = 1$, $S(6) = 6$.

Un entier naturel non nul est dit Parfait s'il est la somme de ses diviseurs propres.

$$n \in \mathbb{N}^* \text{ est Parfait si et seulement si } S(n) = n$$

Par exemple, 6 est Parfait car $S(6) = 1 + 2 + 3 = 6$. 10 n'est pas Parfait car $S(10) = 1 + 2 + 5 = 8 \neq 10$

La première question à résoudre est donc de construire une fonction sous Python permettant le calcul de $S(n)$. En reprenant la définition "brute" de $S(n)$, on est conduit à décomposer ce problème de la façon suivante.

1. Déterminer si un entier d divise ou pas un entier n.
2. Calculer la somme des diviseurs propres d'un entier n.
3. Savoir si n est Parfait ou pas, cela revient à comparer cette somme avec n. Dans le cas où $S(n) = n$, n est un nombre Parfait.

Pour le premier point, le problème revient simplement à connaître le reste dans la division euclidienne de n par d. L'instruction % donnant ce reste, une fonction renvoyant "TRUE" ou "FALSE" suivant que d divise n ou pas peut alors s'écire.

```
def divise(n,d) :
    return (n%d == 0)
```

Pour le second point, une simple boucle utilisant la fonction précédente permet le calcul de $S(n)$.

Méthode (1) de calcul de $S(n)$

```
def S1(n) :
    s=0
    for d in range(1,n) :
        s=s+d*divise(n,d)
    return s
```

Maintenant, on peut écrire une fonction qui retourne "TRUE" ou "FALSE" suivant que n soit parfait ou pas.

```
def Parfait(n) :
    return S(n)==n
```

Par cette méthode, il n'y a pas besoin de connaître autre chose que la définition d'un diviseur de n pour calculer $S(n)$. Mais il faut tout de même remarquer qu'elle est loin d'être optimale si l'on considère le nombre de calculs et le temps nécessaire à l'obtention d'une valeur comme $S(2358777777)$. Pour les *petites valeurs* de n, elle est acceptable et permet surtout d'effectuer un controle *à la main* du résultat obtenu. Un élève sans aucune connaissance en arithmétique, autre que la notion de diviseur entier, peut la mettre en place pour écrire la fonction S sur Python.

Néanmoins, en reconsidèrant la question et en mettant en avant le rôle symétrique dans le cas d'une division euclidienne à reste nul, entre diviseur et quotient, il est possible d'améliorer le temps de calcul de $S(n)$ en faisant de simples constatations.

Méthode (2) : On limite la recherche des diviseurs aux entiers $\leq \sqrt{n}$

a est un diviseur de n si et seulement si il existe $b \in \mathbb{N}^*$ tel que $n = a \times b$. Dans ce cas, b est aussi un diviseur de n. Dans le calcul de $S(n)$, on peut alors faire jouer un rôle symétrique entre a et $\frac{n}{a}$ si a divise n. Et comme on peut toujours supposer que $a \leq b$, donc que $a^2 \leq n$, on peut limiter le test "d divise n" aux entiers $d \leq \sqrt{n}$.
Et considèrer que la somme $S(n)$ peut s'écrire :

$$S(n) = \left(\sum_{d \leq \sqrt{n}, d \in \mathcal{D}(n)} (d + \frac{n}{d}) \right) - n$$

En première lecture, tout semble correct ! Mais en relisant cette écriture de $S(n)$, on peut voir que si n est un carré dans $\mathbb{N}$, la valeur retournée par cette "expression" de $S(n)$ n'est pas exacte !
Exemple, $S(4) = 1 + 2 = 3$, mais $\sum_{d \leq \sqrt{4}, d \in \mathcal{D}(4)} (d + \frac{4}{d}) - 4 = (1 + 4) + (2 + 2) - 4 = 5$.
Il faut donc prendre en compte le cas où n est un carré dans $\mathbb{N}$ ou non, dans la formule et écrire :

$$S(n) = \sum_{d < \sqrt{n}, d \in \mathcal{D}(n)} (d + \frac{n}{d}) - n + \begin{cases} \sqrt{n} & \text{si } n \text{ est un carré} \\ 0 & \text{sinon} \end{cases} \quad (2)$$

Le calcul de $S(n)$ peut alors être fait de la façon suivante :

Méthode (2) de calcul de $S(n)$

```
def S2(n) :
    s=0
    d=2
    while d**2 < n :
        s=s+(d + n//d)*divise(n,d)
        d=d+1
    s=s+ d*divide(n,d)
    return (s+1)*(n !=0)
```

On peut estimer que, si la méthode (1) demande un temps de calcul proportionnel à n, la méthode (2) ne demande qu'un de temps de calcul proportionnel à $\sqrt{n}$. Le gain temporel en passant de (1) à (2) devient très vite "constatable" si on effectue le calcul de $S(n)$ pour n *assez grand*. Les élèves peuvent alors voir que la méthode choisie pour effectuer un calcul est un élément à prendre en considération.

Méthode (3) : Calcul de $S(n)$ en utilisant la décomposition en facteurs premiers

La connaissance de la décomposition en facteurs premiers permet d'avoir une expression directe de $S(n)$.
C'est la "formule" suivante, en notant $\mathcal{P}$ l'ensemble des nombres premiers,

$$\text{Si } n = \prod_{p_i \in \mathcal{P}} p_i^{\alpha_i} \text{ est la décomposition en facteurs premiers de } n, \text{ Alors : } S(n) = \prod_{p_i \in \mathcal{P}} \frac{p_i^{\alpha_i + 1} - 1}{p_i - 1} - n \quad (3)$$

Pour pouvoir obtenir la fonction S en partant de cette "formule", on doit alors mettre en place plusieurs plusieurs points :

- Obtenir pour un entier n, le plus petit nombre premier divisant n
- Pour p premier divisant n, obtenir l'exposant α (sa valuation !) dans la décomposition en facteur premier.
- Définir une fonction qui donne $\frac{p^{a+1} - 1}{p - 1}$ en fonction de p et a.
- Calculer $S(n)$ en utilisant une boucle.

Voilà alors une suite de fonctions permettant de calculer $S(n)$ en utilisant $S(n) = \prod_{p_i \in \mathcal{P}} \frac{p_i^{\alpha_i + 1} - 1}{p_i - 1} - n \quad (3)$

DiviseInf renvoie, si $n > 1$, le plus petit nombre premier divisant n et 0 si $n = 1$.

```
def DiviseurInf(n) :
    d=2
    while d**2<n and n%d!=0 :
        d=d+1
    return (n*(n%d!=0)+d*(n%d==0))*(n!=1)
```

Valuation renvoie le plus grand nombre α tel que p^α divise n.

```
def Valuation(n,p) :
    v=0
    N=n
    while N%p==0 :
        v=v+1
        N=N//p
    return v
```

Sp renvoie $\dfrac{p^{a+1}-1}{p-1}$

```
def Sp(p,a) :
    return (p**(a+1)-1)//(p-1)
```

S3 renvoie la somme des diviseurs propres de n en utilisant la "formule" (3)

```
def S3(n) :
    s=1
    N=n
    while N!=1 :
        p=DiviseurInf(N)
        v=Valuation(N,p)
        s=s*Sp(p,v)
        N=N//p**v
    return s-n
```

On peut alors réécrire la fonction indiquant si un nombre est parfait ou pas de la façon suivante :

```
def Parfait(n) :
        return S3(n)==n
```

Deux petites questions se posent alors !
- De ces trois méthodes, laquelle est la plus simple à comprendre ?
- Laquelle est la plus efficace ?

De toute évidence, la première est la plus simple à comprendre dans la mesure où le calcul de S(n), dans ce cas, revient à un simple test de division.
La méthode (2) semble plus "compliquée" mais le temps de calcul est nettement moins important que celui demandé pour la méthode (1).
La méthode (3) est la plus élaborée mais demande une connaissance plus précise des notions d'arithmétique.
Un test très simple, par exemple en prenant $n = 2^5 * 3^{10} * 17^{10} = 533756191302432$ montre que le calcul de $S(n)$ est trop long avec la méthode (1) et quasi instantané avec la méthode (2) et sans attente avec la méthode (3).
On peut bien sûr tenter d'améliorer la méthode (1), mais in fine, chercher les diviseurs de n revient à chercher les diviseurs premiers de n. On peut d'ailleurs voir ici le lien entre la méthode du crible d'Erathostène et avec la méthode (3). La connaissance des nombres premiers inférieurs à N permet d'avoir une méthode quasi-optimale pour déterminer $S(n)$ pour $n \leq N$.

Si on peut résumer, la méthode (1) qui repose sur la définition "brute" de $S(n)$ est simple mais naïve.
La méthode (2), à peine plus compliquée, est plus performante que la méthode (1).

La méthode (3) la plus performante et la mise en mémoire de la liste des nombre premiers en utilisant le crible d'Erasthotène apporte en plus un gain certain

Liste non exhaustive de questions prologeant le calcul de $S(n)$

— **Liste des nombres parfaits inférieurs à un certain seuil M**

```
def ListeSeuil(M) :
  l=[  ]
  k=1
  while k<=M :
  if Parfait(k) :
    l.append(k)
    k=k+1
  return l
```

— **Liste des $S(n)$ pour n inférieurs à M**

```
def ListeS(M) :
  l=[  ]
  k=1
  while k<=M
    l.append(S(k))
    k=k+1
  return l
```

— **Ensemble des $S(n)$ pour n inférieurs à M**

```
def EnsembleS(M) :
  l=set( )
  k=1
  while k<=M
    l.add(S(k))
    k=k+1
  return l
```

— **Moyenne des $S(n)$ pour $n \leq M$**

```
def MoyenneS(M) :
  l=0
  k=1
  while k<=M
    l=l+S(k)
    k=k+1
  return l/(k-1)
```

— **Recherche des antécédents $\leq M$ de n par S**

```
def AntecedentS(n,M) :
  k=1
  l=set()
  while k<=M :
    if S(k)==n :
        l.add(k)
    k=k+1
  return l
```

Pour tous ces exemples, on pourra comparer le temps de calcul pour obtenir le résultat demandé suivant que l'on utilise les méthode (1) , (2) ou (3) pour calculer $S(n)$.

Et de conclure que les formules les plus "compliquées" à écrire peuvent être d'une très grande utilité dans la pratique.

— **Amélioration de la méthode (3) :**

On pose DiviseurInfSup la fonction écrite sous Python suivante :

```
def DiviseurInfSup(n,p) :
    d=p+1
    while d**2<n and n%d!=0 :
        d=d+1
    return (n*(n%d!=0)+d*(n%d==0))*(n!=1)
```

On écrit alors la fonction Python suivante S4.

```
def S4(n) :
    s=1
    N=n
    P=1
    while N!=1 :
        p=DiviseurInfSup(N,P)
        v=Valuation(N,p)
        s=s*Sp(p,v)
        N=N//p**v
        P=p
    return s-n
```

Question : Expliquer en quoi la fonction S4 calcule bien $S(n)$.

Nombres Aimables ou Amiables
Dans l'étude de la fonction S, les notions d'image et d'antécédent sont bien présentes. Un nombre Parfait n'est rien d'autre qu'un point fixe ou invariant par S, c'est à dire une solution de l'équation :

$$n \in \mathbb{N}^*, \quad S(n) = n$$

En partant d'une valeur initiale $u_0 = a \in \mathbb{N}$, on peut aussi définir la suite récurrente (u_n) par :

$$u_0 = a \in \mathbb{N}^*, \quad \forall n \in \mathbb{N}^*,\ u_{n+1} = S(u_n)$$

Définition Suite Aliquote
Une suite (u_n) est dite *Aliquote* si et seulement si $\forall n \in \mathbb{N}^*$, $u_{n+1} = S(u_n)$

A priori, cette suite n'est pas définie sur $\mathbb{N}^*$. Effectivement, le calcul "récurrent" de celle-ci s'arrête dès que l'on obtient u_n tel que $S(u_n) = 0$.
Par exemple en prenant $u_0 = 12$, on a les termes suivants :

$$u_1 = 12, u_2 = 16, u_3 = 15, u_4 = 9, u_5 = 4, u_6 = 3, u_7 = 1, u_8 = 0$$

On peut remarquer que la suite (u_n) est constante si et seulement si u_0 est un nombre Parfait.
Parmi toutes les suites récurrentes définies de cette façon, on peut alors chercher celles qui sont périodiques, et en particulier, celles de période 2, autrement dit, les suites dont la condtion initiale conduit à :

$$\forall n \in \mathbb{N}^*, \quad u_{n+2} = u_n$$

Cela conduit à déterminer les entiers a et b distincts vérifiant :

$$S(a) = b \quad \text{et} \quad S(b) = a$$

Un tel couple d'entiers (a , b) est dit formé d'entiers *Aimables* ou *Amiables*. On dit aussi que a et b sont une paire d'entiers amiables.
Par exemple, 284 et 220 sont Amiables car S(220) =284 et S(284) = 220.

Définition Suite Aliquote
Une suite (u_n) est dite *aliquote* si et seulement si $\forall n \in \mathbb{N}^*$, $u_{n+1} = S(u_n)$

On s'intéresse ici au cas où la suite aliquote est périodique. Si on pose que l'on appelle Période le plus petit entier N non nul vérifiant $u_{n+N} = u_n$, on peut dire que :

- La Période de (u_n) est $N = 1$ si et seulement si u_0 est Parfait.
- La Période de (u_n) est $N = 2$ si et seulement si u_0 et u_1 sont une paire d'entiers Amiables.
- Si la suite (u_n) est périodique et si N est la Période de cette suite, on dit aussi que la suite u_0, u_1, ... u_{N-1} forme une chaîne sociable d'ordre N.
- On peut aussi s'intéresser aux entiers a tels que la suite (u_n) avec la condition initiale $u_0 = a$ soit périodique à partir d'un certain rang.

Bref, le domaine dans ces questions n'est pas limité et sujet à toutes sortes de recherches. A titre d'exemple, on peut chercher les paires de nombres Amaibles inférieurs à N. Pour cela, il suffit d'écrire une fonction qui indique si les conditions $S(a) = b$, $a \neq b$ et $S(b) = a$ sont remplies et "stocker" les résultats dans une liste ou un ensemble, suivant ce que l'on veut obtenir comme résultat.
A titre d'exemple, voici quelques fonctions Python sur le thème des nombres Amiables ou les chaînes "sociables" de longueur N.

1. **Fonction Simple qui teste si la paire d'entiers distincts (a, b) est Amiable**

```
def TestAmiable(a,b) :
    return (a==S(b) and b==S(a))
```

2. **Fonction qui teste si a est le premier terme d'une chaîne sociable de longueur $\leq$ N**

```
def TestSociable(a,N) :
    aa=S(a)
    L=[a]
    k=1
    while aa!=0 and k<N and not(aa in L) :
      L.append(aa)
      aa=S(aa)
      k=k+1
    return L*(a==aa)
```

3. **Fonction qui retourne la liste des entiers $a \leq n$ tels que a soit le premier terme d'une chaîne sociable de longueur $\leq N$ et l'ordre de cette chaîne**

```
def ListDebutSociable(n,N) :
    L=[]
    for a in range(1,n+1) :
      l=len(TestSociable(a,N))
      if l!=0 :
        L.append([a,l])
    return L
```

A titre d'exemple, le lancement de **ListDebutSociable(16000,10)** retourne :

```
[[6, 1], [28, 1], [220, 2], [284, 2], [496, 1], [1184, 2], [1210, 2], [2620, 2], [2924, 2], [5020, 2], [5564, 2], [6232, 2], [6368, 2],
    [8128, 1], [10744, 2], [10856, 2], [12285, 2], [12496, 5], [14264, 5], [14288, 5], [14536, 5], [14595, 2], [15472, 5]]
```

On retrouve les cinq entiers 12496, 14288, 15472, 14536, 14264 formant une chaîne sociable d'ordre 5 suivants indiqués dans le document de J-P DELAHAYE (page 100).

Codage RSA

Stage Laboratoire Mathématiques, Mai-Juin 2019

Alain SEBAOUN , Lycée Jeanne d'Albret

La méthode de codage RSA, inventé en 1978 par Rivest, Shamir et Adleman *d'où les initiales ...RSA*, est toujours considérée comme un des codages de cryptographie les plus performants. L'idée générale repose sur le Petit Théorème de Fermat, la notion d'inverse modulo n et la fonction Indicatrice d'Euler.

Petit Théorème de Fermat, Pour rappel ....
Soit p un entier premier.
- Si a est un entier non divisible par p , alors : $a^{p-1} \equiv 1\ [p]$.
- Pour tout entier a, on a : $a^p \equiv a\ [p]$.

On ne revient pas ici sur les démonstrations de ce Théorème.

Fonction Indicatrice d'Euler :
On appelle Fonction Indicatrice d'Euler la fonction notée φ définie par :

$$\forall n \in \mathbb{N}^*, \quad \varphi(n) = \text{nombre d'entiers } a \text{ compris entre 1 et } n \text{ tels que } a \text{ et } n \text{ soient premiers entre eux.}$$

Autrement dit, $\varphi(n) = \text{Card}\{a \in \mathbb{N} \setminus\ 1 \leq a \leq n \ \text{ et } \text{Pgcd}(a, n) = 1\}$.

Généralisation du Petit Théorème de Fermat :
Soit $n \in \mathbb{N}^*$ et soit a un entier premier avec n. Alors $a^{\varphi(n)} \equiv 1\ [n]$.

Inverse modulo n
Soit a un entier. On dit que a est inversible modulo n si et seulement si il existe $b \in \mathbb{Z}$ tel que $a \times b \equiv 1\ [n]$.
b est appelé "un inverse de a modulo n".
Ceci ne signifiant rien d'autre qu'il existe un entier u tel que $ab + nu = 1$, on peut dire que a est inversible modulo n si et seulement si a et n sont premiers entre eux. a et n étant premiers entre eux, la recherche d'un inverse de a modulo n revient donc déterminer une solution $(b\ ,\ u)$ à l'équation $ab + nu = 1$, solution que l'on peut obtenir par l'algorithme d'Euclide.

Algorithme Cryptage RSA : Principe

Considérons un entier n produit de deux entiers premiers p et q donc, tel que $n = p \times q$ avec p et q premiers.
Posons $\omega = (p-1)(q-1)$ et soit un entier d premier avec ω et soit e un entier e tel que $de \equiv 1\ [\omega]$.
Alors : Pour tout entier M et tout entier N : $M^d \equiv N\ [n] \iff M \equiv N^e\ [n]$
Expliquons comment fonctionne ce cryptage.

- On détermine deux nombres premiers *assez grands* p et q et on pose $\omega = (p-1)(q-1)$.
- On détermine un nombre d premier avec ω *assez grand*.
- On calcule alors $n = p \times q$ puis on détermine un inverse e modulo ω de d.
- On fournit à une personne A l'entier n et l'entier e.
  La connaissance de n et e lui permet de calculer pour tout entier M , $M^e\ [n]$.
- On fournit à une personne B l'entier n et l'entier d.
  La connaissance de n et e lui permet de calculer pour tout entier M , $M^d\ [n]$.
- A ce stade, si A veut transmettre à B un entier M, qui a valeur de message "secret", il calcule $N = M^d$ modulo n et transmet ce résultat C à B.
- B calcule alors N^e modulo n et obtient la valeur de M

La démonstration de la validité de cet algorithme est donné en TS-Spé-Maths mais sa mise en place technique est rarement effectué à ce niveau. Pourtant, celle-ci fait appel à une suite de fonctions simples et le lancement de RSA à la fin, que ce soit pour le codage ou le décodage est assez aisé. On peut alors demander de contruire une suite de fonctions sous Python qui conduisent à la mise en place finale de RSA.

Voici à titre d'exemple, une liste de fonctions arithmétiques diverses écrites sous Python.

1. **Déterminer si p est premier ou pas**

```
def premier(n) :
    x=2
    y=1
    while n%x !=0 and x**2<n :
      x=x+1
    if n%x==0 and n>2 :
      y=0
    return not(n%x==0 and n>2)  Retourne False si n n'est pas premier et True sinon
```

2. **Liste des nombres premiers $\leq n$ en utilisant le crible d'Erastothène**

```
def liste(n) :
    tableau = [i for i in range(0, n+1)]
    for i in range(0, int(floor(sqrt(n+1))+1)) :
    if tableau[i] >1 :
      for j in range(i*2, n+1, i) :
        tableau[j] = 0
    return [p for p in tableau if p>1]
```

3. **Chercher le plus petit nombre premier supérieur à n**

```
def premiersup(n) :
    x= int(abs(n-1/2)+3/2)
    while premier(x)==0 :
      x=x+2
    return x
```

4. **Calcul du pgcd de a et b par l'algorithme d'Euclide**

```
def pgcd(a,b) :
    x,y=a,b
    r=x%y
    while r>0 :
      x,y=y,r
      r=x%y
    return y
```

5. **Fonction Indicatrice d'Euler**

```
def Euler(n) :
    y=0
    for k in range(1,n) :
      if pgcd(n,k)==1 :
        y=y+1
    return y Calcule tout simplement le nombre d'entiers < n et premiers avec n
```

6. **Détermination d'une solution (u , v) de l'équation $au + bv =$ pgcd(a, b), reposant sur l'algorithme d'Euclide**

```
def bezout(a,b) :

    r0,r1=b,a
    u0,u1=0,1
    v0,v1=1,0
    while r1 !=0 :
      q= r0//r1
      r0,r1=r1,r0-q*r1
      u0,u1=u1,u0-q*u1
      v0,v1=v1,v0-q*v1
    R=r0
    U=u0
    V=v0
    return [U,V]
```

Cette fonction repose que le principe suivant :
(u_0, v_0) étant une solution de $ax+by = c_0$ et (u_1, v_1) une solution de $ax + by = c_1$ avec $c_1 \neq 0$, alors $(u_0 - qu_1, v_0 - qv_1)$ est une solution de $ax + by = r$ où r et q sont le reste et le quotient dans la division euclidienne de c_0 par c_1.
On considère alors la suite des couples (q_k, r_k) des (quotient, reste) successifs dans l'algorithme d'Euclide avec $a = q_1 b + r_1$, r_n étant Pgcd(a,b).
On pose alors $(u_0, v_0) = (1, 0)$ et $(u_1 v_1) = (0, 1)$ et la relation de récurrence $(u_k, v_k) = (u_{k-2} - q_k u_{k-1}, v_{k-2} - q_k v_{k-1})$.
(u_0, v_0) et (u_1, v_1) sont solutions respectivement de $ax + by = a$ et $ax + by = b$.
D'où, par récurrence, (u_n, v_n) est solution de $ax+by =$ pgcd(a, b).

7. **Fonction qui génére des entiers p, q, n, ω, d et e utilisés dans RSA**

```
def genere(N) :
    p=premiersup(N)
    q=premiersup(p+N)
    n=p*q
    w=(p-1)*(q-1)
    d=N
    while pgcd(w,d) !=1 :
      d=d+1
      e=bezout(d,w)[0]%w
    return [p,q,n,w,d,e]
```

8. **Fonction qui code un message M, en connaissant n et d**

```
def Code(M,n,d) :
    y=1
    for k in range(d) :
      y=(y*M)%n
    return y
```

9. **Fonction qui décode un message C, en connaissant n et e**

```
def Decode(C,n,e) :
    y=1
    for k in range(e) :
      y=(y*C)%n
    return y
```

Bien sûr, les deux dernières fonctions sont identiques en terme de calcul !

Exemple d'utilisation
Le lancement de **genere(10000)** retourne : [**10007, 20011, 200250077, 200220060, 10001, 5005001**].
On peut donc choisir $n = 200250077$, $d = 1001$ et $e = 5005001$

On donne les valeurs $n = 200250077$ et $d = 1001$ à une personne A.
Elle code un nombre M en demandant : **Code(M,**n**,**d**)**.

Elle envoie alors le résultat C à une personne B qui a connaissance de $n = 200250077$ et de $e = 5005001$.
B décode le message en demandant **Decode(C,**n**,**e**)**.

Par exemple :
Si A veut "coder" **M = 123456**, il obtient **Code(M,**n**,**d**) = 85564036**.
Une fois ce résultat transmis à B, ce dernier demande : **Decode(85564036 ,** n**,** e**)** et obtient bien 123456.

Seule la personne ayant généré la suite des entiers [**10007, 20011, 200250077, 200220060, 10001, 5005001**] peut à la fois, en principe !, Coder et Décoder car la connaissance de d et e simultanément suppose la connaissance de $\omega = (p-1)(q-1)$ qui suppose la connaissance de la décomposition en facteurs premiers de n.

"Casser" RSA demande donc d'être capable de trouver la décomposition en facteurs premiers d'un entier très grand.

Probabilité, Simulation

Stage Laboratoire Mathématiques, Mai-Juin 2019

Alain SEBAOUN , Lycée Jeanne d'Albret

Principe de base

Le module **random** permet d'utiliser les trois fonctions suivantes :
- randint(a,b) qui revoie un entier choisi de façon aléatoire entre a et b.
- random() qui renvoie un "réel choisi de façon aléatoire dans l'intervalle $]0\,;\,1[$.
- choice(L) qui renvoie un élément de la liste L choisi de façon aléatoire.

On peut considèrer que random() simule une variable aléatoire suivant la loi uniforme sur $]0\,;\,1[$. La simulation des lois usuelles peuvent alors être faite en partant de celle de la loi uniforme sur $]0\,;\,1[$.

I : Simulation des variables usuelles

1. **Loi de Bernoulli de paramètre $p \in \,]0\,;\,1[$**
   Il s'agit de retourner la valeur 0 ou 1, la probabilité de retourner 1 devant être p.
   Si U suit la loi uniforme sur $]0\,;\,1[$, on sait que, pour $p \in \,]0\,;\,1[$, on a $\mathbb{P}(U < p) = p$. On peut alors simuler la loi de Bernoulli de la façon suivante :

```
def SimuleBern(p) :
    return random()<p
```

2. **Loi Binomiale de paramètres (n , p)**
   On se rappelle que X suit la loi binomiale de paramètres $(n,\, p)$ signifie que X est la somme de n variables de Bernoulli mutuellement indépendantes et de même paramètre p.
   Pour simuler une variable aléatoire suivant la loi binomiale de paramètre $(n\ ,\ p)$, il suffit alors de sommer n simulations de la loi de Bernoulli. On peut alors écrire ceci :

```
def SimuleBino(n,p) :
    X=0
    for k in range(n) :
      X=X+SimuleBern(p)
    return X
```

3. **Loi Géométrique de paramètre p**
   Il suffit de répéter la loi de Bernoulli jusqu'à l'obtention d'un succés ! D'où ..

```
def SimuleGeo(p) :
    X=1
    alea=random()
    while alea>p :
      alea=random()
      X=X+1
    return X
```

4. **Loi Uniforme sur** $\{1$ **,** 2 **,** 3 **,** $\cdots$ **,** $n\}$
On utilise randint(a,b), tout simplement !

```
def SimuleUnif(n) :
    return randint(1,n)
```

Mais on peut aussi passer par random(), en utilisant les fonction floor() ou ceil(), mais ces fonctions demandent l'importation du module math.

```
def SimuleUnif(n) :
    return ceil(n*random())
```

5. **Loi Uniforme sur l'intervalle** $[a$ **,** $b]$.
Il suffit de faire un changement affine en partant de la loi uniforme sur [0 , 1].

```
def SimuleUnif(a,b) :
    return (b-a)*random()+a
```

6. **Exemple d'une variable suivant une loi discrète**

X est la variable aléatoire telle que $\mathbb{P}(X = x_1) = p_1$, $\mathbb{P}(X = x_2) = p_2$, $\mathbb{P}(X = x_3) = p_3$ ... $\mathbb{P}(X = x_n) = p_n$.
On remarque que l'on peut partitionner l'intervalle]0 , 1] en n intervalles de longueur respective p_1 , p_2 , p_3 ... p_n de la façon suivante :

$$I_1 =]0; p_1], \quad I_2 =]p_1; p_1 + p_2], \quad I_3 =]p_1 + p_2, p_1 + p_2 + p_3], \quad \text{et } I_n =]p_1 + p_2 + p_3 + \cdots p_{n-1}; 1]$$

La longueur de l'intervalle I_k est p_k. On a donc, si note U la variable suivant la loi uniforme sur]0 ; 1] :

$$\mathbb{P}(X = x_k) = \mathbb{P}(U \in I_k)$$

La simulation simple de X se fait alors de la façon suivante :
— On fait une simulation de U avec random().
— On teste l'appartenance de ce résulat à chacun des intervalles I_k.
— Dès que l'on constate que le résultat est dans un intervalle I_k, on retourne la valeur x_k.
X étant la liste des valeurs que prend la variable aléatoire X et P la liste des probabilités correspondantes :

$$\text{X} = [x_1, x_2, x_3, \cdots, x_n] \quad \text{et P} = [p_1, p_2, p_3, \cdots, p_n]$$

on a alors une simulation de X par la fonction suivante :

```
def SimuleDiscrete(X,P) :
    k , a , b =0 , 0 , P[0]
    alea=random()
    while not(a<alea and alea<=b) :
      a,b,k=a+P[k],b+P[k+1],k+1
    return X[k]
```

On peut remarquer que cette méthode évite l'usage des intructions conditionnelles **if .. elif .. else ...** qui peuvent devenir lourdes à gérer quand le nombre de valeurs que peut prendre X est trop important.
Par exemple, considèrons le lancer d'un dé dont les faces sont numérotées de 1 à 6 et tel que la probabilité d'obtenir une face est proportionnelle au numéro de la face. La probabilité d'obtenir la face k est alors $p_k = \dfrac{k}{21}$.
Pour simuler ce dé, il suffit de saisir les deux listes suivantes :

$$\text{X=[1,2,3,4,5,6]} \quad \text{et} \quad \text{P=[1/21,2/21,3/21,4/21,5/21,6/21]}$$

puis d'appeller la fonction SimuleDiscrete en tapant : `SimuleDiscrete(X,P)`

7. **Loi Exponentielle de paramètre $\lambda > 0$**

   En partant d'une variable U suivant la loi uniforme sur $]0\ ,\ 1[$, on peut définir la variable $E = -\dfrac{\ln(U)}{\lambda}$. On vérifie alors que E suit la loi exponentielle de paramètre λ. Une simulation de $E \hookrightarrow \mathcal{E}(\lambda)$ peut alors être faite :

```
def SimuleExpo(lambda)
    return -log(random())/lambda
```

II : Méthode Monte Carlo

On appelle **méthode de Monte-Carlo** toute méthode visant à calculer une valeur numérique en utilisant des procédés aléatoires. Dans le cas du calcul d'une intégrale, cette méthode permet d'en déterminer une valeur approchée par la simulation d'un variable aléatoire et le calcul de l'espérance de cette variable, ou en tout cas, une estimation de son espérance en prenant la moyenne observée. Voici quelques exemples :

1. **Utilisation de la Valeur Moyenne de f sur $[a\ ,\ b]$.**
   Rappelons que la valeur moyenne d'une fonction f continue sur $[a\ ,\ b]$ avec $a < b$ est donnée par :

$$\mu = \frac{1}{b-a}\int_a^b f(x)dx$$

   On peut alors estimer $\displaystyle\int_a^b f(x)dx$ en posant $X = f((b-a)U + a)$ où U suit la loi uniforme sur $[0\ ,\ 1]$. On sait, en utilisant le Théorème de transfert, que l'espérance de X est μ.
   On a alors une estimation de $\displaystyle\int_a^b f(x)dx$ en simulant X un grand nombre de fois et en effectuant la moyenne des résultats observés, multipliée par $(b-a)$.
   On ne revient pas ici sur les arguments théoriques à mettre en place pour une utilisation "totalement" rigoureuse, mais voilà une fonction Python dont les variables sont a, b, f et n où a et b sont les bornes de l'intégrale à déterminer, f la fonction à intégrer et n le nombre de fois où on simule X.

```
def MonteCarloMoyenne(a,b,f,n) :
    S=0
    for k in range(n) :
      S=S+f((b-a)*random()+a)
    return (b-a)*S/n
```

   A titre d'exemple, pour avoir une petite idée de la valeur de ln(10), on part de $\ln(10) = \displaystyle\int_1^{10} \frac{\mathrm{d}x}{x}$. On définit alors la fonction Python suivante :

```
def f(x) :
    return 1/x
```

   Puis on appelle la fonction MonteCarloMoyenne. **MonteCarloMoyenne(1,10,f,10000)** si on veut le calcul sur la base de 10000 simulations. On pourra observer que la valeur obtenue n'est pas trop éloignée de la valeur ln(10).

2. **Utilisation de l'aire sous la courbe !**

On se place ici dans le cas où la fonction f est positive. On sait que pour $a < b$, $\int_a^b f(x)\mathrm{d}x$ est l'aire de la partie du plan située entre la courbe de f et l'axe des abscisses sur l'intervalle [a,b].

On sait que cette aire est donnée par : $Aire = \int_a^b f(x)\mathrm{d}x$, en unité d'aire !

Considérons alors un rectangle $ABCD$ contenant cette partie du plan avec A(a,0) et B(b,0), C(b, h) et D(a,h), h étant un majorant de f sur $[a, b]$. L'aire de $ABCD$ est alors $h(b-a)$.

On choisit alors au hasard un point à l'intérieur du rectangle $ABCD$. Ce point est, soit sous la courbe de f, soit au-dessus.

La probabilité que ce point soit sous la courbe de f est alors $p = \dfrac{Aire}{h(b-a)}$.

Si on choisit n points mutuellement indépendants dans le rectangle $ABCD$, on peut alors dire que le nombre X_n de points situés sous la courbe de f suit une loi binomiale de paramètres (n,p).

Donc, la variable $F_n = \dfrac{X_n}{n}$ a pour espérance p. Une simulation de X_n et le calcul de F_n donne alors une estimation de p, d'où une valeur approchée de l'aire cherchée et donc de l'intégrale !

On peut résumer en disant que la méthode de Monte-Carlo ci-dessus n'est rien d'autre que la simulation d'une loi binomiale dont on ne connait pas p !

Voici alors un exemple de fonction Python effectuant cette simulation.

Les variables a,b,h,f et n sont respectivement :
- pour a et b, les bornes de l'intervalle d'intégration,
- pour h, la "hauteur" du rectangle ABCD de base AB. h n'est rien d'autre qu'un majorant de f sur [a , b].
- f la fonction que l'on intégre,
- n le nombre de simulations effectuées, c'est à dire, le nombre de points tirés au hasard dans le rectangle $ABCD$.

```
def MonteCarloInt(a,b,h,f,n) :
    S=0
    for k in range(n) :
      x=(b-a)*random()+a
      y=h*random()
      if y<f(x) :
        S=S+1
    return h*(b-a)*S/n
```

Mais cette méthode présente un petit inconvénient par rapport à la méthode précédente ! Il faut connaître un majorant de f sur [a, b] pour la lancer.

Des exemples classiques !

(a) Utilisation de $f(x) = \dfrac{1}{1+x^2}$ sur [a , b] = [0 , 1] pour obtenir une valeur approchée de $\dfrac{\pi}{4}$.

(b) Utilisation de $f(x) = \sqrt{1-x^2}$ sur [a ,b] = [-1 , 1] pour obtenir une valeur approchée de $\dfrac{\pi}{2}$.

3. **Une généralisation de la méthode de Monte Carlo pour f quelconque.**

On se place ici dans le cas où f est une fonction continue sur l'intervalle $I = [a\ ,\ b]$. Une rapide étude de f a montré que pour tout $x \in I$, $m \leq f(x) \leq M$ où m et M sont deux constantes réelles.

— Comment alors utiliser une méhode style "Monte-Carlo" pour obtenir une estimation de $\displaystyle\int_a^b f(x)\mathrm{d}x$?

A priori,le signe de f n'est pas connu !

Le principe est de choisir de façon aléatoire un point M de coordonnées $(x\ ,\ y)$ avec

$$a \leq x \leq b \text{ et } m \leq y \leq M$$

puis de savoir si ce point est situé entre la courbe de f et l'axe des abscisses.
On définit alors une variable aléatoire X de la façon suivante : le point M étant choisi,

$$X = \begin{cases} 1 & \text{si } M \text{ est compris entre la courbe de } f \text{ et l'axe des abscisses et au-dessus de cet axe} \\ -1 & \text{si } M \text{ est compris entre la courbe de } f\text{et l'axe des abscisses et en dessous de cet axe} \\ 0 & \text{dans les autres cas} \end{cases}$$

Réponse Possible :

```
def MonteCarlo(a,b,m,M,f,n) :
    S=0
    for k in range(n) :
      x=(b-a)*random()+a
      y=(M-m)*random()+m
      S=S+(0<y and y <f(x)) - (y<0 and f(x)<y)
    return (M-m)*(b-a)*S/n
```

4. **Un exemple simple de simulation, Choix d'un point dans un disque**

La méthode précédente pour estimer une valeur de $\int_a^b f(x)\mathrm{d}x$ peut s'étendre à la simulation d'une variable aléatoire dont on ne connait pas la loi de probabilité à priori.

Prenons, par exemple, le choix aléatoire d'un point M situé dans un disque $\mathcal{D}$ de centre O et de rayon $r = 1$ et notons X la variable aléatoire égale à distance OM. Si on formule l'hypothèse que la probabilité de choisir le point M dans une partie $\mathcal{A}$ fixée de $\mathcal{D}$ est proprotionnelle à l'aire de $\mathcal{A}$, on peut peut considèrer que le choix de M est "uniforme" dans $\mathcal{D}$.
Le choix de M, ou sa simulation, peut alors être fait de la façon suivante :
— On choisit un point suivant la loi uniforme dans un carré de côté 2 contenant $\mathcal{D}$ en utilisant deux lois uniformes sur [-1 , 1] X_1 et Y_1.
— Si $X_1^2 + Y_1^2 \leq 1$, on a un point M de coordonnées (X_1, Y_1) du disque.
— Sinon, on rejette le point obtenu.
Cette méthode s'apparente à la Méthode du Rejet en version "simplifiée". Elle permet de choisir de façon aléatoire un point du disque $\mathcal{D}$ en respectant le choix uniforme. La fonction suivante donne alors une simulation du choix de M dans $\mathcal{D}$.

```
def SimuleDisque() :
    X1=2*random()-1
    Y1=2*random()-1
    while X1**2+Y**2¿1
      X1=2*random()-1
      Y1=2*random()-1
    return [X1,Y1]
```

Si on cherche alors à estimer l'espérance de $X = OM$, il suffit de faire une boucle en utilisant la fonction précédente :

```
def MoyenneX(n) :
    m=0
    for k in range(n) :
      M=SimuleDisque()
      m=m+sqrt(M[0]**2+M[1]**2)
    return m/n
```

Il apparait, en prenant des valeurs assez grandes pour n, qu'une estimation de l'espérance de X est de l'ordre de 0.666.., la valeur exacte étant $\frac{2}{3}$.
Cette valeur exacte est un résultat accessible en Terminale simplement par détermination de la fonction de répartition F de X, puis de sa fonction de densité f.

$$\forall x \in [0,1];\ F(x) = x^2 \quad \text{et} \quad f(x) = 2x$$

On peut remarquer que simuler le choix d'un point dans le disque $\mathcal{D}$ n'est pas toujours faire un choix suivant une loi uniforme dans ce disque et que la méthode mise en place peut conduire à des simulations semblant être celle d'une loi uniforme, mais qu'une étude un peu poussée montre inexacte. En voici deux exemples.

Exemple I :

Si on se place dans le plan muni d'un repère orhonormé direct $(O\,;\,\vec{i},\vec{j})$ du plan, un point M du disque de centre O et de rayon 1 est déterminé par la distance OM et son angle polaire, autrement dit, en passant par ses coordonnées polaires (r,θ) avec $r \in [0;1]$ et $\theta \in [0;2\pi[$.

Choisir M dans ce disque revient à choisir r dans $[0\,;\,1]$ et θ dans $[0\,;\,2\pi[$ en suivant des lois uniformes.

Mais le point M serait-il lui aussi choisi de façon "uniforme" dans le disque par cette méthode ?

Si on fait un tel choix de M, la variable $X = OM$ n'est rien d'autre que r. Dans ce cas, X suit la loi uniforme sur $[0\,;\,1]$ et n'a pas pour fonction de densité $x \to 2x$.

On voit donc que ce choix de M n'est pas uniforme, malgré l'intervention de deux lois uniformes auxiliaires.

Exemple II :

On se place toujours dans un repère orthonormé $(O\,;\,\vec{i},\vec{j})$ et on considère toujours le disque de centre O et de rayon 1.

L'équation cartésienne de ce disque est : $\mathcal{D}:\ x^2+y^2 \leq 1$.

Donc, $M(x,y)$ appartient à $\mathcal{D}$ si et seulement si $\begin{cases} -1 \leq x \leq 1 \\ \text{et} \\ -\sqrt{1-x^2} \leq y \leq \sqrt{1-x^2} \end{cases}$

Choisir un point $M \in \mathcal{D}$ revient donc à choisir :

— X dans l'intervalle [-1 ; 1] ,

— puis Y dans l'intervalle $[-\sqrt{1-X^2}\ ,\ \sqrt{1-X^2}]$.

On considère alors que le couple $(X\ ,\ Y)$ représente les coordonnées de M.

Voici alors une fonction Python simulant la choix de M par cette méthode :

```
def SimuleM3() :
    X=2*random()-1
    Y=2*sqrt(1-X**2)*random() - sqrt(1-X**2)
    return [X,Y]
```

Tout semble bien être posé ! Il ne reste qu'à voir ce qu'il se passe si on veut estimer la moyenne des distances OM, c'est-à-dire, en posant $D = \sqrt{X^2+Y^2}$, l'espérance de D.

On écrit pour cela la fonction suivante :

```
def MoyenneD(n) :
    m=0
  for k in range(n)
      M=Simule3()
      m = m +sqrt(M[0]**2 + M[1]**2)
  return m/n
```

En lançant, par exemple **MoyenneD(100000)**, on arrive sur une valeur de l'ordre de 0.708 ... valeur assez éloignée du $\dfrac{2}{3}$ attendu.

En fait, la variable $D = \sqrt{X^2+Y^2}$ suit une loi dont la fonction de densité est trop complexe pour être montrée à des élèves de Terminale mais gérable par des élèves de MP.

Elle fait intervenir des fonctions elliptiques et on montre que la fonction de répartition de D est définie par :

$$\forall x \in [0\ ;\ 1], \quad F(x) = \int_0^x \sqrt{\frac{x^2-t^2}{1-t^2}}\,\mathrm{d}t$$

On peut voir qu'il ne suffit pas de dire "on choisit un point au hasard", encore faut-il bien préciser quelle est la méthode de choix mise en place.