Contenus

Articles

Références

Licence des articles

Mathématiques avec Python et Ruby

Introduction

Les deux langages de programmation Python et Ruby ont en commun :

1. d'être libres (en particulier on peut aisément consulter leur code source, écrit dans le langage lui-même) ;
2. d'être des langages objets (et des objets mathématiques, il y en a) ;
3. d'être munis de consoles légères et interactives (IDLE pour *Python*, irb (*interactive Ruby*) pour *Ruby*)

Il est donc intéressant d'explorer ces langages pour résoudre des problèmes de nature mathématique. Dans ce livre, nous énumèrerons ce que ces langages apportent à l'enseignement des mathématiques et à celui de l'algorithmique. En particulier, nous étudierons comment certaines structures mathématiques sont gérées par ces deux langages.

Important : Certaines fonctionnalités de *Python 3.2* seront utilisées ici (par exemple, le fait que la division par défaut est la division exacte et pas la division euclidienne, la présence de *print*, le fait que l'objet *fraction* est fourni avec *Python 3.2*...).

Deux moyens ont été utilisés pour mettre au point les scripts *Python* ci-dessous :

1. la console *IDLE* qui est interactive ;
2. l'écriture d'un fichier *test.py* puis l'écriture dans une console système de *python test.py* ou *python3.2 test.py* selon le cas.

Pour *Ruby*, c'est la version 1.9.1 qui sera utilisée. Là encore, deux moyens ont été utilisés :

1. l'interpréteur *irb* (*Interactive Ruby*) qui est écrit en *Ruby* (son code source est donc consultable) et qui est interactif ;
2. l'écriture d'un script dans un fichier *test.rb* puis l'exécution dans la console système, de *ruby test.rb*.
3. Freeride est un éditeur léger qui fonctionne bien avec Ruby. FreeRide peut exécuter le programme Ruby dans la fenêtre active sans avoir à quitter l'éditeur, en cliquant simplement sur *Exécuter*. On peut préférer NetBeans pour son jeu de fonctionnalités plus étendues et l'intégration avec Java, mais NetBeans requiert plus de ressources système. Les deux programmes sont disponibles pour Windows, Mac OS X, et Linux, et les deux peuvent gérer

des projets Ruby, qui peuvent inclure plusieurs fichiers texte (connexe) contenant des programmes Ruby.

4. Quoi qu'il en soit, pour ce livre, Geany a été utilisé, principalement parce qu'il gère à la fois Python et Ruby (parmi beaucoup d'autres), et est assez petit.
5. bien que cette fonctionnalité n'ait pas été utilisée ici, *Ruby* possède un interpréteur en ligne, qui permet donc de faire du *Ruby* sans installation (certes, il y a un équivalent pour *Python*, c'est SAGE). L'interpréteur en ligne de *Ruby* est disponible en suivant ce lien : [1].

Sommaire

Python

Ruby

- Nombres
  - Nombres entiers
  - Fractions
  - Réels
  - Complexes
  - Quaternions
- Probabilités
  - Évènements
  - Nombres pseudo-aléatoires
- Analyse
  - Suites
  - Fonctions
  - Analyse numérique
- Géométrie
  - Points
  - Vecteurs
  - Droites
- Résolution de problèmes
  - Systèmes
  - Triplets pythagoriciens
  - Systèmes congruentiels
  - Problème de Freudenthal
  - Théorie de Joukovski

Pour aller plus loin

- Pour faire des mathématiques sous Python, on consultera avec bonheur et intérêt les recettes de Tyrtamos [2].
- Pour approfondir votre découverte de ces langages, Programmation Ruby, Exemples de code ruby, Programmation Python, Utilisons Python pour enseigner les algorithmes ou encore Apprendre à programmer avec Python

Références

[1] http://tryruby.org/
[2] http://python.jpvweb.com/mesrecettespython/doku.php?

Nombres entiers en Python

Les nombres entiers ne sont pas les seuls nombres, comme on le verra dans les chapitres suivants. Alors comment fait *Python* pour savoir qu'un nombre est entier? Comme le langage est faiblement typé, il doit le deviner. Le critère est simple: Pour qu'un nombre soit entier, il ne doit pas avoir de virgule (représentée dans *Python* par un point décimal).

Les nombres entiers dans Python

Ainsi, si on entre

```
a=3
print(type(a))
b=3.14
print(type(b))
c=int(b)
print(c)
```

on constate que *Python* sait que *a* est entier, que *b* ne l'est pas, et que *c* peut être entier bien qu'obtenu à partir de *b* (qui est réel).

Certains calculs devant donner un résultat entier ne le font pas toujours en *Python*. Par exemple, alors que $\sqrt{100} \in \mathbb{N}$, *Python* considère ce nombre comme un réel (non entier)!

```
from math import *
a=sqrt(100)
print(a.is_integer())
```

Opérations

Addition, soustraction et multiplication

Les trois premières opérations se notent avec les symboles +, - et * comme dans la plupart des langages de programmation. La somme, la différence et le produit de deux entiers (ou plus) sont des entiers (relatifs):

```
a=5
b=-8
print(a+b)
print(a-b)
print(a*b)
```

La tortue

On peut représenter l'addition par des mouvements successifs de la tortue. Pour cela il faut bien entendu importer le module *turtle*. Après ça pour additionner 55 et 34, on peut faire avancer la tortue successivement de 55 pixels puis 34 pixels, et regarder où elle est:

```
from turtle import *
forward(55)
forward(34)
print(position())
```

Pour soustraire le deuxième nombre au lieu de l'additionner, il suffit de faire reculer la tortue au lieu de la faire avancer; et ça marche même avec des nombres négatifs:

```
from turtle import *
forward(55)
backward(-34)
print(position())
```

Divisions

Il y a deux sortes de divisions d'entiers en *Python*: Le quotient euclidien, qui est un entier, et le quotient exact, qui est une fraction (pour *Python*, un réel):

Quotients

Dans les anciennes versions de *Python*, le script suivant

```
a=3
b=2
print(a/b)
```

affichait 1 au lieu de 1.5, parce que pour *Python*, comme *a* et *b* sont entiers, leur quotient était logiquement euclidien. Ceci a changé, maintenant le script ci-dessus produit bien 1.5; mais du coup, si par hasard on voulait quand même calculer une division euclidienne avec *Python*? Et bien dans ce cas il faudrait dédoubler le *slash* qui code la division:

```
a=3
b=2
print(a//b)
```

Ce peut être utile parce que même en divisant 4 par 2, le résultat est un réel non entier, alors que la division euclidienne produit bien le nombre entier 2.

Reste euclidien

Le reste de la division euclidienne de 13 par 8 est 5. Pour le calculer, on utilise l'opérateur infixé %:

```
a=13
b=8
print(a%b)
```

Cette opération permet de travailler sur les congruences. Remarque: Si b=0, on a le même message d'erreur que lorsqu'on divise par 0.

Divisibilité

Deux entiers quelconques on un pgcd. En *Python*, on l'obtient avec *gcd*. Mais cette fonction ne se trouve pas par défaut dans *Python*; elle se trouve dans le fichier *fractions.py*, dont on peut importer la totalité des objets par *from fractions import **:

```
a=13572468
b=12345678
print(gcd(a,b))
```

Comme *Python* est libre, on peut consulter le source de *fractions.py*, où on trouve ceci:

```
def gcd(a, b):
    while b:
        a, b = b, a%b
    return a
```

On reconnaît (d'autant plus aisément que le langage est concis) l'algorithme d'Euclide.

Puissances

Beaucoup de langages de programmation utilisent le chapeau pour représenter les puissances. Pas *Python* pour qui le chapeau est déjà pris par une autre opération (le *ou exclusif* bit à bit). Alors c'est l'astérisque de la multiplication qui est utilisé pour les puissances, mais en le dédoublant:

```
a=4
b=2
print(a**b)
print(b**a)
```

Remarque: Si l'exposant est négatif ou non entier, le résultat est un réel. En particulier, les deux opérations ci-dessous ont le même effet:

```
print(100**0.5)



from math import *
print(sqrt(100))
```

Priorités opératoires

En *Python* comme en algèbre, on effectue dans l'ordre

1. Les parenthèses
2. Les fonctions (comme l'élévation à une puissance)
3. Les multiplications et divisions
4. Les additions et soustractions.

Ainsi

```
print(2+3*5)
```

affiche 17 et non 25: Les opérations ne sont pas effectuées de gauche à droite, mais en suivant les priorités opératoires.

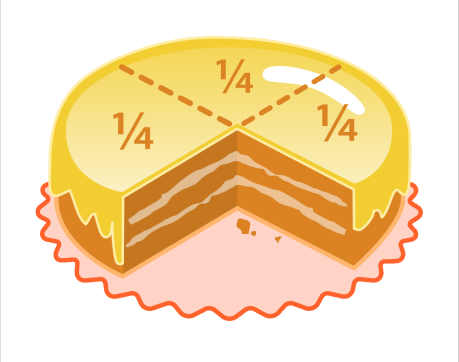
Itérateurs

Un itérateur de *Python* est une liste d'entiers. Par exemple, la liste 10, 13, 16, 19, 22, 25 (allant de 3 en 3, de 10 jusqu'à 25) s'appelle *range(10,26,3)*. On peut abréger les itérateurs en omettant le troisième argument (par défaut 1) voire le premier (par défaut 0). Ainsi *range(5)* contient les entiers 0, 1, 2, 3 et 4 (0 compris, 5 non compris, ce sont donc bien les 5 premiers entiers naturels).

Lorsqu'on lance un dé, le résultat obtenu peut être 1, 2, 3, 4, 5 ou 6. Il peut donc être décrit par *range(1,7)*. Pour vérifier que l'évènement "le dé tombe sur 3" est possible alors que l'évènement "le dé tombe sur 7" est impossible, on peut faire

```
dice=range(1,7)
print(3 in dice)
print(7 in dice)
```

Fractions en Python



L'écriture de nombres non entiers sous forme de fractions a de loin précédé celle des nombres décimaux, puisque les égyptiens et les babyloniens les utilisaient déjà. Chaque fois que le dénominateur n'est pas une puissance de 10, on continue encore de nos jours à utiliser des écritures fractionnaires plus ou moins cachées, comme dans les exemples suivants:

1. Lorsqu'on dit qu'un homme mesure 5 pieds 7 pouces, ça signifie que sa taille, en pieds, est $5+\frac{7}{12}=\frac{67}{12}$(il y a douze pouces dans un pied);
2. Lorsqu'on dit qu'il est 8 heures 13, c'est que depuis minuit, il est passé exactement $8+\frac{13}{60}=\frac{493}{60}$heures (soit 493 minutes).
3. Lorsque Roméo se plaint d'avoir attendu Juliette pendant plus de trois quarts-d'heure, il exprime la durée de son attente insoutenable sous la forme d'une fraction...
4. Les probabilités se donnent aussi souvent sous forme de fractions (le plus souvent des fractions égyptiennes). Comme dans "j'ai une chance sur 10 millions de recevoir une météorite sur la tête" ou "Casaque Jaune est donné favori à 5 contre 1".
5. Idem parfois pour les statistiques: "5 Français sur 7 estiment qu'il y a trop de sondages sur les sondages"...

L'égalité 0,2+0,5=0,7 peut s'écrire $\frac{1}{5}+\frac{1}{2}=\frac{7}{10}$mais l'égalité $\frac{1}{2}+\frac{1}{3}=\frac{5}{6}$ne peut pas s'écrire sous forme décimale exacte parce que le résultat n'est pas décimal. *Python* affiche 1/2+1/3=0.8333333333333333 et malgré l'abondance de chiffres, cette égalité n'est pas exacte.
Pour faire des calculs exacts avec des fractions, *Python* a un module *fractions.py*. Pour transformer *Python* en un langage de programmation spécialisé dans les fractions, il suffit de précéder les scripts de ce chapitre de la ligne

```
from fractions import *
```

qui importe le module *fractions* en entier (pourquoi faire dans le détail?). Une alternative est de chercher le fichier *__init__.py*, initialement vide, et d'y mettre la ligne précédente.

Obtention d'une fraction

Pour entrer la fraction $\frac{n}{d}$dans *Python*, on entre *Fraction(n,d)* non sans avoir importé le module *fractions*:

```
from fractions import *
a=Fraction(24,10)
print(a)
```

On constate que la fraction $\frac{24}{10}$a été automatiquement simplifiée par *Python* au moment de son instanciation.
Si on entre 0 comme dénominateur, la fraction ne se crée pas et on a un message d'erreur: Comme une fraction est un quotient, on ne peut pas diviser par 0.

Une fois qu'une fraction est calculée, on peut obtenir son numérateur et son dénominateur par

```
from fractions import *
a=Fraction(24,10)
print(a.numerator)
print(a.denominator)
```

Bien entendu, le numérateur de $\frac{24}{10}$n'est pas 24...

Pour obtenir la valeur de la fraction (le quotient de son numérateur par son dénominateur), on peut lui additionner 0.0: La somme d'une fraction et d'un réel, même nul, est un réel.

```
from fractions import *
a=Fraction(24,10)
print(a+0.0)
```

Réciproquement, on peut convertir un nombre réel en fraction, mais le résultat est parfois surprenant, si on prend un nombre dont le développement décimal s'arrête alors que le développement binaire ne le fait pas:

```
from fractions import *
a=Fraction.from_float(1.2)
print(a)
```

On s'attendait à $\frac{6}{5}$, on a l'approximation un peu surprenante $\frac{5404319552844595}{4503599627370496}$...

Opérations

Les opérations sur les fractions se notent comme celles sur les autres nombres, mais en général le résultat est une fraction.

Opérations unaires

Opposé

L'opposée d'une fraction s'obtient en la faisant précéder du signe -. Par exemple, la fraction suivante est positive:

```
from fractions import *
a=Fraction(2,-3)
print(-a)
```

Inverse

Pour obtenir l'inverse d'une fraction, on divise 1 par celle-ci:

```
from fractions import *
a=Fraction(5,4)
print(1/a)
```

Addition

La somme de deux fractions est une fraction:

```
from fractions import *
a=Fraction(34,21)
b=Fraction(21,13)
```

```
print(a+b)
```

Soustraction

La différence de deux fractions est une fraction:

```
from fractions import *
a=Fraction(34,21)
b=Fraction(21,13)
print(a-b)
```

Multiplication

Le produit de deux fractions est une fraction:

```
from fractions import *
a=Fraction(34,21)
b=Fraction(21,13)
print(a*b)
```

Division

Le quotient de deux fractions est une fraction (à condition que la deuxième ne soit pas nulle):

```
from fractions import *
a=Fraction(34,21)
b=Fraction(21,13)
print(a/b)
```

Le reste euclidien continue à être défini pour des fractions, et le résultat est une fraction:

```
from fractions import *
a=Fraction(32,7)
b=Fraction(7,2)
print(a%b)
```

Puissance

Si l'exposant est entier, la puissance d'une fraction est une fraction:

```
from fractions import *
a=Fraction(3,2)
print(a**12)
print(a**(-1))
```

Mais si l'exposant est un réel, la puissance n'est pas une fraction mais un réel:

```
from fractions import *
a=Fraction(9,4)
print(a**0.5)
```

Algorithmes

Réduite de Farey

La création de réduite de Farey est aisée avec le module *fractions* de *Python*:

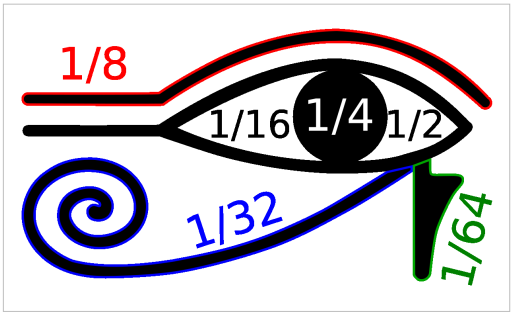
```
from fractions import *
def Farey(a,b):
    n=a.numerator+b.numerator
    d=a.denominator+b.denominator
    return Fraction(n,d)


a=Fraction(3,4)
b=Fraction(1,13)
print(Farey(a,b))
```

C'est si facile qu'on en vient tout de suite à la question suivante : à quoi ça peut bien servir ?

Si on répond que ça sert à fabriquer un arbre de Stern-Brocot, ça n'éclaire peut-être pas beaucoup, mais disons que ça sert à quelque chose de mathématique...

Fractions égyptiennes

Une fraction égyptienne est définie comme une somme d'inverses d'entiers. En effet les égyptiens avaient la réputation de ne pas utiliser de numérateurs. Toute fraction peut s'écrire comme une somme de fractions égyptiennes, et l'algorithme de Fibonacci permet d'en trouver un exemple à partir de la fraction. Dans la version *Python* ci-dessous, l'algorithme fournit une liste de fractions, toutes de numérateur 1, dont la somme est une fraction donnée *f*. Mais au cas où *f* serait supérieure à 1, on commence la liste par un entier:




```
from fractions import *
from math import *

def egypt(f):
      e=int(f)
      f-=e
      liste=[e]
      while(f.numerator>1):
            e=Fraction(1,int(ceil(1/f)))
            liste.append(e)
            f-=e
      liste.append(f)
```

```
      return liste

a=Fraction(21,13)
print(egypt(a))
```

Quelques explications sur le bricolage ci-dessus: Le dénominateur d'une fraction égyptienne est choisi entier (bien sûr) et plus grand que l'inverse de la fraction *f* (pour que l'algorithme converge). Une solution serait de prendre la troncature *int* de l'inverse de *f* et ajouter 1. Mais si l'inverse de *f* est entier, on ne doit pas ajouter 1 (sinon la suite est infinie). Alors on utilise la fonction *ceil*. Donc

1. Il a fallu importer le module *math* qui contient cette fonction *ceil*;
2. Du coup l'objet *ceil(1/f)* n'est plus un entier mais un réel, et ne peut plus être le dénominateur d'une fraction (message d'erreur de *Python*). Alors il faut convertir ce réel (qui est déjà entier, mais *Python* ne le sait pas) en entier, ce qui se fait par *int*.

Enfin, *Python* ne possédant pas de boucle *do..while*, il faut ajouter la dernière fraction égyptienne à la liste, pour que celle-ci soit complète.

En lançant le script ci-dessus, on apprend que $1+\frac{1}{2}+\frac{1}{9}+\frac{1}{234}=\frac{21}{13}$.

Nombres réels en Python

Écriture décimale

Si les fractions paraissent si abstraites, c'est sans doute à cause de l'écriture décimale qui est jugée plus concrète que l'écriture fractionnaire.

Nombres décimaux

Un nombre décimal est un nombre dont le développement décimal s'arrête. Un nombre non décimal (comme les premiers réels connus qui ne soient pas des fractions) a donc une infinité de chiffres, et ne peut donc pas être représenté de façon exacte en machine: On est obligé de travailler sur des approximations. Mais **même avec des nombres décimaux**, on travaille parfois sur des valeurs approchées: En effet les nombres sont représentés en base 2 dans la mémoire de l'ordinateur, et *Python* ne gère que les nombres binaires, ce que n'est pas 0,1.

Fractions

Une fraction se reconnaît à ce que son développement décimal est périodique à partir d'un certain rang.

```
print(1/3)
print(1/9)
print(1/11)
print(1/13)
```

(ces illustrations nécessitent la version 3.2 de *Python*. En cas de version inférieure, il faut remplacer les numérateurs par des 1.0 pour que la division se fasse dans $\mathbb{R}$).

Pour mieux voir ces chiffres, on peut en afficher plus:

```
from decimal import *
print(Decimal(1)/Decimal(3))
```

```
print(Decimal(1)/Decimal(9))
print(Decimal(1)/Decimal(11))
print(Decimal(1)/Decimal(13))
```

Irrationnels

Pour "construire" un nombre irrationnel (un réel qui ne soit pas une fraction), on peut donc inventer une suite de chiffres qui ne risque pas d'être répétitive, comme avec la constante de Champernowne ou un autre nombre univers.

Nombres algébriques

Le plus vieil irrationnel connu est sans doute la racine carrée de 2:

```
print(2**0.5)

#autre version:
from math import *
print(sqrt(2))
```

D'autres connus depuis longtemps sont le nombre d'or, la racine cubique de 2, etc. Les nombres *algébriques* sont définis comme solutions d'équations polynomiales, donc leurs valeurs décimales approchées sont calculées avec des méthodes comme la dichotomie, la méthode de Newton etc.

Nombres transcendants

Deux nombres transcendants très connus sont *e* et π :

```
from math import *
print(e)
print(pi)
```

Voici une manière de calculer une valeur approchée du nombre de Champernowne en *Python*:

```
c='0.'
for n in range(1,40):
    c+=str(n)


print(float(c))
```

Exercice pour un premier avril: Calculer une valeur approchée à 3 décimales près de la constante de Chaitin...

Fonctions réelles

Opérations

Les quatre opérations sont notées +, -, * et / en *Python*, et leur résultat (sauf si on divise par 0) est un réel. On peut aussi calculer le reste euclidien d'un réel par un réel!

```
from math import *
angle=100%pi
print(angle)
```

Le signe - désigne aussi l'opposé d'un réel. Sa valeur absolue est notée *abs*. Sa racine carrée *sqrt*.

Pour additionner *h* au nombre *x*, on peut écrire *x+=h* au lieu du classique *x=x+h*.

Puissances, exponentielles et logarithmes

Puissances et exposants

Comme pour les entiers, les puissances se notent avec l'astérisque de la multiplication, dédoublé. L'exemple du haut de la page montre comment ceci permet de calculer $\sqrt{2}$sans utiliser le module *math*.

Logarithmes

Le script ci-dessous calcule et affiche l'image de 0,5 par le logarithme népérien, par le logarithme décimal, par les fonctions réciproques du cosinus hyperbolique, du sinus hyperbolique, de la tangente hyperbolique:

```
print(log(0.5))
print(log10(0.5))
print(acosh(0.5))
print(asinh(0.5))
print(atanh(0.5))
```

Exponentielles

Pour calculer une puissance de 10, on peut utiliser la notation **.

Pour calculer l'exponentielle d'un nombre, on peut utiliser *exp*:

```
from math import *
a=e**pi
b=exp(pi)
print(a==b)
print(a-b)
```

Le script suivant calcule les cosinus, sinus et tangente hyperbolique de 2:

```
from math import *
print(cosh(2))
print(sinh(2))
print(tanh(2))
```

Fonctions trigonométriques

On peut convertir des angles de radians en degrés et vice-versa avec *degrees(x)* et *radians(x)*. Les fonctions trigonométriques directes se notent *cos*, *sin* et *tan* et ces trois fonctions sont en radians. Les fonctions inverses se notent *acos*, *asin* et *atan* et sont aussi en radians.

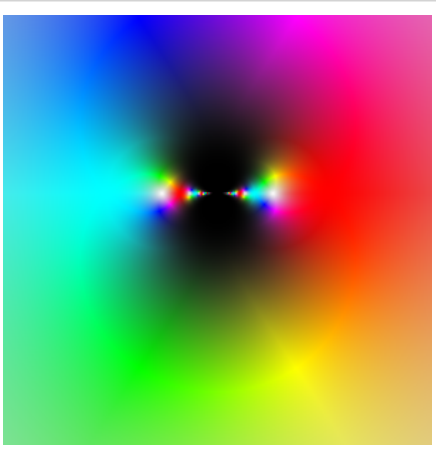
On peut calculer la fonction de deux variables $(x,y)\mapsto\sqrt{x^2+y^2}$avec *hypot*:

```
from math import *
a=sqrt(3**2+4**2)
b=hypot(3,4)
print(a==b)
```

Pour connaître l'angle aigu d'un triangle de côtés *x* et *y*, on peut, outre le calcul *atan(y/x)*, faire *atan2(x,y)*. Par exemple, si on veut connaître les angles et l'hypoténuse d'un triangle rectangle de côtés 12 cm et 5 cm, on peut utiliser ce script:

```
from math import *
a=12
b=5
print(degrees(atan2(a,b)))
print(degrees(atan2(b,a)))
print(hypot(a,b))
```

Nombres complexes en Python



Python est un langage très utilisé dans le domaine scientifique, comme le montre par exemple le choix de SAGE. Et les sciences, en particulier, font grand usage des nombres complexes, essentiellement depuis leur choix par Cauchy. Les physiciens et les électriciens notant *j* le nombre complexe dont le carré vaut -1, *Python* suit ce choix.

Instanciation d'un nombre complexe

Dans *Python*, il suffit d'écrire *complex(x,y)* pour avoir un nombre complexe:

```
z=complex(4,3)
print(z)
```

Même si les coordonnées *x* et *y* du point sont entières ou des fractions, elles deviennent des réels lorsque *Python* instancie le complexe. Voir les propriétés du complexe ci-dessous pour le vérifier.

Si on veut quand même que la lettre *i* désigne le complexe de carré -1, il suffit de le déclarer comme tel:

```
i=complex(0,1)
print(i**2)
```

Opérations

Les quatre opérations se notent respectivement +, -, * et /, et donnent toujours un complexe, même si celui-ci est réel (exemple de la soustraction ci-dessous):

```
a=complex(2,3)
b=complex(4,3)
print(a+b)
print(a-b)
print(a*b)
print(a/b)
```

L'élévation à un exposant se note de la même manière que pour les autres nombres, par **. Mais l'exposant peut même être un complexe!

```
i=complex(0,1)
print(i**i)
```

On constate que $i^i \in \mathbb{R}$ ...

La racine carrée d'un complexe peut aussi s'obtenir par une élévation de celui-ci à la puissance 0,5 mais dans ce cas on n'obtient qu'une seule des deux racines carrées:

```
c=complex(7,24)
print(c**0.5)
```

Mais *-4-3i* a aussi pour carré *7+24i*. Comment fait *Python* pour choisir entre les deux racines carrées?

Même -1 a deux racines carrées dans $\mathbb{C}$, et comme on s'en doute, *Python* ne choisit pas *-i* mais *i*... ou plutôt un complexe proche de celui-ci:

```
print((-1)**0.5)
```

Propriétés d'un nombre complexe

Les parties réelle et imaginaire d'un complexe sont des propriétés de l'objet:

```
z=complex(4,3)
print(z.real)
print(z.imag)
```

Par contre, le conjugué d'un complexe est une méthode de celui-ci:

```
z=complex(4,3)
print(z.conjugate())
```

(on remarque la présence des parenthèses après *conjugate*)

Forme trigonométrique

Pour avoir le module d'un nombre complexe, on entre *abs*:

```
z=complex(4,3)
print(abs(z))
```

Bien entendu, le résultat est réel.

Cependant, pour avoir l'argument de *a*, il faut charger le module (c'est le cas de le dire!) *cmath*:

```
from cmath import *
z=complex(4,3)
print(phase(z))
```

On remarque que *Python* utilise le mot *phase* et non le mot *argument*. *cmath* permet aussi de calculer d'un coup le module et l'argument d'un nombre complexe avec *polar*:

```
from cmath import *
z=complex(4,3)
print(polar(z))
```

Pour réaliser l'opération inverse (calculer l'exponentielle d'un nombre imaginaire), on utilise *rect*:

```
from cmath import *
print(rect(2,pi/3))
```

Par exemple, si on veut calculer le plus petit angle et l'hypoténuse d'un triangle rectangle de côtés 12 cm et 5 cm, on peut faire ceci:

```
from cmath import *
a=12
b=5
z=complex(a,b)
print(phase(z))
print(abs(z))
```

Fonctions

Avec *cmath*, on peut appliquer certaines fonctions de la variable réelle à des complexes.

Exponentielles

Pour vérifier numériquement que $e^{\frac{\pi}{3}i} = \frac{1}{2} + i\frac{\sqrt{3}}{2}$, on peut utiliser l'exponentielle d'un nombre complexe (en l'occurence, imaginaire):

```
from cmath import *
t=complex(0,pi/3)
z=exp(t)
print(z.real==0.5)
print(z.real-0.5)
print(z.imag==sqrt(3)/2)
```

On voit que la partie réelle n'est pas tout-à-fait égale à 0,5 (la différence est minime mais non nulle), c'est encore une conséquence de la représentation binaire des nombres en machine, puisque le développement binaire de 0,5 est infini, contrairement à son développement décimal.

Le script suivant calcule et affiche les fonctions trigonométriques hyperboliques d'un complexe:

```
from cmath import *
z=complex(4,3)
print(cosh(z))
print(sinh(z))
print(tanh(z))
```

Logarithmes

On peut même calculer le logarithme d'un nombre complexe:

Le script suivant calcule et affiche les fonctions trigonométriques hyperboliques d'un complexe:

```
from cmath import *
z=complex(4,3)
print(log(z))
```

Le script suivant calcule et affiche les arguments des fonctions trigonométriques hyperboliques d'un complexe:

```
from cmath import *
z=complex(4,3)
print(acosh(z))
print(asinh(z))
print(atanh(z))
```

Fonctions trigonométriques

Directes

Le script suivant calcule et affiche les fonctions trigonométriques d'un complexe:

```
from cmath import *
z=complex(4,3)
print(cos(z))
print(sin(z))
print(tan(z))
```

Inverses

Le script suivant calcule et affiche les arcs des fonctions trigonométriques d'un complexe:

```
from cmath import *
z=complex(4,3)
print(acos(z))
print(asin(z))
print(atan(z))
```

Quaternions et octonions en Python

Complexes

On a vu dans le chapitre précédent que pour Python, un nombre complexe z est essentiellement une structure abritant deux réels, accessibles par *z.real* et *z.imag* respectivement. La construction de Cayley-Dickson généralise ce point de vue: En prenant deux complexes a et b, on peut les regrouper dans une nouvelle structure qui est considérée comme un nombre: Un quaternion.

Pour toute la suite, il est conseillé d'importer les fonctions du module *math* de Python:

```
from math import *
```

Mais en réalité, seule la méthode *hypot* sera utilisée, ce qui signifie que le minimum nécessaire était

```
from math import hypot
```

Quaternions

Définition

Toutes les méthodes (qui permettent de manipuler les quaternions) peuvent être regroupées dans une classe nommée *Quaternion*:

```
class Quaternion:

      def __init__(self,a,b):
            self.a=a
            self.b=b
```

La première méthode, l'initialisation, crée donc deux variables a et b (qui seront des complexes, mais Python ne le sait pas encore) et les rendre accessibles par la notation avec un point, les nombres $q.a$ et $q.b$ (les deux complexes qui définissent le quaternion q) étant des propriétés du quaternion.

Affichage

Pour y voir quelque chose, une méthode d'affichage est nécessaire. Comme Python en possède déjà une (la conversion en chaîne de caractères, ou *string*, notée *__str__*), on va la **surcharger**:

```
      def __str__(self):
            aff='('
            aff+=str(self.a.real)+')+('
            aff+=str(self.a.imag)+')i+('
            aff+=str(self.b.real)+')j+('
            aff+=str(self.b.imag)+')k'
            return aff
```

Un quaternion possède deux propriétés qui sont des nombres complexes, mais chacun d'eux est formé de deux nombres réels, donc un quaternion est formé de 4 nombres réels, ce sont ceux qui sont affichés par la méthode ci-dessus, lorsqu'on entre par exemple *print(q)*.

Fonctions

Opposé

```
      def __neg__(self):
            return Quaternion(-self.a,-self.b)
```

En écrivant *-q*, on aura désormais l'opposé de *q*.

Module

```
      def __abs__(self):
            return hypot(abs(self.a),abs(self.b))
```

Grâce à cette méthode, *abs(q)* retourne un réel, la racine carrée de sa norme.

Conjugué

```
      def conjugate(self):
            return Quaternion(self.a.conjugate(),-self.b)
```

abs(q.conjugate()) renvoie le même résultat que *abs(q)* parce que tout quaternion a la même norme que son conjugué.

Opérations

Addition

Pour additionner deux quaternions, on additionne leurs *a* respectifs, et leurs *b* respectifs:

```
      def __add__(self,other):
            return Quaternion(self.a+other.a,self.b+other.b)
```

Soustraction

```
      def __sub__(self,other):
            return Quaternion(self.a-other.a,self.b-other.b)
```

Multiplication

Le produit de deux quaternions est plus difficile à définir:

```
      def __mul__(self,other):
            c=self.a*other.a-self.b*other.b.conjugate()
            d=self.a*other.b+self.b*other.a.conjugate()
            return Quaternion(c,d)
```

Ce produit admet le quaternion *Quaternion(1,0)* comme élément neutre, et il est associatif comme tout produit qui se respecte. Mais il n'est pas commutatif:

```
p=Quaternion(-2+1J,2+3J)
q=Quaternion(3-2J,5+1J)
print(p*q)
print(q*p)
```

Division

Multiplication par un réel

Pour définir le plus facilement possible le quotient de deux quaternions, on a intérêt à définir le produit d'un quaternion par un réel (on peut déjà l'effectuer avec la méthode de produit, en assimilant le réel *r* avec la quaternion *Quaternion(r,0)*). Mais comme le symbole de multiplication est déjà utilisé pour la multiplication des quaternions, il en faut un autre pour la multiplication d'un quaternion par un réel. Or il se trouve que *__rmul__* (multiplication à l'envers) est encore disponible, donc pour peu qu'on multiplie à droite dans la définition, et à gauche en pratique, on peut ajouter cette méthode:

```
	def __rmul__(self,k):
		return Quaternion(self.a*k,self.b*k)
```

Alors, pour tripler un quaternion *q*, on peut faire, au choix, *q*Quaternion(3,0)*, ou *3*q*.

Division

Le quotient de deux quaternions est désormais simple à définir:

```
	def __div__(self,other):
		return self.conjugate()*(1./abs(other)**2*other)
```

Le quotient d'un quaternion par son conjugué est de norme 1:

```
p=Quaternion(-2+1J,2+3J)
print(p/p.conjugate())
```

Cet exemple révèle que $\left(-\frac{5}{9}\right)^2+\left(\frac{2}{9}\right)^2+\left(\frac{4}{9}\right)^2+\left(\frac{6}{9}\right)^2=1$, ce qui revient à la décomposition suivante de 81 (un carré) comme somme de 4 carrés: $5^2+2^2+4^2+6^2=25+4+16+36=81=9^2$.

Puissances

Même si la multiplication des quaternions n'est pas commutative, elle est associative, c'est tout ce qu'il faut pour définir les puissances des quaternions à exposants entiers (et donc, par formules de Taylor, de la trigonométrie et des exponentielles de quaternions):

```
	def __pow__(self,n):
		r=1
		for i in range(n):
			r=r*self
		return r
```

Par exemple, on peut calculer le carré d'un quaternion:

```
q=Quaternion(2J/7,(3+6J)/7)
print(q**2)
```

Plus généralement, l'ensemble des solutions de $q^2=-1$ est une sphère.

L'étude des itérés de q^2+c où q et c sont des quaternions, mène à des ensembles de Mandelbrot quaternionniques ([1])

Résumé

Voici le contenu complet de la classe *Quaternion* de Python:

```
from math import hypot

class Quaternion:

      def __init__(self,a,b):
            self.a=a
            self.b=b

      def __str__(self):
            aff='('
            aff+=str(self.a.real)+')+('
            aff+=str(self.a.imag)+')i+('
            aff+=str(self.b.real)+')j+('
            aff+=str(self.b.imag)+')k'
            return aff

      def __neg__(self):
            return Quaternion(-self.a,-self.b)

      def __add__(self,other):
            return Quaternion(self.a+other.a,self.b+other.b)

      def __sub__(self,other):
            return Quaternion(self.a-other.a,self.b-other.b)

      def __mul__(self,other):
            c=self.a*other.a-self.b*other.b.conjugate()
            d=self.a*other.b+self.b*other.a.conjugate()
            return Quaternion(c,d)

      def __rmul__(self,k):
            return Quaternion(self.a*k,self.b*k)

      def __abs__(self):
            return hypot(abs(self.a),abs(self.b))

      def conjugate(self):
            return Quaternion(self.a.conjugate(),-self.b)

      def __div__(self,other):
            return self.conjugate()*(1./abs(other)**2*other)

      def __pow__(self,n):
            r=1
```

```
            for i in range(n):
                  r=r*self
            return r
```

Il suffit de placer tout ça dans un fichier appelé *quaternions.py* et disposer de toutes ces méthodes en important le module nouvellement créé par

```
from quaternions import *
```

Ce qui permet alors de déclarer des quaternions, puis d'effectuer des opérations dessus.

Octonions

Ce qui est intéressant avec la construction de Cayley-Dickson utilisée ci-dessus pour les quaternions, c'est qu'elle se généralise: En définissant une structure (un objet) comprenant deux quaternions a et b, on définit un octonion.

Définition et affichage

Définition

```
from math import hypot

class Octonion:

      def __init__(self,a,b):
            self.a=a
            self.b=b
```

Affichage

Comme $\mathbb{O}$ est de dimension 8 sur $\mathbb{R}$, l'affichage est plus compliqué que celui des quaternions:

```
      def __str__(self):
            aff='('
            aff+=str(self.a.a.real)+')+('
            aff+=str(self.a.a.imag)+')i+('
            aff+=str(self.a.b.real)+')j+('
            aff+=str(self.a.b.imag)+')k+('
            aff+=str(self.b.a.real)+')l+('
            aff+=str(self.b.a.imag)+')li+('
            aff+=str(self.b.b.real)+')lj+('
            aff+=str(self.b.b.imag)+')lk'
            return aff
```

On voit une arborescence apparaître, le *a.a.real* désignant la partie réelle du a du quaternion a de l'octonion. La notation avec les points prend ici tout son intérêt, permettant une concision pythonienne qui rendrait presque apprivoisés ces redoutables octonions!

Fonctions

Les fonctions sur les octonions se définissent presque comme celles sur les quaternions, Cayley-Dickson oblige:

Opposé

```
      def __neg__(self):
            return Octonion(-self.a,-self.b)
```

Module

```
      def __abs__(self):
            return hypot(abs(self.a),abs(self.b))
```

C'est pour permettre cette concision qu'on a importé la méthode *hypot* du module *math*.

Conjugué

```
      def conjugate(self):
            return Octonion(self.a.conjugate(),-self.b)
```

Opérations

Addition

```
      def __add__(self,other):
            return Octonion(self.a+other.a,self.b+other.b)
```

Encore une fois, le fait d'avoir surchargé la méthode *__add__* de Python permet de noter simplement $m+n$ la somme des octonions m et n.

Soustraction

```
      def __sub__(self,other):
            return Octonion(self.a-other.a,self.b-other.b)
```

Multiplication

```
      def __mul__(self,other):
            c=self.a*other.a-other.b*self.b.conjugate()
            d=self.a.conjugate()*other.b+other.a*self.b
            return Octonion(c,d)
```

Non seulement la multiplication des octonions n'est pas commutative, elle n'est plus associative non plus:

```
m=Octonion(Quaternion(3+4J,2-7J),Quaternion(1+3J,5-3J))
n=Octonion(Quaternion(2+1J,1-3J),Quaternion(2-2J,1+1J))
o=Octonion(Quaternion(3-2J,-5+3J),Quaternion(1-2J,2-1J))
print((m*n)*o)
print(m*(n*o))
```

Division

Grâce à la notion de conjugué, on peut facilement définir le quotient de deux octonions, mais c'est encore plus facile en multipliant un octonion par un réel:

Produit par un réel

```
      def __rmul__(self,k):
            return Octonion(k*self.a,k*self.b)
```

Quotient de deux octonions

```
      def __div__(self,other):
            return self.conjugate()*(1./abs(other)**2*other)
```

Là encore, le quotient d'un octonion par son conjugué est de norme 1:

```
m=Octonion(Quaternion(3+4J,2-7J),Quaternion(1+3J,5-3J))
n=Octonion(Quaternion(2+1J,1-3J),Quaternion(2-2J,1+1J))

print(m/m.conjugate())
print(abs(n/n.conjugate()))
```

Ces calculs permettent de décomposer un carré en somme de 8 carrés.

Puissances

Comme la multiplication des octonions n'est pas associative, les puissances des octonions ne présentent guère d'intérêt, sauf pour la puissance 2, et sur $\mathbb{O}$, l'ensemble des solutions de l'équation $z^2 = -1$est une sphère de dimension 6.

Résumé

La classe *Octonion* de Python peut se résumer à ceci:

```
class Octonion:

      def __init__(self,a,b):
            self.a=a
            self.b=b

      def __str__(self):
            aff='('
            aff+=str(self.a.a.real)+')+('
            aff+=str(self.a.a.imag)+')i+('
            aff+=str(self.a.b.real)+')j+('
            aff+=str(self.a.b.imag)+')k+('
            aff+=str(self.b.a.real)+')l+('
            aff+=str(self.b.a.imag)+')li+('
            aff+=str(self.b.b.real)+')lj+('
            aff+=str(self.b.b.imag)+')lk'
            return aff
```

```
    def __neg__(self):
        return Octonion(-self.a,-self.b)

    def __add__(self,other):
        return Octonion(self.a+other.a,self.b+other.b)

    def __sub__(self,other):
        return Octonion(self.a-other.a,self.b-other.b)

    def __mul__(self,other):
        c=self.a*other.a-other.b*self.b.conjugate()
        d=self.a.conjugate()*other.b+other.a*self.b
        return Octonion(c,d)

    def __rmul__(self,k):
        return Octonion(k*self.a,k*self.b)

    def __abs__(self):
        return hypot(abs(self.a),abs(self.b))

    def conjugate(self):
        return Octonion(self.a.conjugate(),-self.b)

    def __div__(self,other):
        return self.conjugate()*(1./abs(other)**2*other)
```

Pour peu qu'on l'ait enregistrée dans un fichier *octonions.py*, il suffit pour pouvoir effectuer des calculs sur les octonions, d'importer ce fichier par

```
from octonions import *
```

Bibliographie

- De par leur utilité en infographie 3D, les quaternions sont utilisés dans Blender, avec cette description: [2]
- Sur les octonions, le livre de John Baez est une lecture hautement conseillée: [3]

Références

[1] http://en.wikibooks.org/wiki/Pictures_of_Julia_and_Mandelbrot_Sets/Quaternions
[2] http://www.zoo-logique.org/3D.Blender/scripts_python/API/Mathutils.Quaternion-class.html
[3] http://math.ucr.edu/home/baez/octonions/

Ensembles en Python

Dans la théorie des probabilités telle qu'elle a été axiomatisée par Kolmogorov, un évènement est noté par la liste des éventualités qui le réalisent, notée entre accolades.

Représentation des évènements en Python

Évènements certain et impossible

L'évènement impossible est noté $\emptyset$.

L'évènement certain (noté Ω) est la liste de toutes les éventualités. Pour savoir si un élément est dans un ensemble, on utilise le mot-clé *in* comme dans *6 in omega* qui est un booléen.

Avec un dé

On s'apprête à lancer un dé. Alors l'évènement "le résultat sera plus petit que 5" est décrit par l'ensemble $\{1,2,3,4\}$. De même, l'évènement "le résultat sera pair" est représenté par $\{2,4,6\}$.
En *Python* cela donne:

```
univers={1,2,3,4,5,6}
petit={1,2,3,4}
pair={2,4,6}
```

Avec des cartes

Cette fois-ci, on extrait au hasard une carte parmi un jeu de 32 cartes.

Faire la liste des 32 cartes (pour constituer l'univers) est un peu fastidieux, alors on va laisser *Python* le faire:


```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers={0}
for v in valeurs:
    for c in couleurs:
        univers.add(str(v)+' '+c)
```

```
univers.remove(0)
print(len(univers))
```

Il a été nécessaire de mettre initialement un *0* dans l'univers, puis de l'enlever à la fin. C'est pour tromper le typage faible de *Python* qui considère les accolades vides comme un objet de type *dictionnaire* et non comme un ensemble. De plus, on transforme les valeurs des cartes en texte même si ce sont des nombres.

L'évènement "la carte est une figure" (pas un nombre) se construit par

```
couleurs={'carreau','coeur','pique','trefle'}
figure={0}
for v in {'Valet','Dame','Roi'}:
    for c in couleurs:
        figure.add(v+' '+c)
figure.remove(0)
print(univers)
```

Et l'évènement "la carte est un pique" se construit de manière analogue:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
pique={0}
for v in valeurs:
    pique.add(str(v)+' pique')
pique.remove(0)
print(pique)
```

Calcul d'évènements

Évènements simultanés

Notation

L'évènement "A et B" se note $A \cap B$, et l'opération se note en *Python* par une esperluette (*&*) qui est d'ailleurs une ancienne représentation du mot *et* en latin.

Avec le dé

```
univers={1,2,3,4,5,6}
petit={1,2,3,4}
pair={2,4,6}
print(petit&pair)
```

Avec les cartes

```
print(figure&pique)
```

L'affichage confirme qu'il n'y a que trois cartes qui sont à la fois des figures et des piques: Les trois figures de pique Ogier, Pallas et David.

Disjonction

Notation

De même l'évènement "A ou B" se note $A \cup B$, et en *Python*, le symbole *pipe* (trait vertical). *Python* enlève automatiquement les doublons.

Avec le dé

```
univers={1,2,3,4,5,6}
petit={1,2,3,4}
pair={2,4,6}
print(petit|pair)
```

Avec les cartes

```
print(figure|pique)
print(len(figure|pique))
```

On peut compter les 17 cartes à la main, mais le comptage par *Python* est plus sûr. On constate que pour *Python*, le nombre d'éventualités d'un évènement s'appelle sa longueur.

Contraire d'un évènement

Pour calculer le contraire d'un évènement, on le soustrait à l'univers.

Avec le dé

```
univers={1,2,3,4,5,6}
petit={1,2,3,4}
pair={2,4,6}
print(univers-petit)
print(univers-pair)
```

On constate que le contraire de *pair* est *impair*...

Avec les cartes

```
print(univers-figure)
print(univers-pique)
```

Probabilités

On a vu ci-dessus que pour *Python*, le nombre d'éléments d'un évènement est appelé sa longueur. On peut alors définir la probabilité d'un évènement comme le quotient de sa longueur par celle de l'univers.

Avec le dé

```
def proba(evenement):
    return len(evenement)/len(univers)



print(proba(petit))
print(proba(pair))
print(proba(univers))


p1=proba(petit)+proba(pair)-proba(petit&pair)
p2=proba(petit|pair)
print(p1==p2)
```

Avec les cartes

```
def proba(evenement):
    return len(evenement)/len(univers)



print(proba(figure))
print(proba(pique))
print(proba(univers))

p1=proba(figure)+proba(pique)-proba(figure&pique)
p2=proba(figure|pique)
print(p1==p2)
```

Nombres pseudoaléatoires en Python

Pour faire des simulations en proba, on a besoin de nombres pseudoaléatoires.

Obtention

Recette

Versions jusqu'à la 2.7

Python 2.7 calculait les nombres pseudo-aléatoires avec l'algorithme de Wichmann-Hill, basé sur 3 générateurs congruentiels linéaires en parallèle (extrait du code source - on remarquera l'allusion à Fermat dans l'avant-dernière ligne):

```
        # This part is thread-unsafe:
        # BEGIN CRITICAL SECTION
        x, y, z = self._seed
        x = (171 * x) % 30269
        y = (172 * y) % 30307
        z = (170 * z) % 30323
        self._seed = x, y, z
        # END CRITICAL SECTION

        # Note:  on a platform using IEEE-754 double arithmetic, this
can
        # never return 0.0 (asserted by Tim; proof too long for a
comment).
        return (x/30269.0 + y/30307.0 + z/30323.0) % 1.0
```

Ceci suggère quelques exercices d'arithmétique:

1. Vérifier que les modules 30269, 30307 et 30323 sont premiers.
2. Vérifier que les multiplicateurs 171, 172 et 170 sont primitifs modulo leurs modules respectifs.
3. Simuler l'un des générateurs congruentiels (*x* par exemple est une suite géométrique de raison 171 dans le corps de Galois $\mathbb{Z}/30269\mathbb{Z}$).
4. Il résulte du point 2 ci-dessus que la suite *x* est périodique de période 30268. L'addition de la dernière ligne donnerait une période de $30268 \times 30306 \times 30322$. Calculer ce nombre et le décomposer en facteurs premiers...

Version 3.2

Python 3.2 utilise le Mersenne twister, implémenté en C. La période est annoncée comme égale à $2^{19\,937} - 1$. Démontrer la primalité de ce nombre de Mersenne est difficile, mais un exercice intéressant est de démontrer la primalité de son exposant 19 937.

Une fois qu'on sait construire un nombre entier pseudo-aléatoire compris entre 0 et un entier fixe N-1, il suffit de le diviser par N pour simuler un réel pseudo-aléatoire compris entre 0 (inclus en théorie) et 1 (exclu). Ce réel pseudo-aléatoire est celui appelé *random* et sa loi est uniforme sur l'intervalle allant de 0 à 1. Une transformation affine permet alors d'obtenir à partir de lui un réel pseudo-aléatoire uniforme entre deux réels donnés, puis une troncature permet de retrouver des entiers aléatoires, modulo un nombre entier donné.

Le module *random* de *Python 3.2* ne sait pas simuler de variables aléatoires binomiales ou de Poisson, mais il fournit des variables aléatoires normales avec l'algorithme de Kinderman et Monahan:

```
NV_MAGICCONST = 4 * _exp(-0.5)/_sqrt(2.0)



    def normalvariate(self, mu, sigma):
        random = self.random
        while 1:
            u1 = random()
            u2 = 1.0 - random()
            z = NV_MAGICCONST*(u1-0.5)/u2
            zz = z*z/4.0
            if zz <= -_log(u2):
                break
        return mu + z*sigma
```

Il est basé sur deux variables aléatoires *u1* et *u2* uniformes entre 0 et 1 (sauf que *u2* ne peut être nulle). *u1* est ensuite centrée, et divisée par *u2*. Le tout est normalisé avec la *constante magique* $4\frac{e^{-\frac{1}{2}}}{\sqrt{2}}$, élevé au carré et on finit par lui appliquer une transformation affine pour que ses espérance et écart-type soient ceux désirés.

En plus, une variable aléatoire normale centrée reduite peut être obtenue avec *gauss* qui utilise l'algorithme de Box-Muller.

Python 3.2 possède aussi des simulateurs de tirage sans remise, et même de permutations aléatoires, où on mélange les cartes en multipliant les échanges de deux cartes (technique analogue à celle consistant à couper le jeu plusieurs fois de suite):

```
        for i in reversed(range(1, len(x))):
            # pick an element in x[:i+1] with which to exchange x[i]
            j = int(random() * (i+1))
            x[i], x[j] = x[j], x[i]
```

Cet algorithme illustre le fait que toute permutation est engendrée par des transpositions.

Syntaxe

Toutes ces merveilles se trouvant dans le module *random*, on doit charger celui-ci pour faire des simulations avec *Python*:

```
from random import *
```

Variables uniformes

Continue

Le plus simple à obtenir c'est le fameux nombre pseudo-aléatoire entre 0 et 1:

```
from random import *
print(random())
```

Pour avoir un nombre entre -2 et 3, on peut faire

```
from random import *
print(uniform(-2,3))
```

Entière

Pour lancer un dé, on peut utiliser l'une des méthodes suivantes:

```
from random import *
print(randint(1,6))
print(randrange(1,7))
print(choice(range(1,6)))
from math import *
print(ceil(6*random()))
```

Variables continues

La loi exponentielle de paramètre quelconque (ici 3) peut être simulée:

```
from random import *
print(expovariate(3))
```

Une variable gaussienne peut être simulée par deux méthodes (ici centrée et réduite):

```
from random import *
print(normalvariate(0,1))
print(gauss(0,1))
```

random simule aussi d'autres lois continues comme les lois Beta, Gamma, de Weibull etc.

Variables binomiales et de Poisson

On l'a vu plus haut, *random* ne simule pas de variables binomiales. Mais une variable binomiale de paramètres n et p pouvant être définie comme somme de n variables de Bernoulli de probabilité p indépendantes entre elles, est simulable par boucle:

```
from random import *

def Bernoulli(p):
    if random()<p:
        return 1
    else:
        return 0


def binomial(n,p):
    somme=0
    for k in range(n):
        somme+=Bernoulli(p)
    return somme


print(binomial(25,0.2))
```

Cet algorithme nécessite d'être amélioré, en effet rien n'empêche de l'appeler avec des valeurs de p supérieures à 1 ou des valeurs de n non entières.

Pour simuler une loi de Poisson, on peut utiliser une variable binomiale qui l'approche, comme par exemple

```
def Poisson(l):
    return binomial(1000000,l/1000000)
```

mais c'est très long à calculer. Une méthode plus rapide utilise la loi exponentielle de même paramètre:

```
from random import *
def Poisson(l):
    t=1.0/expovariate(l)
    return int(t)
```

Permutations et arrangements

Permutations

En 1741, dans les *Mémoires de l'Académie des Sciences de Berlin*, Leonhard Euler publiait un article titré *Calcul de la probabilité du jeu de rencontre*. Le nom initial du jeu de rencontre était *jeu de treize* parce qu'il se jouait à 13 cartes. Mais Euler généralise le jeu à n cartes.

Il le définit ainsi:

Le jeu de rencontre est un jeu de hasard, où deux personnes ayant chacune un entier jeu de cartes, en tirent à la fois une carte après l'autre, jusqu'à ce qu'il arrive, qu'elles rencontrent la même carte: et alors l'une des deux personnes gagne. Or, lorsqu'une telle rencontre n'arrive point du tout, alors c'est l'autre des deux personnes qui gagne. Cela posé, on demande la probabilité, que l'une et l'autre de ces deux personnes aura de gagner.

Dans cet excellent article (16 pages en Français),

Euler montre que

Pourvu donc que le nombre de cartes ne soit pas moindre que 12, l'espérance de A sera toujours à celle de B à peu près comme 12 à 7 ... Ou bien parmi 19 jeux qu'on joue, il y en aura probablement 12 qui font gagner A, et 7 qui feront gagner B.

On constate qu'Euler utilisait le mot *espérance* là où aujourd'hui on écrit *probabilité*, et sa conclusion s'écrit algébriquement $P(A) \simeq \frac{12}{19}$et $P(B) \simeq \frac{7}{19}$, expliquant pourquoi dans les diligences de l'époque, les gens pariaient à 12 contre 7 sur une rencontre dans ce jeu.

Pour vérifier cela avec 2 jeux de 32 cartes, ça peut prendre du temps (bien que le jeu s'arrête dès la première rencontre). Alors la simulation offerte par *Python* permet de rapidement simuler un grand nombre de parties (*Python* est une sorte de diligence supersonique).

Pour mélanger un jeu de 32 cartes, on doit d'abord le construire, avec la technique vue précédemment, sauf que le mélange ne peut être fait que sur une liste et pas sur un ensemble:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
jeu1=[str(v)+' '+c for v in valeurs for c in couleurs]
jeu2=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *

shuffle(jeu2)
```

```
diagnostic='pas de rencontre'
for i in range(32):
    if jeu1[i]==jeu2[i]:
        diagnostic='rencontre sur la '+str(i)+'ème carte, qui est le
'+jeu1[i]
        break

print(diagnostic)
```

On a créé deux jeux de 32 cartes, mélangé l'un des deux (*jeu2*) puis comparé carte à carte les deux jeux. Par défaut le texte de sortie est *pas de rencontre*. En effet c'est seulement s'il y a une rencontre qu'il est modifié, et remplacé par le nom et le numéro de la carte commune aux deux jeux. L'instruction *break* sert à éviter de continuer à jouer après la fin du jeu.

Tirage sans remise

Un magicien demande à une personne de l'assistance de tirer une carte d'un jeu de 32. Quelle est la probabilité que ce soit l'as de pique?

Pour simuler l'expérience, on va utiliser la fonction *choice* qui permet de tirer une carte. On va alors construire le jeu de cartes comme dans l'article précédent, à ceci près que *choice* attend une liste et non un ensemble:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *
for n in range(100):
    if choice(univers)=="1 pique":
        print('victoire !')
```

Une estimation de la probabilité de l'évènement peut alors se faire en comptant le nombre de fois que le mot *victoire* a été écrit: Il est, en pourcents, la fréquence de l'évènement.

Pour jouer au poker, on peut simuler le choix d'une main par un tirage de 5 éléments (sans répétition) parmi les 32:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *

hand=sample(univers,5)
print(hand)
```

Simulation avec Python

Avec des nombres pseudo-aléatoires et des boucles qui permettent de répéter un grand nombre de fois une expérience élémentaire, on peut simuler des phénomènes aléatoires, et utiliser la loi des grands nombres pour estimer des probabilités (parfois difficiles voire impossibles à calculer).

Pour faire des statistiques, on a besoin de tableaux (d'effectifs). Une difficulté supplémentaire apparaît alors: la première valeur d'un tableau est donnée par l'indice 0 et non l'indice 1 de celui-ci. Ce qui oblige parfois à des décalages d'indice.

Lancers de dés

Un dé

Pour vérifier que le dé virtuel défini par *Python* est équilibré, on peut le lancer un grand nombre de fois (par exemple 6000) et compter les différents résultats:

```
from random import *

effectifs=[0,0,0,0,0,0]
for n in range(6000):
    dice=randint(1,6)
    effectifs[dice-1]+=1

print(effectifs)
```

On peut simplifier la création du tableau en multipliant par 6 un tableau d'une seule case:

```
from random import *

effectifs=[0]*6
for n in range(6000):
    dice=randint(1,6)
    effectifs[dice-1]+=1

print(effectifs)
```

Deux dés

Pour étudier la somme des résultats donnés par deux dés indépendants l'un de l'autre (voir par exemple si elle est équidistribuée), on fait comme avec un seul dé sauf qu'on a deux dés, et qu'on les additionne:

```
from random import *

effectifs=[0]*11
for n in range(6000):
    de1=randint(1,6)
    de2=randint(1,6)
    twodice=de1+de2
    effectifs[twodice-2]+=1
```

```
print(effectifs)
```

Avec des cartes

Une carte

On tire une carte d'un jeu de 32. Pour estimer la probabilité que ce soit l'as de pique, on répète 3200 fois l'expérience, et on divise le nombre de parties gagnées par 3200:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers=[str(v)+' '+c for v in valeurs for c in couleurs]
from random import *

somme=0
for n in range(3200):
    if choice(univers)=="1 pique":
        somme+=1

print(somme/3200)
print(1/32)
```

Une main

On tire 5 cartes d'un jeu de 32. Quelle est la probabilité des évènements suivants:

1. On a une couleur (les 5 cartes sont de la même couleur);
2. On a un carré d'as (4 des 5 cartes sont des as)?

On reconnaît la couleur d'une carte en regardant les deux dernières lettres de son nom:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *



somme=0
for n in range(1000000):
    main=sample(univers,5)
    couleurs_dans_main={0}
    for carte in main:
        couleurs_dans_main.add(carte[-2:])
    if len(couleurs_dans_main)==1:
        somme+=1

print(somme/1000000)
```

Les couleurs sont très rares!

On reconnaît un as à ce que son nom commence par un *1* non suivi par un *0* (sinon ce serait un 10). On compte les as de chaque main, et on compte combien de fois on en a 4 (un carré):

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
univers=[str(v)+' '+c for v in valeurs for c in couleurs]
from random import *


somme=0
for n in range(10000):
    main=sample(univers,5)
    NombreAs=len([carte for carte in main if carte[0:2]=='1 '])
    if NombreAs==4:
        somme+=1

print(somme/10000)
```

Jeu de rencontre

On cherche à estimer expérimentalement la probabilité d'une "rencontre" avec deux jeux de 32 cartes (qu'à un moment donné, les deux joueurs, dont l'un a mélangé son jeu, déposent la même carte sur la table). Pour cela, on répète 19000 fois le jeu de rencontre, et on compte combien de rencontres on a eu:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
jeu1=[str(v)+' '+c for v in valeurs for c in couleurs]
jeu2=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *

rencontres=0
for n in range(19000):
    shuffle(jeu2)
    for i in range(32):
        if jeu1[i]==jeu2[i]:
            rencontres+=1
            break

print(rencontres/19000)
print(12/19)
```

Méthode de Monte-Carlo

Pour calculer π par la méthode de Monte-Carlo, on "crée" un nuage de points à coordonnées uniformes entre 0 et 1, et on compte combien d'entre eux sont à une distance de l'origine inférieure à l'unité. La fréquence de ces points converge vers $\frac{\pi}{4}$:

```
from math import hypot
from random import random
p=len([n for n in range(1000000) if hypot(random(),random())<1])

print(p/1000000*4)
```

Heureusement, il y a des moyens plus rapides pour calculer π !

Statistique inférentielle avec Python

En statistique inférentielle, on cherche à connaître l'inconnu. Pour ce faire, on émet des hypothèses ou on estime des grandeurs partiellement inconnues,

1. On se fixe des probabilités *a priori* de se tromper dans ses estimations ou son test;
2. On cherche (difficile) pour quels choix des paramètres (intervalle ou estimateur) ces probabilités sont atteintes;
3. On prend ses décisions à partir des choix faits ci-dessus.

La simulation permet de prendre le problème à l'envers, en créant un modèle qui se comporte comme l'échantillon qu'on observe, et en estimant les probabilités difficiles à calculer par des fréquences. Il arrive que, de ces simulations, on puisse inférer (on est là pour ça) des conjectures, qui serviront à élaborer les algorithmes de test ou d'estimation. Certes, dire *on prend 2 parce que la simulation a suggéré que, pour 2, la probabilité est de 0,95*, ce n'est pas très mathématique, mais la théorie qui disait elle aussi *on prend 1,96; concrètement 2* est très souvent basée sur des approximations normales de lois compliquées. On va donc expérimenter et observer sans expliquer ce qu'on observe, mais en se servant de ces observations pour expliquer pourquoi les statisticiens font comme ci et pas comme ça.

Estimations

Pour faire des statistiques, il faut un échantillon de données aléatoires ou non. Et pour avoir des données sous *Python*, le plus simple est de les fabriquer sous *Python*. Par exemple si on veut faire des statistiques sur les 100 premiers carrés d'entiers, on peut fabriquer une liste contenant ces 100 nombres:

```
donnees=[n**2 for n in range(100)]

print(len(donnees))
```

Moyenne

Pour calculer la moyenne des nombres qui sont dans *donnees*, on les additionne et on divise la somme par le nombre de nombres qu'il y a dans *donnees*:

```
def moyenne(tableau):
    return sum(tableau, 0.0) / len(tableau)

print(moyenne(donnees))
```

L'algorithme est améliorable puisque si une donnée n'est pas numérique, il ne donne qu'un message d'erreur.

Variance

La variance est définie comme la moyenne des carrés des écarts à la moyenne:

```
def variance(tableau):
    m=moyenne(tableau)
    return moyenne([(x-m)**2 for x in tableau])

print(variance(donnees))
```

Une variante pour la variance est donnée par la formule de Huyghens: Moyenne des carrés moins le carré de la moyenne.

Écart-type

L'écart-type est défini comme la racine carrée de la variance:

```
def ecartype(tableau):
    return variance(tableau)**0.5

print(ecartype(donnees))
```

Échantillons

On peut créer un échantillon de 100 nombres gaussiens d'espérance 16 et d'écart-type 2, puis calculer sa moyenne et son écart-type:

```
from random import *
echantillon=[gauss(16,2) for n in range(100)]

print(moyenne(echantillon))
print(ecartype(echantillon))
```

On voit que la moyenne est proche de 16 et l'écart-type proche de 2. C'est rassurant. Mais si on y regarde de plus près, on voit un problème: En prenant des échantillons plus petits, on s'attend à ce que leurs moyenne et écart-type fluctuent mais que la moyenne des moyennes (sur beaucoup de petits échantillons) soit 16 et que la moyenne des écarts-types soit proche de 2. C'est vrai pour la moyenne des moyennes mais visiblement pas pour la moyenne des écarts-types:

```
from random import *
m=[] #liste des moyennes des echantillons
```

```
s=[] #liste des ecarts-types des echantillons

for n in range(10000):
    echantillon=[gauss(16,2) for k in range(5)]
    m.append(moyenne(echantillon))
    s.append(ecartype(echantillon))


print(moyenne(m)) # Voisin de 16, c'est rassurant!
print(moyenne(s)) # Largement plus petit que 2!
print(moyenne(s)*2**0.25)
print(ecartype(m)) # Le moyennage resserre les ecarts-types
print(2/5**0.5) # en les divisant par la racine de la taille de
l'echantillon
```

En théorie, le nombre par lequel on doit multiplier la moyenne des écarts-types pour estimer l'écart-type de la population est $\sqrt{\frac{5}{4}}$. Ce n'est pas le cas ici: Il semble que l'algorithme de *Python* pour effectuer des tirages avec remise introduise un biais. Par contre on découvre expérimentalement ici que la moyenne des écarts-types doit être multipliée par $\sqrt{\frac{10}{7}}$pour avoir un estimateur sans biais de l'écart-type...

Intervalles de confiance

Pour des fréquences

Une situation fictive

Sur les 100 000 électeurs d'une ville, 43 000 s'apprêtent à voter pour le maire sortant, mais celui-ci ne le sait pas. Alors il commande un sondage, et l'institut de sondage constitue un échantillon de 100 habitants sur lesquels 52 disent vouloir voter pour le maire. Fou de joie, celui-ci achète le champagne pendant la campagne, avant les élections, et se fait battre lors des élections. Il accuse l'institut de sondage d'avoir triché. Qu'en penser?

Comme *Python* sait faire des tirages sans remise, on peut constituer une liste ordonnée de *pour* et de *contre*, et y puiser des échantillons au hasard. On peut estimer la proportion d'entre eux qui donne de faux espoirs au maire (au moins 50 *pour* parmi les 100).

```
from random import *
population=['contre' for n in range(57000)]+['pour' for n in
range(43000)]

shuffle(population)

print(len(population))

#On choisit 1000 echantillons de 100 et on compte combien sont
favorables au maire:
p=len([n for n in range(1000) if len([v for v in sample(population,100)
 if v=='pour'])>=50])
```

```
print(p)
print(p/1000)
```

L'échantillon choisi par l'institut de sondage, s'il a réellement été choisi au hasard, était favorable au maire avec une probabilité d'environ 0,1. Cette probabilité n'est pas si ridicule que ça, et l'institut de sondage aurait pu répondre au maire "c'est la faute à pas de chance": Il est tombé sur les 10 % d'échantillons favorables... Un épisode analogue s'est déroulé lors des élections présidentielles de 1995, le ministre du budget de l'époque ayant un peu trop vite cru aux sondages !

Plus sérieux (et plus prudent, pour éviter la vindicte de l'ancien maire, désormais dans l'opposition, et qui a maintenant le temps de mener une croisade contre les instituts de sondage) eût été la publication par l'institut de sondage, d'un intervalle de confiance, par exemple à 95% (c'est-à-dire un intervalle qui contient en moyenne 95% des échantillons). Expérimentalement, on peut s'inventer un intervalle et compter la fréquence des échantillons de 100 personnes qui sont dedans. Ce sera un estimateur de la probabilité que l'échantillon soit représentatif de l'ensemble de la population:

```
from random import *

h=0.1

p=0
for n in range(1000):
    pourcentage=len([v for v in sample(population,100) if
v=='pour'])/100
    if pourcentage>0.43-h and pourcentage<0.43+h:
        p+=1

print(p/1000)
```

On voit que l'intervalle *[0,33 ; 0,53]* obtenu avec *h=0,1* est un intervalle à 95 %. En modifiant la valeur de *h* on constate que si *h* diminue (l'intervalle rétrécit), on perd de la confiance (la probabilité qu'il soit bon diminue aussi). On trouve par tâtonnements la valeur de *h* pour laquelle la confiance de l'intervalle vaut 95 %, puis par changement de la taille de l'échantillon, on peut conjecturer le lien entre *h* et la taille de l'échantillon.

Pour des moyennes

Là encore, on peut facilement tester des intervalles de confiance, pour des moyennes de variables aléatoires normales, par exemple d'espérance 16, d'écart-type 2 et indépendantes entre elles:

```
from random import *

h=0.1

p=0
for n in range(1000):
    echantillon=[]
    for k in range(100):
```

```
        echantillon.append(gauss(16,2))

    m=moyenne(echantillon)
    if m>16-h and m<16+h:
        p+=1

print(p/1000)
```

On découvre que l'intervalle de confiance *[15,9 ; 16,1]* donné ci-dessus (pour *h=0,1*) est à environ 40% de confiance. En modifiant la valeur de *h*, on retrouve expérimentalement que pour celle-ci égale à environ $\frac{2\sigma}{\sqrt{100}}$, l'intervalle est à 95 % de confiance.

Test d'équirépartition

En lançant un dé 100 fois, on constate que le 6 est sorti un peu souvent par rapport aux autres nombres:

Résultat du tirage	1	2	3	4	5	6
Effectifs	15	16	17	16	16	20

On se demande si le dé est équilibré. Pour cela, on se choisit comme critère de test la somme des carrés des écarts aux fréquences théoriques: $d2 = \sum_{k=1}^{6} \left(f_k - \frac{1}{6} \right)^2$:

```
d2=(0.15-1/6)**2+(0.16-1/6)**2+(0.17-1/6)**2+(0.16-1/6)**2+(0.16-1/6)**2+(0.2-1/6)**2
```

Soit mais encore, que faire avec *d2*: Est-ce qu'on doit dire que 0,0015 est anormalement élevé et que le dé est truqué, ou que 0,0015 est suffisamment petit pour attribuer ces résultats à la fluctuation d'échantillonnage? Pour le savoir, on va simuler 10000 lancers de dés et calculer l'équivalent de *d2* pour chacun d'entre eux, puis faire une étude statistique sur le *d2* observé. La réponse statistique à la question *Qu'est-ce qui est normal?* est en général fournie par les déciles: On dira que le dé est vraisemblablement truqué si le *d2* observé est supérieur au neuvième décile de la série. Pour calculer ce décile, on va devoir trier les données.

```
from random import *
khi2=[]
for n in range(10000):
    effectifs=[0]*6
    for k in range(100):
        effectifs[randint(0,5)]+=1
    dsquare=0
    for e in effectifs:
        dsquare+=(e/100-1/6)**2
    khi2.append(dsquare)

khi2.sort()
decile9=khi2[int(0.9*len(khi2))]

print(decile9)
print(d2)
```

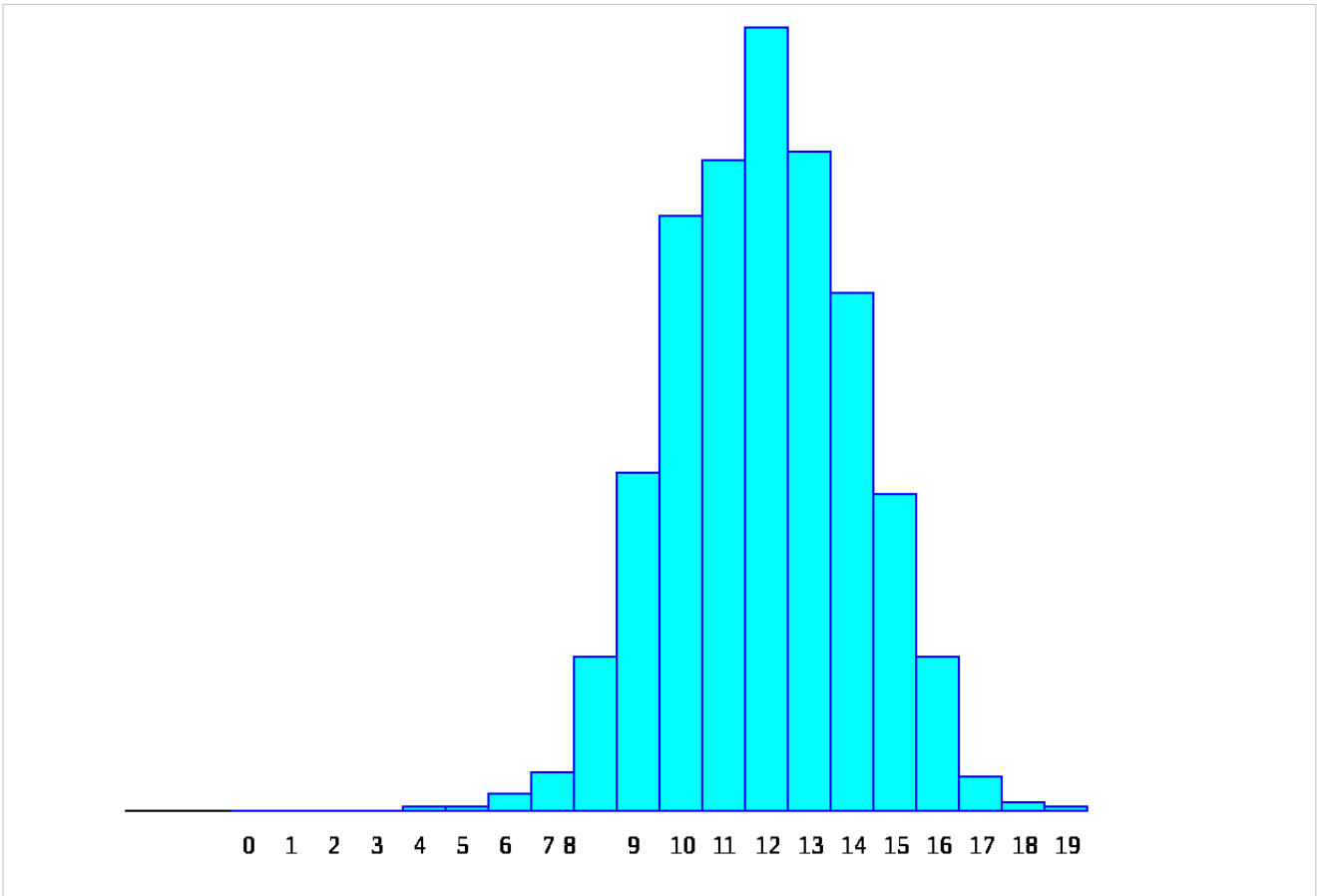
d2 est environ 10 fois plus petit que le neuvième décile de la série, donc on se trompe de plus de 10 % en considérant que le dé est truqué: Il est parfaitement normal pour autant qu'on sache.

Problème des rencontres d'Euler

Euler affirme que, si on joue 19 fois au jeu de treize, le joueur A gagnera *probablement 12 fois*. On peut détailler cette affirmation, en jouant 19 fois au jeu pour chaque échantillon, et en répétant l'expérience 1000 fois (donc 1000 échantillons de 19 parties). Voici le script:

```
valeurs={1,7,8,9,10,'Valet','Dame','Roi'}
couleurs={'carreau','coeur','pique','trefle'}
jeu1=[str(v)+' '+c for v in valeurs for c in couleurs]
jeu2=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *

pg=[0]*20
for n in range(1000):
    rencontres=0
    for p in range(19):
        shuffle(jeu2)
        for i in range(32):
            if jeu1[i]==jeu2[i]:
                rencontres+=1
                break
    pg[rencontres]+=1

print(pg)
```

Le tableau affiché par ce script est celui des parties gagnées par A parmi les 19; il s'agit d'un tableau d'effectifs (le total des nombres entiers affichés est d'ailleurs 1000). Voici un exemple de diagramme en bâtons de ce tableau, traîtreusement déguisé en histogramme:



On voit que le **mode** de cette série est 12, c'est peut-être ce que voulait dire Euler. Mais peut-être aussi voulait-il dire que l'espérance de la série (en fait, sa moyenne puisqu'on fait de la statistique) est proche de 12. C'est bien le cas, cette moyenne étant proche de 11,9 (avec un écart-type proche de 2). En fait le phénomène aléatoire simulé ci-dessus est assez bien modélisé par une variable aléatoire binomiale de paramètres 19 et $\frac{12}{19}$, dont l'espérance est 12 et l'écart-type $\sqrt{19 \times \frac{12}{19} \times 719} \simeq 2,103$.

Suites en Python

Une suite est une fonction de $\mathbb{N}$ dans $\mathbb{R}$. Tout ce qui est dit dans le chapitre suivant peut donc être appliqué aux suites. On va donc essentiellement parler de suites récurrentes ici. Numériquement, les suites servent surtout à faire des calculs, selon la méthode suivante:

1. Pour calculer un nombre ℓ , on *invente* une suite u_n telle que $\lim_{n\to\infty} u_n = \ell$;
2. On estime que par exemple 1000 est suffisamment proche de ∞ pour que $u_{1000} \simeq \ell$.
3. On calcule alors u_{1000} (en général par récurrence).
4. On considère donc qu'on a fini de calculer ℓ .

C'est ainsi par exemple, qu'on calcule

1. des racines carrées par la méthode de Heron;
2. Le nombre pi par les différentes méthodes connues (suites qui convergent vers π)
3. Les intégrales se calculent numériquement par des méthodes analogues;
4. La constante d'Euler est définie comme limite d'une suite.
5. Les nombres de Bernoulli sont aussi définis comme une suite, bien que pour une fois, ce ne soit pas à sa limite qu'on s'intéresse.
6. Même le nombre d'Or peut s'obtenir comme limite d'une suite basée sur les nombres de Fibonacci...

Suites récurrentes

Les suites récurrentes sont celles pour lesquelles u_{n+1} dépend de u_n . Mais pour la suite de Fibonacci, la dépendance va plus loin, non seulement le dernier terme intervient, mais également le pénultième.

Suite logistique

La suite est définie par $u_{n+1} = 4u_n\left(1-u_n\right)$. Son comportement dépend grandement de la valeur de u_0 mais elle est souvent chaotique. Pour calculer ses 20 premiers termes, on peut écrire une simple boucle:

```
u=0.1
for n in range(20):
    u=4*u*(1-u)
    print(u)
```

En effet, une suite récurrente se représente, si *n* est le temps, par l'affectation d'une variable avec l'image de son ancienne valeur par une fonction (ici, $x \mapsto 4x\left(1-x\right)$). Si le premier terme de la suite est une fraction, il en est de même pour tous les termes suivants:

```
from fractions import *

u=Fraction(1,10)
for n in range(10):
    u=4*u*(1-u)
    print(u)
```

Suites arithmétiques et géométriques

Une suite est arithmétique si on passe de chaque terme au suivant en additionnant le même nombre, appelé *raison* de la suite. L'objet *range* de *Python* est donc une suite arithmétique d'entiers.

Par exemple, si on place 2000 € avec des intérêts simples s'élevant à 3 % de 2000, soit 60 € par an, l'évolution du capital pendant 20 ans s'obtient avec ce script:

```
C=2000
I=2000*3/100
for n in range(1,20):
    C+=I
    print(C)
```

Une suite est géométrique si on passe de chaque terme au suivant en multipliant par un même nombre appelé également *raison*. Par exemple, si on place 2000 € avec des intérêts composés au taux de 2 %, l'évolution du capital année après année est donnée par ce script:

```
C=2000
for n in range(1,20):
    C*=1.02
    print(round(C,2))
```

Le module *cmath* permet aussi d'étudier les suites géométriques complexes. On constate alors que si le module de la raison est plus petit que 1, la suite tend vers 0, et si le module de la raison est supérieur à 1, la suite tend vers l'infini. C'est bien entendu lorsque le module est égal à 1 qu'il se passe les choses les plus intéressantes...

Suite de Collatz

Algorithmiquement, la suite de Collatz est intéressante parce que son calcul est basé sur un test de parité, et qu'elle utilise une boucle à condition de sortie:

```
u=65
while(u>1):
    if u%2:
        u=3*u+1
    else:
        u//=2
    print(u)
```

La division par 2 est une division euclidienne, en effet on souhaite que *u* reste entier (et non flottant) au cours de la boucle.

Suite de Fibonacci

Calcul de la suite

La récurrence de la suite de Fibonacci est double, avec $u_{n+1}=u_n+u_{n-1}$. Son calcul pose donc un problème algorithmique, puisqu'il faut trois variables (les deux termes à calculer et une variable *tampon* pour stocker temporairement l'un des deux termes, afin qu'il ne soit pas écrasé par la somme). Ce problème n'existe pas en *Python* qui permet les affectations simultanées.

```
a=1
b=1
for n in range(20):
    a,b=b,a+b
    print(a)
```

Nombre d'Or

Un problème numériquement intéressant (et c'était la motivation initiale de Fibonacci) est d'étudier le comportement du rapport entre deux termes successifs de la suite de Fibonacci:

```
a=1
b=1
for n in range(20):
    a,b=b,a+b
    print(b/a)

print((1+5**0.5)/2)
```

On constate la convergence vers le nombre d'Or.

Suites définies par des sommes

Un exemple

La suite définie par $u_n=\frac{1}{2}+\frac{1}{6}+\frac{1}{12}+\frac{1}{20}+...=\frac{1}{1\times 2}+\frac{1}{2\times 3}+\frac{1}{3\times 4}+\frac{1}{4\times 5}+...+\frac{1}{n(n+1)}=\sum_{k=1}^{n}\frac{1}{k(k+1)}$ tend vers 1, il est relativement aisé de le démontrer, et presque aussi facile de le vérifier avec *Python*:

```
somme=0
for n in range(1,50):
    somme+=1/(n*(n+1))
    print(somme)
```

Un autre exemple

La suite $u_n = \frac{n}{n^2+1} + \frac{n}{n^2+2} + \frac{n}{n^2+3} + \frac{n}{n^2+4} + ... + \frac{n}{n^2+n} = \sum_{k=1}^{n} \frac{n}{n^2+k}$ converge aussi, bien que ce ne soit pas évident en voyant son expression algébrique.

```
for n in range(1,20):
    for k in range(1,n):
        somme+=n/(n**2+k)
    print(somme)
```

La constante d'Euler

On peut la calculer (et vérifier la lenteur de la convergence) avec

```
from math import *
somme=0
for n in range(1,50):
    somme+=1/n
    print(somme-log(n))
```

Calcul de racines carrées

Méthode de Heron

En constatant que

1. Si $u \simeq \sqrt{5}$ alors $\frac{5}{u} \simeq \sqrt{5}$ aussi;
2. Si $u < \sqrt{5}$ alors $\frac{5}{u} > \sqrt{5}$ et *vice-versa*;
3. Par conséquent, on s'attend à ce que la moyenne entre u et $\frac{5}{u}$ soit une valeur approchée encore meilleure de $\sqrt{5}$,

on a l'ébauche d'une suite récurrente qui tend vers $\sqrt{5}$:

Application

```
u=1
while(abs(u**2-5)>1e-14):
    u=(u+5/u)/2
    print(u)

print(5**0.5)
```

Fonctions en Python

On va traiter un exemple de fonction, issu du Bac STG CGRH Métropole-Réunion de Septembre 2007:

Définition des fonctions

Énoncé

Extrait du sujet de l'exercice 3:

Une entreprise produit des appareils électroménagers. Le coût horaire de production de x appareils est donné en euros par :

$C(x) = x^2 + 50x + 100$ pour $5 \leqslant x \leqslant 40$

Et plus bas, dans le même exercice,

le coût moyen est défini par

Le coût moyen de production d'un objet est égal à $f(x) = \frac{C(x)}{x}$ pour x appartenant à [5 ; 40].

Fonctions et procédures sous Python

Une fonction est un petit bout de programme *Python* qui possède un nom (typiquement *f*), et qui renvoie une valeur (l'image de *x*, son unique antécédent, par *f*). Une procédure est une fonction qui ne renvoie rien.

Les fonctions du sujet de Bac ci-dessus peuvent être définies par *def* en *Python*:

```
def C(x):
    return x**2+50*x+100.0


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

Si on essaye de calculer l'image de 0 par *f*, on remarque qu'en lieu et place du calcul, *Python* affiche un message d'erreur: La fonction *f* n'est pas definie en 0. On peut aussi la définir directement par

```
def f(x):
    return x+50+100.0/x
```

On peut aussi définir des fonctions de plusieurs variables (comme le pgcd) ou des fonctions définies par des boucles ou des tests, comme la valeur absolue:

```
def abs(x):
    if x>0:
        return x
    else:
        return -x
```

Tableau de valeurs

Suite de l'énoncé du Bac 2007

Par la suite, on demande de reproduire et compléter le tableau suivant, arrondi au centième d'euro:

x	5	10	20	30	40
f(x)					

Solution en Python

Pour remplir le tableau, on peut

1. Faire 5 fois un *print(f(x))* pour les 5 valeurs de *x* de l'énoncé;
2. Faire une boucle sur les 5 valeurs de *x* de l'énoncé;
3. Faire une boucle sur suffisamment de valeurs de *x* pour couvrir les 5 valeurs de l'énoncé;
4. Transformer *f* en une fonction qui peut s'appliquer à une liste de valeurs et non à une valeur unique.

La méthode 3 peut se faire avec un itérateur de *Python*:

```
for x in range(5,40,5):
    print("l\'image de "+str(x)+" par f est "+str(f(x)))
```

Trop de calculs tuent le calcul! Les images de 25 et 35 par exemple encombrent le tableau. Mais *Python* permet d'utiliser des itérateurs beaucoup plus souples, et adaptés au problème présent:

```
for x in [5,10,20,30,40]:
    print("l\'image de "+str(x)+" par f est "+str(f(x)))
```

La méthode 4 peut s'implémenter en *Python* avec:

```
print([f(x) for x in [5,10,20,30,40]])
```

C'est aussi simple que ça!

Arrondis

L'énoncé demandait d'arrondir au centième d'euro près

```
for x in [5,10,20,30,40]:
    print("l\'image de "+str(x)+" par f est "+str(round(f(x),2)))
```

Représentation graphique

Pour représenter graphiquement une fonction comme *f* ci-dessus, on trace un polygone ayant suffisamment de sommets, et ceux-ci suffisamment proches les uns des autres, pour que la courbe ait l'air courbe. Plusieurs outils permettent de faire du graphisme avec *Python* mais le plus simple semble être le module *tortue*.

Avec Turtle

Courbe

Le plus simple est de faire

```
from turtle import *
setpos(5,f(5))
for x in range(5,40):
    setpos(x,f(x))
```

Difficile de rêver plus simple mais on peut voir trois problèmes:

1. Il manque les axes
2. Le trait qui relie l'origine au point de coordonnées (5;f(5)) est en trop
3. La tortue gêne la visibilité de la figure

Courbe améliorée

Pour résoudre le deuxième problème, il suffit de lever le crayon avec *penup()* avant de commencer le tracé (et de le redescendre pour le tracé lui-même). Pour résoudre le troisième problème, il suffit de rendre la tortue invisible avec *hideturtle()*. Pour les axes, on peut les tracer avec la tortue:

```
from turtle import *
penup()
setpos(5,f(5))
pendown()
for x in range(5,40):
    goto(x,f(x))

penup()
setpos(0,0)
pendown()
for x in range(0,50,10):
    left(90)
    forward(2)
    backward(4)
    forward(2)
    right(90)
    forward(10)

stamp()
backward(50)
left(90)
for y in range(0,100,10):
    left(90)
    forward(2)
    backward(4)
    forward(2)
    right(90)
    forward(10)
```

```
stamp()
```

Le graphique est correct mais un peu petit et mal cadré.

Avec TkInter

TkInter permet de créer un affichage dans un *canevas*, lui-même membre d'une fenêtre. Il y a donc plusieurs lignes de *Python* à écrire avant même de commencer à dessiner. Et l'axe des ordonnées est dirigé vers le bas, ce qui oblige à une transformation des ordonnées. En contrepartie, on a un outil de dessin tout-à-fait correct, et même relativement classique. Et, bien que ce ne soit pas utile dans le cas présent, on peut avoir des boutons, curseurs etc.

On doit donc commencer par

1. importer tkInter (s'installer dans l'atelier du peintre)
2. créer une fenêtre (monter le chevalet, un cadre...)
3. y placer un *canevas* (tendre la toile sur le chevalet)
4. afficher le canevas avec *pack* (enlever la couverture qui cache le chef-d'œuvre)
5. Enfin, dessiner (axes et fonction)

Ce qui peut donner ceci:

```
from tkinter import *
fenetre=Tk()
graphique=Canvas(fenetre,width=640,height=480)
graphique.pack()

#axe des abscisses
graphique.create_line(20,460,520,460)
for n in range(0,50,10):
    graphique.create_line(20+10*n,460,20+10*n,455)
for n in range(0,50,5):
    graphique.create_line(20+10*n,460,20+10*n,457)
for n in range(0,50):
    graphique.create_line(20+10*n,460,20+10*n,459)

#axe des y
graphique.create_line(20,460,20,60)
for n in range(0,100,10):
    graphique.create_line(20,460-4*n,25,460-4*n)
for n in range(0,100,5):
    graphique.create_line(20,460-4*n,24,460-4*n)
for n in range(0,100):
    graphique.create_line(20,460-4*n,22,460-4*n)

#courbe
for x in range(5,40):
    graphique.create_line(20+10*x-10,460-4*f(x-1),20+10*x,460-4*f(x))
```

Analyse numérique en Python

Fonction

Dans ce chapitre, on va effectuer des calculs sur la fonction $x \mapsto x^2 - 5$; on va appeler cette fonction *f*. Pour se faciliter la suite, on va créer cette fonction:

```
def f(x):
    return x**2-5
```

Résolution numérique d'une équation

Pour résoudre l'équation *f(x)=0*, on cherche un intervalle sur lequel on est certain que *f* s'annule. C'est le cas pour *[1;3]* parce que f(1) est négatif et f(3) est positif. La méthode de dichotomie vise à resserrer un tel intervalle. On constate ci-dessous que la fonction *f* est traitée comme une entrée de l'algorithme au même titre que les bornes *a* et *b* de l'intervalle:

```
def zero(f,a,b):
    if f(a)*f(b)>0:
        print('pas de solution entre '+str(a)+' et '+str(b)+'!')
        return 0
    while(abs(a-b)>1e-14):
        m=(a+b)/2.
        if f(m)*f(a)>0:
            a=m
        else:
            b=m
    print('la solution de f(x)=0 est '+str(m))
    return m



print(zero(f,1,3))
```

La résolution de l'équation $x^2 = 0$ n'est pas terminée, puisque le script ci-dessus n'a donné qu'une seule des deux solutions de cette équation. Par ailleurs, la solution trouvée n'est affichée qu'à 10^{-14} près.

Calcul numérique de nombre dérivé

Pour calculer le nombre dérivé de *f* en 5, on va utiliser l'approximation $f'(x) \simeq \frac{f(x+h)-f(x)}{2h}$:

```
def NDer(f,a):
    h=1e-10
    return (f(a+h)-f(a-h))/(2*h)



print(NDer(f,5))
```

Calcul numérique d'une intégrale

La méthode des rectangles dit que $\int_a^b f(t)\,dt \simeq \sum_{k=0}^{N} h \times f(a+kh)$ où $h = \frac{b-a}{N}$ et N est suffisamment grand pour que h soit petit (ci-dessous N=1 000 000):

```
def Int(f,a,b):
    h=(b-a)/1000000.0
    somme=0
    for n in range(1000000):
        somme+=h*f(a+n*h)
    return(somme)



print(Int(f,0,2))
```

Points en Python

L'objet *Point* est une bonne manière d'aborder la programmation objet. En géométrie repérée, un point est constitué de deux nombres, son abscisse et son ordonnée.

Voici l'énoncé de l'exercice:

Dans un repère orthonormé, on considère $A(-1;3)$, $B(5;1)$et $C(1;5)$. Calculer les distances AB, AC et BC et en déduire la nature du triangle ABC. Puis en déduire les coordonnées du centre de son cercle circonscrit.

Création de l'objet

Le point de coordonnées *(x,y)* est, en *Python*, une classe:

```
class Point:
    def __init__(self,x,y):
        self.x=x
        self.y=y
```

Lorsqu'on crée un point, ses coordonnées sont stockées à l'intérieur de l'objet. On note *p.x* et *p.y* les coordonnées de *p*.

Affichage

La méthode peut ressembler à ceci:

```
    def affichage(self):
        return '('+str(self.x)+';'+str(self.y)+')'
```

mais on peut envisager d'y rajouter des instructions avec *TkInter* pour réellement dessiner le point sur la figure. Voir à ce sujet le chapitre sur les fonctions.

Avec deux points

Le plus simple quand on a deux points, c'est leur milieu, parce que c'est aussi un point (donc un objet de même nature).

Milieu

Les coordonnées du milieu d'un segment sont les moyennes de celles des extrémités:

```
    def milieu(self,p):
        return Point((self.x+p.x)/2,(self.y+p.y)/2)
```

En se rappelant que l'équivalent en Java de *self* est *this*, on remarque une certaine ressemblance avec les codes sources de logiciels de géométrie dynamique:

Tout d'abord, CaRMetal:

```
                    setXY((P1.getX() + P2.getX()) / 2, (P1.getY() +
P2.getY()) / 2);
```

Ensuite, GeoGebra:

```
            M.setCoords(
                (P.inhomX + Q.inhomX) / 2.0d,
                (P.inhomY + Q.inhomY) / 2.0d,
                1.0);
```

Remarque: Ces codes sources sont sous license GPL ce qui autorise à les citer, au nom de la liberté numéro 1 (celle d'étudier le logiciel) de Richard Stallman.

Vecteur

Le vecteur d'origine *A* et d'extrémité *B*, noté $\overrightarrow{AB}$, est un vecteur! On le définira donc au chapitre suivant mais il peut servir ici:

```
    def vecteur(self,p):
        return Vecteur(p.x-self.x,p.y-self.y)
```

Distance

Pour simplifier l'écriture de la distance *AB* on va encore utiliser les vecteurs, la distance *AB* étant égale à $||\overrightarrow{AB}||$:

```
    def distance(self,p):
        return self.vecteur(p).norme()
```

Application au problème

Voici l'objet *Point* en entier:

```
from math import *

class Point:
    def __init__(self,x,y):
        self.x=x
```

```
        self.y=y

    def affichage(self):
        return '('+str(self.x)+';'+str(self.y)+')'

    def milieu(self,p):
        return Point((self.x+p.x)/2,(self.y+p.y)/2)

    def vecteur(self,p):
        return Vecteur(p.x-self.x,p.y-self.y)

    def distance(self,p):
        return self.vecteur(p).norme()
```

Nature de ABC

Pour savoir si ABC est isocèle, on peut calculer les longueurs de ses trois côtés:

```
a=Point(-1,3)
b=Point(5,1)
c=Point(1,5)

print(a.distance(b))
print(a.distance(c))
print(b.distance(c))
```

Visiblement, ABC n'est pas isocèle. Mais

```
print(a.distance(b)**2)
print(a.distance(c)**2+b.distance(c)**2)
```

La réciproque du théorème de Pythagore nous apprend que ABC est rectangle en C, donc d'hypoténuse *AB*.

Centre du cercle

Donc le cercle circonscrit a pour diamètre *[AB]*, donc pour centre le milieu de *[AB]*:

```
m=a.milieu(b)

print(m.affichage())
```

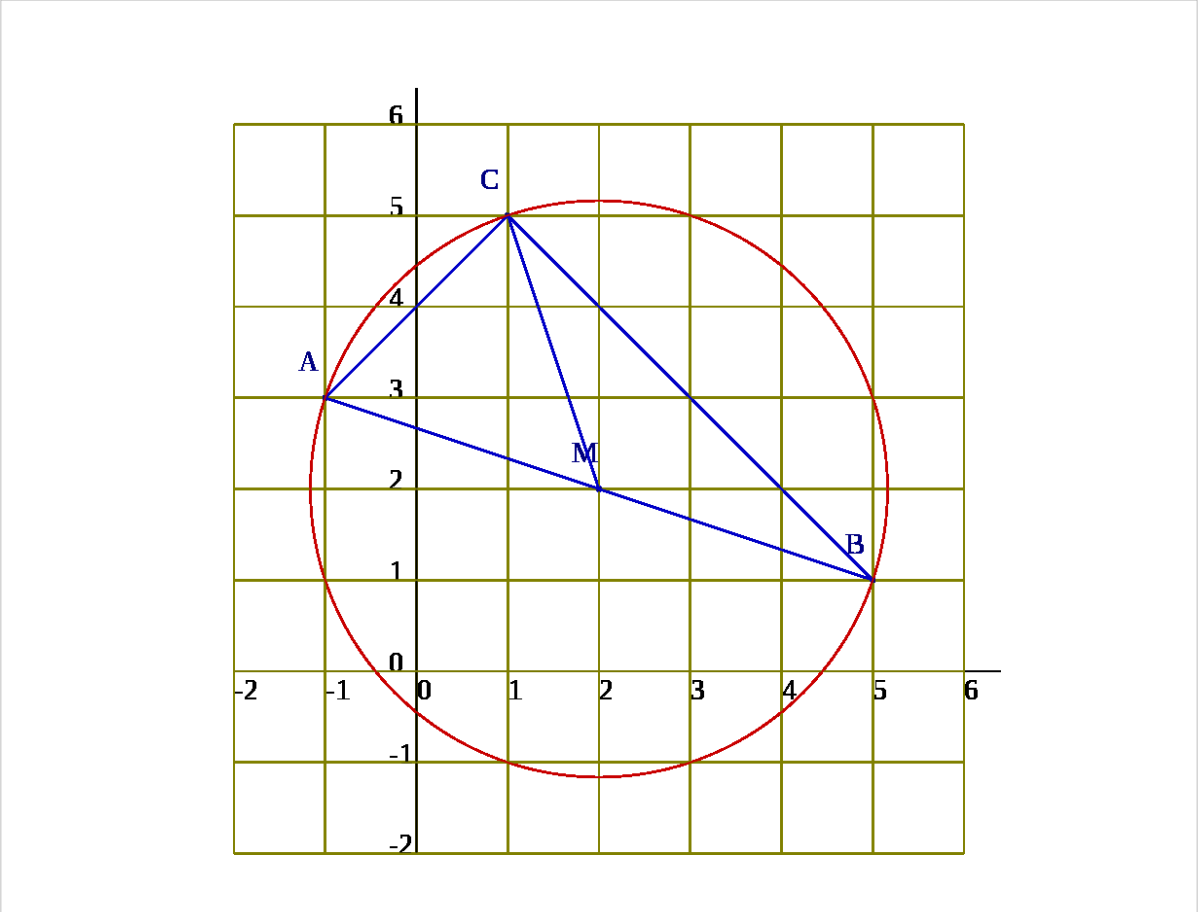
Rayon du cercle

On peut donc diviser par 2 la distance *AB* mais aussi vérifier que *M* est équidistant de *A*, *B* et *C*:

```
print(m.distance(a))
print(m.distance(b))
print(m.distance(c))
```

Figure

On pourrait utiliser *TkInter* pour dessiner le tout (y compris le cercle) mais la figure ci-dessous a été faite avec Ruby, ce langage permettant assez facilement de fabriquer un fichier au format svg:



Vecteurs en Python

Géométriquement, un vecteur peut être défini à partir de deux points (son origine et son extrémité) mais aussi par ses coordonnées. C'est le choix qui sera fait ici.

Définition

Encore une fois, on va être classe sur ce coup-là:

```
class Vecteur:
    def __init__(self,x,y):
```

Lors de son instanciation, un vecteur n'aura donc que deux propriétés: Ses coordonnées.

Coordonnées

Abscisse

```
        self.x=x
```

L'abscisse de *u* s'obtiendra par *u.x*

Ordonnée

```
        self.y=y
```

L'ordonnée de *u* s'obtiendra par *u.y*

La classe *Vecteur* se résume donc pour l'instant à ceci:

```
class Vecteur:
    def __init__(self,x,y):
        self.x=x
        self.y=y
```

Affichage

Pour afficher un vecteur, on fait comme avec les points (on colle ses coordonnées entre parenthèses, séparées par un point-virgule, après les avoir converties en chaînes de caractères avec *str*):

```
    def affichage(self):
        return '('+str(self.x)+';'+str(self.y)+')'
```

Norme

Pour calculer la norme d'un vecteur, on utilise le théorème de Pythagore sous la forme de la fonction *hypot*. Celle-ci doit être importée du module *math*:

```
from math import *
```

La norme du vecteur se calcule donc par

```
    def norme(self):
        return hypot(self.x,self.y)
```

Opérations

Python offre un grand confort, en permettant d'utiliser les signes d'opérations pour les vecteurs. Ainsi, en voyant un signe *moins* devant un vecteur, *Python* sait qu'il ne doit pas calculer l'opposé d'un nombre puisque c'est un vecteur, et non un nombre, qui suit ce signe *moins*. Alors on peut définir (c'est laissé en exercice) une méthode *__neg__* pour les vecteurs, et c'est elle qui sera utilisée quand *Python* verra le signe *moins*. À moins bien entendu que ce soit une soustraction, auquel cas la méthode *__sub__* sera utilisée (cette méthode aussi sera laissée en exercice).

Addition

Pour additionner deux vecteurs, on redéfinit la méthode *__add__*:

```
    def __add__(self,v):
        return Vecteur(self.x+v.x,self.y+v.y)
```

Alors pour avoir la somme de deux vecteurs *u* et *v*, il suffit d'entrer *u+v*. C'est exactement comme ça que fonctionnent les modules *fractions* et *cmath*.

Multiplications

Par un réel

En multipliant un vecteur par un réel, on obtient un vecteur. Pour ne pas interférer avec la méthode suivante, on va mettre le réel en deuxième et utiliser la méthode *multiplication à l'envers* qui est notée *__rmul__* (comme *reverse multiplication*):

```
    def __rmul__(self,r):
        return Vecteur(self.x*r,self.y*r)
```

Pour calculer le triple du vecteur *u*, on entre *3*u* et, pour l'afficher, *(3*u).affichage()*.

Par un autre vecteur

En multipliant un vecteur par un vecteur, on obtient un nombre, ou scalaire. Aussi cette multiplication est-elle appelée produit scalaire des deux vecteurs (en fait il y a bien une multiplication vectorielle qui donne un vecteur, mais celle-ci est définie pour les vecteurs de l'espace, et ne sera donc pas abordée ici). On peut donc implémenter le produit scalaire de deux vecteurs *self* et *v* par

```
    def __mul__(self,v):
        return self.x*v.x+self.y*v.y
```

Pour calculer le produit scalaire de *u* par *v*, on entre juste *u*v*.

Tests

Vecteurs colinéaires

Pour savoir si deux vecteurs sont colinéaires, on compare deux produits en croix:

```
    def colin(self,v):
        return self.x*v.y==self.y*v.x
```

Ce test retourne un booléen, en entrant *u.colin(v)*.

Vecteurs orthogonaux

Pour savoir si deux vecteurs sont orthogonaux, on compare leur produit scalaire à 0:

```
    def ortho(self,v):
        return self*v==0
```

Exemple

Pour l'exercice du chapitre précédent, on peut calculer un produit scalaire pour vérifier que ABC est rectangle:

```
from math import *

class Vecteur:
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def affichage(self):
        return '('+str(self.x)+';'+str(self.y)+')'

    def norme(self):
        return hypot(self.x,self.y)

    def __add__(self,v):
        return Vecteur(self.x+v.x,self.y+v.y)

    def __rmul__(self,r):
        return Vecteur(self.x*r,self.y*r)

    def __mul__(self,v):
        return self.x*v.x+self.y*v.y

    def colin(self,v):
        return self.x*v.y==self.y*v.x

    def ortho(self,v):
        return self*v==0
```

```
a=Point(-1,3.0)
b=Point(5,1)
c=Point(1,5)

u=c.vecteur(a)
v=c.vecteur(b)

print(u*v)
print(u.ortho(v))
```

Droites en Python

De façon analogue à la définition par coordonnées des points et vecteurs dans les deux chapitres précédents, on peut définir l'objet *droite* par une de ses équations. Mais ici, on va définir une droite à partir de deux points. On va donc avoir besoin des classes Point et Vecteur définies dans les deux chapitres précédents. On aura donc aussi besoin d'importer le module *math* (ou au moins, *hypot*).

Définition

On définit donc une classe *Droite* où l'initialisation place deux points:

```
class Droite:
    def __init__(self,a,b):
        self.a=a
        self.b=b
```

Le premier point de la droite *d* s'obtient par *d.a* et le second point par *d.b*.

Vecteurs

Vecteur directeur

Le vecteur $\overrightarrow{AB}$ sera choisi comme vecteur directeur de la droite (AB):

```
    def directeur(self):
        return self.a.vecteur(self.b)
```

On peut s'en servir pour

1. Avoir une représentation paramétrique de la droite;
2. Faire un test pour savoir si un point est, ou non, sur la droite (par des vecteurs colinéaires).

Ces deux ajouts sont laissés en exercice, le deuxième étant d'ailleurs hors sujet ici puisque c'est une propriété de l'objet *Point*.

Vecteur normal

Le vecteur normal est choisi de telle façon que son produit scalaire avec le vecteur directeur ci-dessus soit nul:

```
    def normal(self):
        return Vecteur(-self.directeur().y,self.directeur().x)
```

Ce vecteur directeur peut servir à obtenir une équation cartésienne de la droite *(AB)*:

Équations

Équation cartésienne

```
    def cartesienne(self):
        return
'('+str(self.normal().x)+')x+('+str(self.normal().y)+')y='+str(self.normal().x*self.a.x+self.normal().y*self.a.y)
```

On écrit *print(d.cartesienne())* pour afficher l'équation cartésienne de *d*.

Équation réduite

L'équation réduite $y=mx+p$ est acquise dès que le coefficient directeur m et l'ordonnée à l'origine $p=y-mx$ sont acquis.

Coefficient directeur

Le coefficient directeur est le quotient de l'ordonnée du vecteur directeur par son abscisse:

```
    def cd(self):
        return self.directeur().y/self.directeur().x
```

Ordonnée à l'origine

```
    def oalo(self):
        return self.a.y-self.cd()*self.a.x
```

Équation

On l'obtient à partir des deux nombres précédents:

```
    def reduite(self):
        return 'y='+str(self.cd())+'x+('+str(self.oalo())+')'
```

On l'affiche par *print(d.reduite())*

Point d'intersection

Pour calculer les coordonnées du point d'intersection de deux droites, on doit résoudre un système. Voir à ce sujet le chapitre qui lui est consacré.

Relations entre droites

Parallélisme

Pour savoir si deux droites sont parallèles, on peut

1. Regarder si leurs vecteurs directeurs sont colinéaires (mieux);
2. ou comparer leurs coefficients directeurs (moins bien, il faut déjà qu'elles en aient un!)

C'est la première méthode qui sera appliquée ici, bien que la deuxième soit plus rapide:

```
    def parallele(self,d):
        return self.directeur().colin(d.directeur())
```

Orthogonalité

Pour savoir si deux droites sont perpendiculaires, on regarde si leurs vecteurs normaux le sont:

```
    def perpendiculaire(self,d):
        return self.normal().ortho(d.normal())
```

Application à l'exercice précédent

En résumé, l'objet *Droite* se résume à ceci:

```
from math import *


class Droite:
    def __init__(self,a,b):
        self.a=a
        self.b=b


    def directeur(self):
        return self.a.vecteur(self.b)


    def normal(self):
        return Vecteur(-self.directeur().y,self.directeur().x)


    def cartesienne(self):
        return
'('+str(self.normal().x)+')x+('+str(self.normal().y)+')y='+str(self.normal().x*self.a.x+self.normal().y*self.a.y)


    def cd(self):
        return self.directeur().y/self.directeur().x


    def oalo(self):
        return self.a.y-self.cd()*self.a.x


    def reduite(self):
        return 'y='+str(self.cd())+'x+('+str(self.oalo())+')'


    def parallele(self,d):
```

```
        return self.directeur().colin(d.directeur())


    def perpendiculaire(self,d):
        return self.normal().ortho(d.normal())
```

Pour l'exercice des chapitres précédents, on peut vérifier que le triangle *ABC* est rectangle en *C*, en regardant si les droites *(AC)* et *(BC)* sont perpendiculaires:

```
a=Point(-1,3.0)
b=Point(5,1)
c=Point(1,5)



d1=Droite(c,a)
d2=Droite(c,b)

print(d1.perpendiculaire(d2))
```

Une tortue qui accélère la résolution de problèmes

Depuis la version 2.6, *Python* possède un module appelé **turtle** et qui lui permet de faire du graphisme à la LOGO. Outre l'intérêt que peut présenter la consultation de son code source (on y trouve pratiquement tout ce qui est décrit dans les chapitres précédents sur la géométrie), ce module *turtle* permet de simplifier la résolution de certains problèmes et même d'introduire graphiquement certaines notions mathématiques. Le fil conducteur de ce chapitre est que la tortue LOGO peut mémoriser certaines données de position et se comporte comme une mémoire à la fois plus puissante et moins abstraite que les habituelles variables numériques.

Avant d'utiliser la tortue de *Python*, on doit l'importer, en faisant

```
from turtle import *
```

Ensuite, une connaissance du vocabulaire de situation et de déplacement en anglais peut aider; en voici un échantillon:

1. *forward* ou *fd*: Pour avancer (l'unité de distance est le pixel)
2. *backward* ou *bk*: Pour reculer
3. *left* ou *lt*: Pour tourner à gauche (l'unité d'angle est le degré)
4. *right* ou *rt*: Pour tourner à droite
5. *goto* pour téléporter la tortue (donner l'abscisse puis l'ordonnée)
6. *penup()* ou *pu()* pour que les déplacements de la tortue cessent de laisser des traces à l'écran
7. *pendown()* ou *pd()* pour que les déplacements de la tortue recommencent à laisser des traces
8. *position()* renvoie les coordonnées de la tortue
9. *home()* renvoie la tortue au centre de l'écran (sa position initiale)
10. *reset()* fait pareil mais en effaçant l'écran
11. *circle* dessine un cercle (le rayon est en pixels)
12. *stamp()* donne un coup de tampon sur l'écran, en laissant une empreinte de la tortue

L'écran n'apparaît que lors de l'exécution de la première instruction graphique (un *forward* par exemple). Sous Windows, il est déconseillé de laisser traîner la souris sur cet écran graphique. Dans les exemples qui suivent, l'export vectoriel au format eps du module *TkInter* (dont le module *turtle* hérite) a été utilisé pour produire des figures de meilleure qualité que celles qu'on voit sur l'écran de *turtle*.

Nombres relatifs

Les nombres (réels) peuvent être représentés par des graduations sur une droite, et donc par les emplacements de la tortue à l'écran.

Nombres positifs

Addition

Pour représenter l'addition de 21 et 34, on peut tout simplement entrer

```
from turtle import *

forward(21)
forward(34)

print(position())
```

Ce qui oblige à ignorer une information superflue (l'ordonnée de la tortue). La variante suivante permet d'éviter cela:

```
reset()
forward(21)
forward(34)

print(distance(0,0))
```

Soustraction

Pour soustraire 21 à 34, il suffit de faire reculer la tortue au lieu de la faire avancer:

```
reset()
forward(34)
backward(21)

print(position())
```

Si on intervertit l'amplitude des mouvements, on découvre que *Python* choisit d'afficher négativement une position à gauche de l'origine:

```
reset()
forward(21)
backward(34)

print(position())
```

Assez naturellement, on est amené à poser 21-34=-13: Découverte expérimentale des nombres négatifs...

Nombres négatifs

Une fois qu'on a vu des nombres négatifs, on peut chercher comment réaliser des opérations dessus:

Addition

Pour additionner deux nombres négatifs, on peut faire

```
reset()
backward(34)
backward(21)

position()
```

Tout ceci permet assez rapidement d'explorer les différents autres cas de figure (deux cas différents pour la somme de deux nombres de signes différents). Puis la découverte spontanée du fait que les deux instructions suivantes ont le même effet:

```
forward(-34)
backward(34)
```

Ce qui facilite grandement l'exploration de la soustraction de deux nombres relatifs:

Soustraction

Pour calculer 34-(-21), on peut faire

```
reset()
forward(34)
backward(-21)

position()
```

Pour l'apprentissage des opérations sur les nombres négatifs, *turtle* constitue un outil expérimental intéressant à explorer.

Angles orientés

De même, les deux instructions suivantes ont le même effet (rotation de 60° vers la gauche):

```
left(60)
right(-60)
```

mais ce n'est nullement évident pour des lycéens qui n'ont jamais fait ce genre de manipulation, surtout depuis que la notion de rotation a totalement disparu de l'enseignement des mathématiques. Pourtant le module *turtle* permet de visualiser l'addition des angles et d'introduire des notions comme celle d'angles complémentaires ou supplémentaires avec

```
left(30)
left(60)
```

Fonctions

On a vu dans un chapitre précédent comment la module *turtle* permet de représenter graphiquement une fonction.

Statistique

Chute d'une bille sur la planche de Galton

La planche de Galton réalise une marche aléatoire de dimension 1 (le mouvement vertical de la bille n'ayant aucune influence sur le numéro de la case où elle aboutit). On peut donc simuler le mouvement d'une bille avec ce script:

```
from turtle import *
from random import *

for n in range(24):
    if random()<0.5:
        forward(1)
    else:
        backward(1)

print(position())
```

On peut améliorer ce script en utilisant *randrange* qui va de -1 à 1 (donc 2 exclu) par pas de 2, ce qui économise un test:

```
from turtle import *
from random import *

for h in range(24):
    forward(randrange(-1,2,2))

print(position())
```

Statistiques sur 100 billes

Pour effectuer des statistiques sur 100 billes, on a intérêt à accélérer la tortue, avec

```
from turtle import *
from random import *

speed=1000
hideturtle()
penup()
```

Ensuite on crée un tableau d'effectifs pour simuler le bas de la planche de Galton:

```
effectifs=[0 for x in range(-24,25)]
```

Après ça il n'y a plus qu'à remplir le tableau en recommençant 100 fois l'expérience précédente (lancer d'une bille):

```
from turtle import *
from random import *
```

```
speed=0
hideturtle()
penup()

for n in range(100):
    home()
    for h in range(24):
        forward(randrange(-1,2,2))
    effectifs[int(xcor()]+=1
```

Ce script, bien qu'assez court, met du temps à s'exécuter (de l'ordre d'une minute). Pour l'accélérer, on peut ajouter un degré d'abstraction en n'utilisant pas la tortue. En effet, chaque pas est égal à -1 ou 1 au hasard, donc d'après ce qu'on a vu au début de ce chapitre (opérations sur les nombres relatifs), on ne fait qu'additionner 24 nombres égaux à 1 ou -1, ce qui donne ce script:

```
from random import *
effectifs=[0 for x in range(-24,25)]
for n in range(100):
    effectifs[sum(randrange(-1,2,2) for h in range(24))]+=1
```

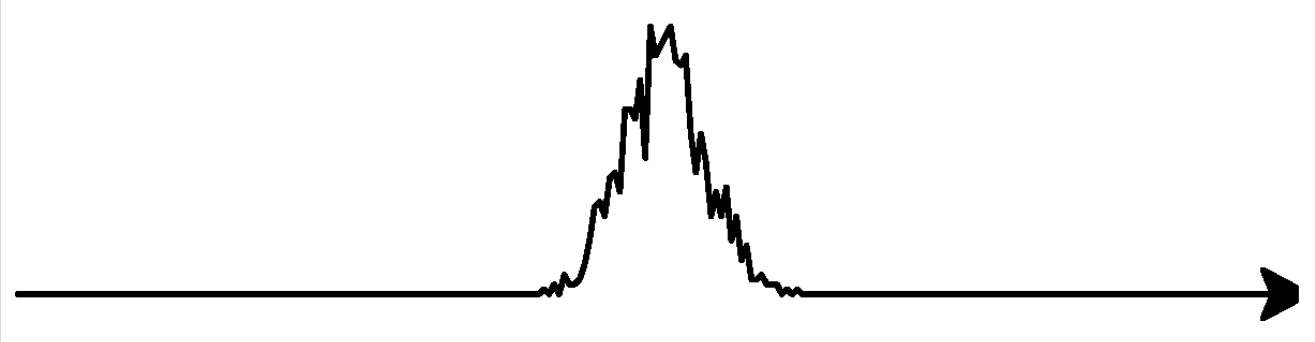
Cette fois-ci, l'effet est presque instantané.

Dessin de l'histogramme

Une fois le tableau d'effectifs rempli, le module *turtle* peut le représenter graphiquement sous forme d'un polygone des effectifs. Comme la méthode précédente est très rapide et que les effectifs des nombres impairs sont nuls, on va plutôt utiliser la variante suivante, avec 256 cases et 1000 essais:

```
from turtle import *
from random import *
effectifs=[0 for x in range(256)]
for n in range(1000):
    effectifs[sum(randrange(2) for h in range(256))]+=1

reset()
for x in range(256):
    goto(x,effectifs[x])
```

On obtient alors un histogramme de ce genre (la tortue est encore visible à droite):



Fractales

La courbe de Von Koch est classiquement définie par la récursivité. Mais elle n'est pas nécessaire si on utilise une expression régulière. En fait, on peut écrire un script Python qui produit un script Python, puis exécuter celui-ci!

Triangle de départ

Comme le script qui va dessiner le triangle fractal sera assez long, on va utiliser des abréviations: *fd* au lieu de *forward*, *lt* au lieu de *left* et *rt* au lieu de *right*. Alors pour dessiner un triangle on peut faire ceci:

```
from turtle import *

fd(100); rt(120); fd(100); rt(120); fd(100); rt(120)
```

Ou mieux, en stockant ce programme en Python dans une variable *programme*:

```
from turtle import *

programme='fd(100); rt(120); fd(100); rt(120); fd(100); rt(120)'
exec(programme)
```

Dans un premier temps, on va abréger encore plus, en notant chaque instruction de ce programme par une seule lettre:

1. *A* (comme *avance*) pour *fd(100)*;
2. *p* (comme *plus*) pour *lt(60)*;
3. *m* (comme *moins*) pour *rt(120)*.

La traduction se fait par une *RegExp*, qui, tel un chien de douane, cherche toutes les occurences d'une lettre, et les remplace par le texte correspondant.

Alors le programme pour créer un programme qui dessine un triangle devient:

```
from turtle import *
from re import *

programme='AmAmAm'
programme=sub('A','fd(100); ',programme)
```

```
programme=sub('m','rt(120); ',programme)
exec(programme)
```

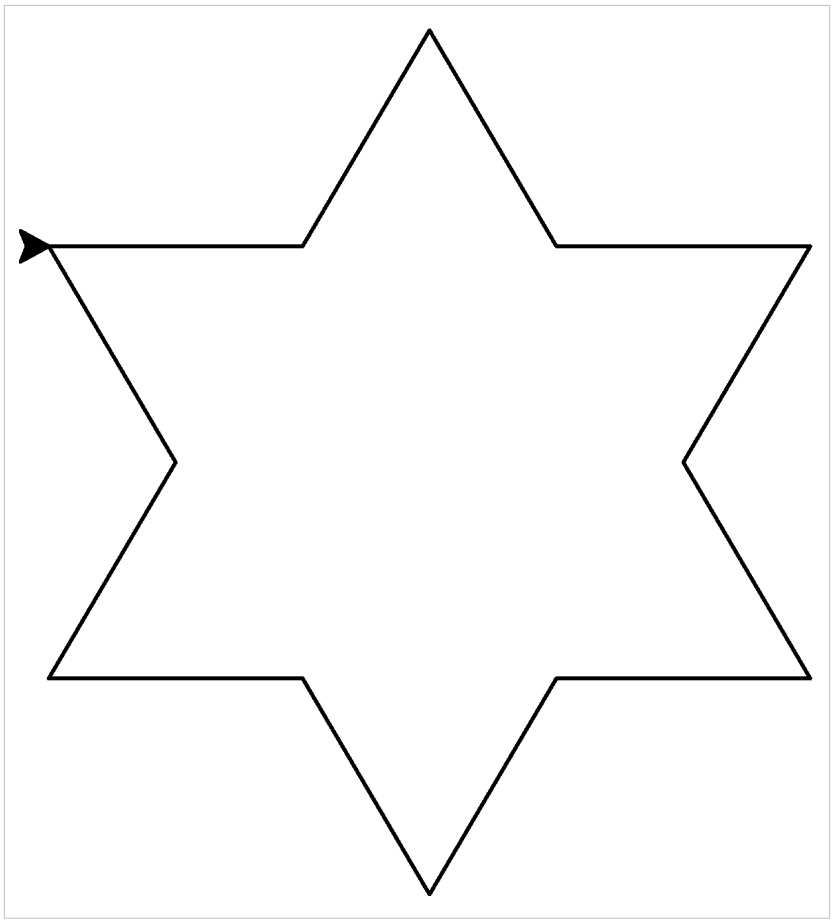
Le remplacement des lettres mnémotechniques par des instructions en Python est à l'image de ce que fait un compilateur comme celui de Python. Avec ça, au moins, la recette pour dessiner un triangle est facile à retenir: *avancer; tourner; avancer; tourner; avancer; tourner*, étant entendu que chaque fois qu'on avance, c'est de 100 pixels, et chaque fois qu'on tourne, c'est de 120° vers la droite.

Modification du script

Pour transformer le triangle en flocon, on doit remplacer chaque instruction *avancer* par la séquence *avancer; gauche; avancer; droite; avancer; gauche; avancer*. Du moment que chaque fois qu'on avance, c'est du même nombre de pixels (par exemple 81) et chaque fois qu'on tourne à gauche, c'est de 60° et chaque fois qu'on tourne à droite, c'est de 120°. Pour obtenir cet effet, il suffit de remplacer chaque *A* par *ApAmApA*:

```
from turtle import *
from re import *

programme='AmAmAm'

programme=sub('A','ApAmApA',programme)

programme=sub('A','fd(81); ',programme)
programme=sub('m','rt(120); ',programme)
programme=sub('p','lt(60); ',programme)
exec(programme)
```

Ce script dessine bien une étoile:

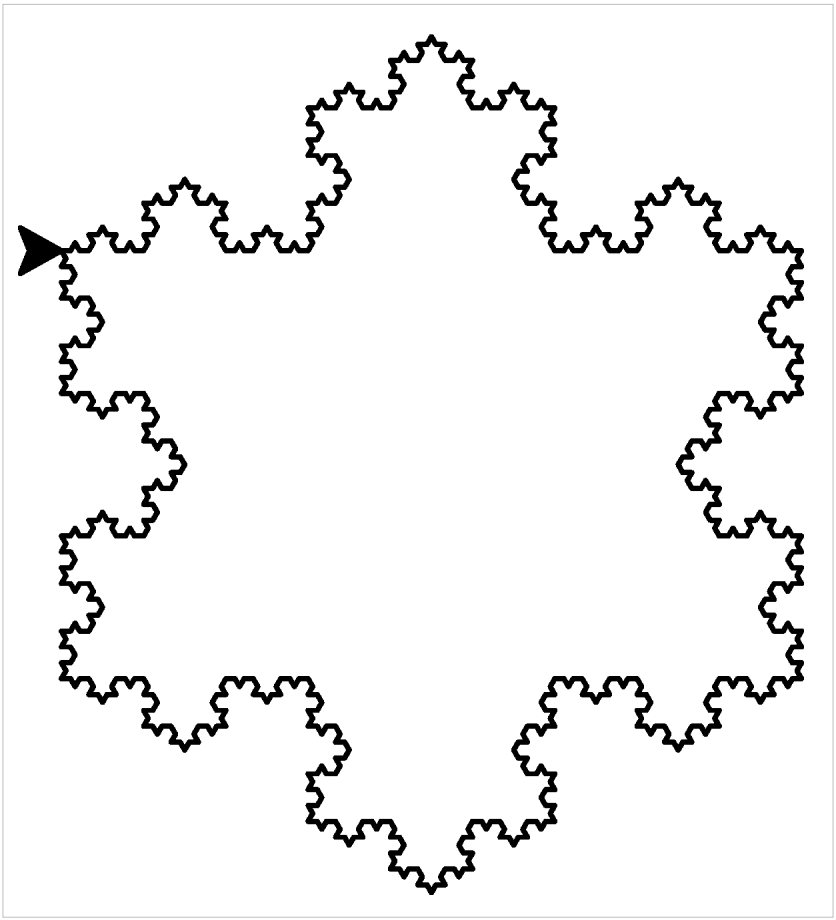


Dessin du triangle de Von Koch

Pour finir le dessin du flocon fractal, il suffit d'itérer le remplacement de chaque *A* par *ApAmApA*:

```
from turtle import *
from re import *

programme='AmAmAm'

for n in range(4):
    programme=sub('A','ApAmApA',programme)

programme=sub('A','fd(2); ',programme)
programme=sub('m','rt(120); ',programme)
programme=sub('p','lt(60); ',programme)
exec(programme)
```

Ce script dessine ceci en 9 lignes de Python (la tortue donne une idée de l'échelle):



Résolution de systèmes en Python

Un petit exercice

Au village de Trokhafairtrade dans le Swazibwana occidental, on pratique un mélange de commerce équitable et de troc. Ainsi, un habitant a acquis 2 youkoulélés d'Hawaii contre 3 xylophones et 1 € et un écotouriste a acquis un xylophone et un youkoulélé pour la modique somme de 8 €. On demande le prix, en euros, d'un xylophone et le prix d'un youkoulélé sur le marché de Trokhafairtrade.

Bien entendu, on va noter *x* le prix d'un *xylophone*, et *y* le prix d'un *youkoulélé*. Les données de l'énoncé se traduisant algébriquement par *2y=3x+1* et *x+y=8*.

On va donc voir comment résoudre le système de deux équations à deux inconnues suivant:

$$\left\{\begin{array}{rcl} 3x-2y & = & -1 \\ x+y & = & 8 \end{array}\right.$$

Méthode graphique

Une méthode simple (surtout ici puisque la solution est formée d'entiers) consiste à tracer les deux droites d'équations respectives $3x-2y=-1$et $x+y=8$et de lire sur le graphique les coordonnées de leur point d'intersection. *Python* tout seul ne sait pas faire ça mais *PyKig* lance le logiciel Kig et le laisse faire le travail, à condition de lui fournir les éléments nécessaires pour faire les constructions géométriques: Les coordonnées de points de chaque droite. Or la droite d'équation $ax+by=c$passe par les points de coordonnées $\left(\frac{c}{a};0\right)$et $\left(0;\frac{b}{a}\right)$(intersections avec les axes de coordonnées) qui sont contructibles par *Kig*. Le script suivant colorie en bleu la première droite (le milieu M sert de point d'ancrage pour son équation réduite) et en rouge la seconde droite, puis affiche en mauve leur point d'intersection:

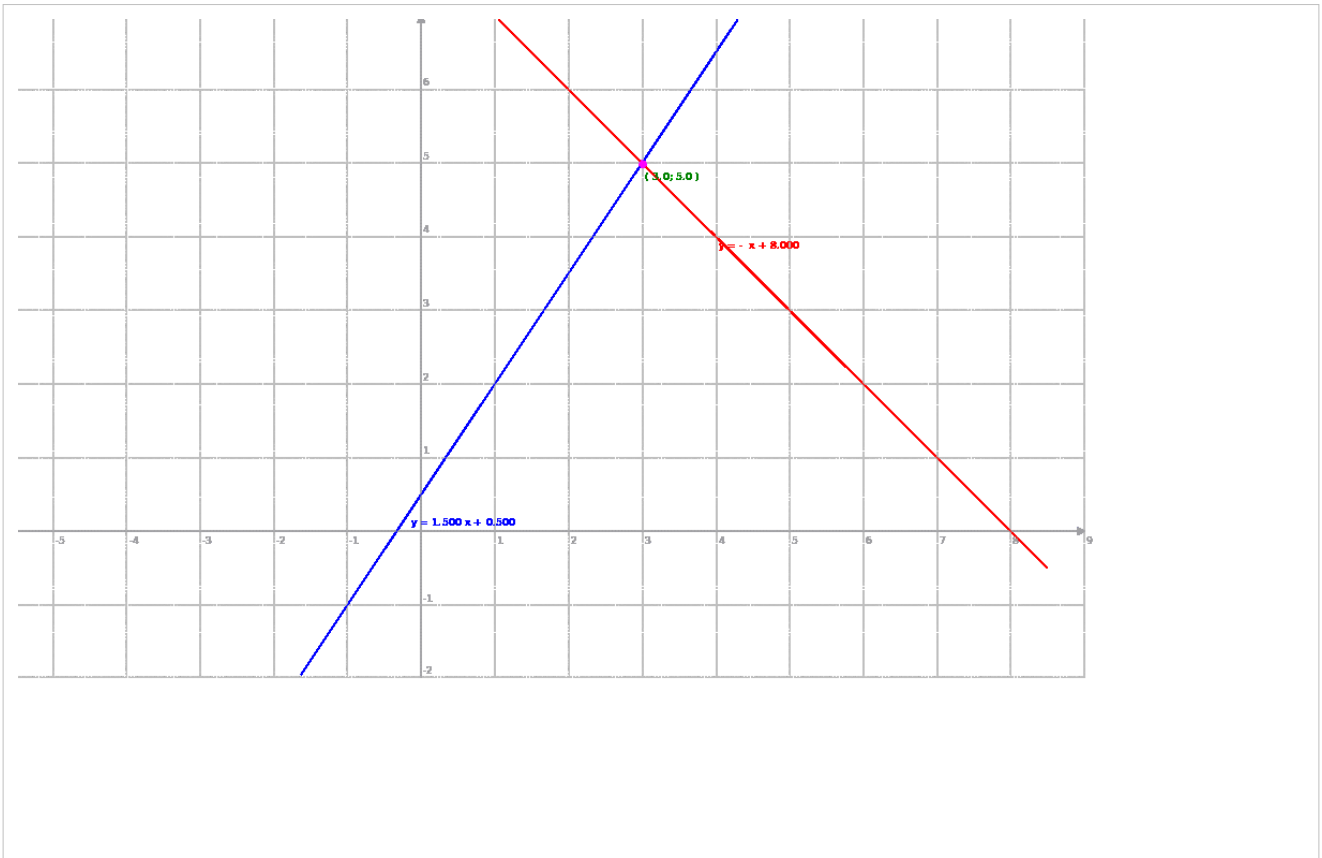
```
e1=[3.0,-2.0,-1.0]
e2=[1.0,1.0,8.0]


A=Point(e1[2]/e1[0],0,HIDDEN)
B=Point(0,e1[2]/e1[1],HIDDEN)
d1=Line(A,B)
d1.setcolor("blue")
M=MidPoints(A,B,HIDDEN)
t=Text(M,Equation(d1),0)
t.setcolor("blue")


C=Point(e2[2]/e2[0],0,HIDDEN)
D=Point(0,e2[2]/e2[1],HIDDEN)
d2=Line(C,D)
d2.setcolor("red")
N=MidPoints(C,D,HIDDEN)
u=Text(N,Equation(d2),0)
u.setcolor("red")
```

```
I=LineLineIntersection(d1,d2)
I.setcolor("magenta")
```

En enregistrant son contenu dans un fichier appelé *system.kpy*, et en exécutant dans une console ceci:

```
pykig.py system.kpy
```

On obtient, après lancement inopiné de *Kig* (à condition bien sûr que celui-ci soit installé, ce qui est très facile sous Linux, beaucoup moins sous les autres systèmes), la figure suivante (où il a été nécessaire de créer un *Label* donnant les coordonnées du point d'intersection, ce que *PyKig* ne sait pas faire):



On lit les coordonnées du point d'intersection, qui constituent la solution du système. Il est possible de déplacer à la souris les points A, B, C et D et voir les coordonnées du point d'intersection se mettre à jour en temps réel: En effet *Kig* est un logiciel de géométrie dynamique.

Cette méthode n'est pas applicable à des systèmes de plus de deux inconnues mais elle est très visuelle pour des systèmes tels que $\left\{\begin{array}{rcl} 4x+2y & = & 3 \\ 2x+y & = & 5 \end{array}\right.$ pour lesquels on voit non seulement qu'il n'y a pas de solution, mais également pourquoi il n'y a pas de solution.

Si l'une des droites est verticale ou horizontale, le fichier ci-dessus doit être modifié avec des tests gérant ces cas. Ceci est laissé en exercice.

Méthode itérative

Si on sait que x et y sont entiers naturels, on peut résoudre le système par une double boucle sur x et y:

```
solutions=[(x,y) for x in range(100) for y in range(200) if 3*x-2*y==-1
 and x+y==8]
print(solutions)
```

Pour "x"et"y" appartenant à R je propose le script suivant :

def systeme (a1,b1,c1,a2,b2,c2):

```
   x=float()
   y=float()
   'a1*x + b1*y =c1,a2*x + b2*y =c2'
   if a1*b2-a2*b1==0:
       print('Pas de solution')
   else:
       y=(c2*a1-c1*a2)/(a1*b2-a2*b1)
       x=(c1-b1*y)/a1
       print('x =',round(x,2),"",'y =',round(y,2))
```

Méthode de Cramer

La méthode de Cramer est également implémentable en *Python*:

```
def affiche(e):
    print(str(e[0])+'x+('+str(e[1])+')y='+str(e[2]))

def resoudre(e1,e2):
    determinant=e1[0]*e2[1]-e1[1]*e2[0]
    if determinant==0:
        print('Pas de solution unique')
    else:
        x=(e1[2]*e2[1]-e1[1]*e2[2])/determinant
        y=(e1[0]*e2[2]-e1[2]*e2[0])/determinant
        print('La solution est ('+str(x)+','+str(y)+')')


e=[3,-2,-1]
f=[1,1,8]

affiche(e)
affiche(f)
resoudre(e,f)
```

Mais le module *fractions* permet d'avoir la valeur exacte de la solution chaque fois que les coefficients du système sont entiers:

```
from fractions import *

def resoudre(e1,e2):
```

```
    determinant=e1[0]*e2[1]-e1[1]*e2[0]
    if determinant==0:
        print('Pas de solution unique')
    else:
        x=Fraction(e1[2]*e2[1]-e1[1]*e2[2],determinant)
        y=Fraction(e1[0]*e2[2]-e1[2]*e2[0],determinant)
        print('La solution est ('+str(x)+','+str(y)+')')


e=[3,-2,-1]
f=[1,1,8]
resoudre(e,f)
```

Avec NumPy

Le module *NumPy* permet de faire du calcul avec *Python*, et même en étant à la fois rapide et précis (parce que ce module est précompilé). Une fois installé, il suffit pour résoudre le système d'entrer le script suivant:

```
from numpy import *

A=matrix([[3,-2],[1,1]])
B=matrix([[-1],[8]])
solution=linalg.solve(A,B)
print(solution)
```

On peut télécharger *NumPy* sur sa page sourceforge [1]

Références

[1] http://sourceforge.net/projects/numpy/files/

Triplets pythagoriciens en Python

L'énoncé du problème est simple, sa solution avec *Python* aussi:

Énoncé

Trouver tous les triplets pythagoriciens (x,y,z) tels que $x+y+z=1000$; autrement dit, on demande les triangles rectangles de périmètre 1000 dont les côtés sont entiers.

En considérant *x*, *y* et *z* comme classés dans l'ordre croissant, on va faire une boucle sur *y* (le plus grand des côtés de l'angle droit), et à l'intérieur de celle-ci, une autre boucle sur *x*. Enfin on calcule l'hypoténuse *z*, puis le périmètre du triangle. Et si celui-ci vaut 1000, on affiche le triplet:

```
from math import hypot

for y in range(1000):
    for x in range(y):
        z=hypot(x,y)
        if x+y+z==1000:
            print(x,y,z)
```

Le script nous apprend qu'il n'y a qu'un seul triplet pythagoricien de somme 1000.

Systèmes congruentiels en Python

Rallye mathématique de la Réunion

Sujet 2005, Exercice 2

Énoncé

Pour organiser une grande manifestation sportive, le professeur d'éducation physique doit rassembler sur le stade un important groupe d'élèves. Le nombre d'élèves est compris entre 2 800 et 2 900. Il en profite pour leur faire remarquer que, regroupés par 2, puis par 3, puis par 4, puis par 5, puis par 6, il en reste toujours 1 ; mais, ô miracle, en se regroupant par 7, il ne reste personne.

On demande combien d'élèves il y a au total.

On peut utiliser une boucle avec plein de tests pour détecter les solutions mais aussi les ensembles. C'est ce qu'on va faire ici.

Construction de l'ensemble 2 par 2

On commence par construire la liste des nombres entre 2800 et 2900 tels que 2 par 2, il en reste 1: Ces nombres constituent une suite arithmétique de raison 2.

```
s2=[n for n in range(2800,2900) if n%2==1]

print(len(s2))
```

À ce stade, il y a beaucoup de candidats possibles:

```
{2817, 2819, 2821, 2823, 2825, 2827, 2829, 2831, 2833, 2835, 2837, 2839, 2841,
2843, 2845, 2847, 2849, 2851, 2853, 2855, 2857, 2859, 2861, 2863, 2865, 2867,
```

```
2869, 2871, 2873, 2875, 2877, 2879, 2881, 2883, 2885, 2887, 2889, 2891, 2893,
2895, 2897, 2899, 2801, 2803, 2805, 2807, 2809, 2811, 2813, 2815}
```

Épuration 3 par 3

Maintenant on va construire un ensemble analogue *s3* pour les nombres tels que, 3 par 3, il en reste 1, choisis parmi les nombres de *s2*:

```
s3=[n for n in s2 if n%3==1]

print(s3)
```

Il en reste déjà moins:

```
{2881, 2851, 2821, 2887, 2857, 2827, 2893, 2863, 2833, 2899, 2869, 2803, 2839,
2809, 2875, 2845, 2815}
```

Cas du nombre 4

Cette fois-ci, dans *s4* on met les nombres précédents tels que pris 4 par 4, il en reste 1:

```
s4=[n for n in s3 if n%4==1]

print(s4)
```

```
{2881, 2821, 2857, 2893, 2833, 2869, 2809, 2845}
```

Ça se précise!

Avec 5

Maintenant on va de 5 en 5:

```
s5=[n for n in s4 if n%5==1]

print(s5)
```

```
{2881, 2821}
```

Il ne reste que deux nombres à tester!

Avec 6

On pourrait les tester l'un après l'autre, mais aussi faire pareil qu'avant (on y prend goût !):

```
s6=[n for n in s5 if n%6==1]

print(s6==s5)
```

Aucun changement, toujours deux solutions:

```
{2881, 2821}
```

Dernière étape

Autant continuer sur la même voie:

```
solutions=[n for n in s6 if n%7==0]

print(solutions)
```

Ce qui donne la réponse (on constate qu'elle est unique) à la question de l'énoncé.

Sujet 2007, Exercice 3

Énoncé

Chaque semaine, Jean ramasse entre 40 et 200 œufs qu'il va vendre au marché.

Ce soir, veille de marché, il est perplexe.

• S'il met ses œufs dans des emballages de 6, il en reste 2.

• S'il utilise des emballages de 10, il en reste encore 2.

• Il me faudrait, dit-il, des emballages de 8 pour tout contenir exactement.

On demande combien il y a d'œufs en tout.

Par 6

On commence par examiner les nombres tels que, pris 6 par 6, il en reste toujours 2:

```
s6=[n for n in range(40,200) if n%6==2]

print(len(s6))
```

26 nombres à examiner, courage!

Par 10

```
s10=[n for n in s6 if n%10==2]

print(s10)
```

Plus que 5 nombres à examiner!

Par 8

```
solutions=[n for n in s10 if n%8==0]

print(solutions)
```

Comme une seule valeur de *n* est affichée, c'est la seule qui a réussi tous les tests, et c'est donc l'unique solution au problème.

Freudenthal en Python

Le problème de Freudenthal est intéressant à traiter en *Python* parce qu'il peut se résoudre en manipulant des objets (tableaux et ensembles). Le problème est d'ailleurs intéressant en soi parce qu'il porte sur la logique épistémique.

En voici l'énoncé traduit du Néerlandais à l'Anglais puis au Français:

On choisit au hasard deux entiers x et y strictement supérieurs à 1, x étant le plus petit des deux, et on donne à Sam leur somme qui est inférieure à 100, et à Polly leur produit. Après un temps de réflexion suffisamment long, le dialogue suivant se déroule entre Sam et Polly:

- Polly: Je ne sais pas qui sont x et y.
- Sam: Je savais que tu ne savais pas!
- Polly: Alors je sais qui sont x et y.
- Sam: Alors je sais aussi!

Le but du problème est donc d'utiliser les connaissances qu'on a sur les connaissances de Polly et Sam pour savoir si on peut savoir quels sont *x* et *y*.

Un Outil

Pour se faciliter la suite, on va créer une fonction *Python* qui, à chaque somme *n*, associe la liste des produits qui ont donné cette somme:

```
def prod(n):
    p=[]
    for k in range(2,n-2):
        p.append(k*(n-k))
    return p
```

Alors *prod(8)* donne la liste *[12, 15, 16, 15, 12]* parce que 8 peut s'écrire 2+6, 3+5, 4+4, 5+3 ou 6+2 et que les produits correspondants sont 12, 15 et 16 (certains apparaissant deux fois).

Première affirmation

L'affirmation apporte une information: Le produit donné à Polly peut s'obtenir de plusieurs manières, sinon Polly connaîtrait les facteurs. Pour exploiter cette information, on va commencer par fabriquer l'énorme liste des produits possibles, puis ne garder que ceux qui apparaissent au moins deux fois dans la liste:

```
produits=[x*y for y in range(3,100) for x in range(2,y) if x+y<=100]
print(len(produits))

polly=[p for p in produits if produits.count(p)>=2]
print(len(polly))
```

Ceci dit, il en reste encore pas mal...

Deuxième affirmation

Si Sam sait que Polly ne sait pas, c'est parce que quelle que soit la décomposition en somme d'entiers de celui qu'on lui a dicté, le produit correspondant est dans la liste précédente. Sinon Polly aurait pu l'entendre et aurait alors su quels en sont les facteurs. Sam ne va donc garder que les sommes *n* pour lesquelles la liste *prod(n)* calculée avec la fonction ci-dessus ne contient que des éléments de la liste *polly*, donc si leur intersection a une longueur maximale (soit *n-4*):

```
sam=[n for n in range(4,100) if len([p for p in prod(n) if p in
polly])==n-4]
print(sam)
```

Mais le nombre 4 qui apparaît est une erreur, dûe à ce que la liste est vide, donc de longueur 4-4=0. Or 4 ne peut s'écrire que 2+2 et *x* et *y* sont supposés différents. Donc on va plutôt commencer par 5:

```
sam=[n for n in range(5,100) if len([p for p in prod(n) if p in
polly])==n-4]
print(sam)
```

On voit alors apparaître la liste des sommes que Sam a pu somme toute ouïr:

[11, 17, 23, 27, 29, 35, 37, 41, 47, 53]

Troisième affirmation

La dernière affirmation de Sam lève toute ambigüité chez Polly. Mais quel genre d'ambiguïté peut-il y avoir encore? Par exemple, un des produits associés à 11 est 30 (car 11=6+5) et 30 est aussi un produit associé à 17 (car 17=15+2). Si le produit 30 avait été confié à Polly, l'ambiguïté en question resterait présente. Polly va donc enlever aux listes *prod(11)*, *prod(17)* etc. les doublons. On va donc créer la liste des doublons avec

```
doublons=[]
for p in sam:
    for q in sam:
        if q<>p:
            doublons+=([r for r in prod(p) if r in prod(q)])
```

Puis on va enlever à chaque liste des produits (de 11, de 17 etc.) chaque doublon. Si Polly connaît la somme de Sam, c'est parce que dans la liste *sam*, il ne restera qu'une liste de produits contenant exactement deux produits ($x \times y$ et $y \times x$). Il ne reste alors plus qu'à la chercher pour en savoir autant que Polly:

```
solutions=[p for p in sam if len([r for r in prod(p) if r not in
doublons])==2]
print(solutions)
```

On connaît donc la somme de Sam.

Quatrième affirmation

Puisque la somme de Sam vaut 17, on recommence l'étape précédente avec 17 seulement: Chercher la liste des produits de 17, doublons enlevés:

```
print([r for r in prod(17) if r not in doublons])
```

On connaît maintenant le produit de Polly.

Recherche de x et y

Trouver deux nombres dont le produit est 52 et la somme 17 est un problème classique sur les équations du second degré. Mais *Python* permet aussi de les trouver avec une double boucle (puisqu'on sait que ces nombres sont entiers):

```
print([(x,y) for y in range(100) for x in range(y) if x+y==17 and
x*y==52])
```

La solution (x,y) apparaît alors entre crochets.

Nombres entiers en Ruby

La particularité des nombres entiers, c'est que chacun possède un successeur et un prédécesseur. Et bien *Ruby* sait les calculer (certes ce n'est pas très difficile, il suffit d'additionner ou soustraire 1 à un nombre entier pour avoir le suivant ou le précédent).

Obtention d'un nombre entier

Avec une chaîne de caractères

Si on entre le script suivant:

```
a=7
puts(a)
```

on a exactement le même effet que si on entre

```
a="7"
puts(a)
```

du moins en apparence. Parce que si on essaye d'additionner 2, avec

```
a=7
puts(a+2)
```

on a 9, alors qu'avec

```
a="7"
puts(a+2)
```

on a un message d'erreur: On ne peut pas additionner un nombre et une chaîne de caractères !

Pour convertir une chaîne de caractères en entier, on utilise la méthode *to_i* de celle-ci. Ainsi

```
a="7"
b=a.to_i
puts(b+2)
```

donne bien 9.

Un autre moyen d'obtenir un entier (naturel) avec une chaîne de caractères, c'est de compter le nombre de lettres de celle-ci. Ce qui se fait avec sa propriété *length*:

```
t="abracadabrantesque"
n=t.length
puts(n)
```

Avec un réel

La méthode *to_i* permet aussi de convertir un réel en entier. Ce qui est parfois nécessaire parce que pour *Ruby*, $\sqrt{100}$ n'est pas un entier:

```
a=Math.sqrt(100)
puts(a.integer?)
```

Affiche *false* parce que pour *Ruby*, le nombre calculé est 10.0 (considéré comme réel et non comme entier) et sa méthode *to_i* change son type, en le transformant en un entier:

```
a=Math.sqrt(100).to_i
puts(a.integer?)
```

affiche bien *true*.

```
a=3.9999999
b=a.to_i
puts(b)
```

n'a peut-être pas l'effet escompté, puisque *a* a été choisi proche de 4, et qu'on obtient 3. C'est que la conversion en entier se fait par une troncature et pas par un arrondi. En fait *to_i* a le même effet que *floor*:

```
a=3.9999999
b=a.floor
puts(b)
```

Si on veut arrondir au-dessus, on utilise la méthode *ceil* d'un réel:

```
a=3.9999999
b=a.ceil
puts(b)
```

mais là on tombe dans le problème inverse:

```
a=3.0000001
b=a.ceil
puts(b)
```

donne aussi 4 !

Pour arrondir au mieux, on utilise la méthode *round*:

```
a=3.9999999
b=a.round
puts(b)
```

Avec un entier

Pour avoir le successeur d'un entier, on utilise la méthode *succ*:

```
puts(7.succ)
```

nous apprend que 7+1=8 (on s'en doutait un peu...), alors que

```
puts(7.pred)
```

montre que 7-1=6. Mais contrairement au premier des axiomes de Peano, 0 possède un prédécesseur (-1) parce que pour *Ruby*, les entiers sont relatifs et pas seulement naturels.

Pour avoir l'opposé d'un entier, on le précède d'un signe "moins". Ainsi,

```
a=-5
puts(-a)
```

Donne 5, car l'opposé de -5 est 5.

tests sur les entiers

Pour savoir si 2 est entier (on ne sait jamais), on peut le vérifier par

```
puts(2.integer?)
```

Ce test a été utilisé ci-dessus pour vérifier que 10 est entier, et on a eu raison de se méfier !

On peut aussi vérifier si un entier est premier, avec *mathn*:

```
require 'mathn'
a=2**32+1
puts(a.prime?)
```

nous apprend que 4 294 967 297 n'est pas premier, contrairement à ce qu'avait conjecturé Fermat.

Opérations

Addition, soustraction et multiplication

Dans *Ruby*, les opérations arithmétiques sont notées +, - et * sans grande surprise. Ces opérations peuvent porter sur des entiers négatifs:

```
a=5
b=-8
puts(a+b)
puts(a-b)
puts(a*b)
```

Division

Par défaut, la division des entiers est la division euclidienne. Son quotient est donc un entier (et n'est pas le quotient exact)

Quotient

le script suivant:

```
num=3
den=2
q=num/den
puts(q)
```

affiche 1 et pas 1,5 parce que le quotient euclidien de 3 par 2 est 1 (avec un reste de 1) et pas 1,5...

Si on veut le quotient exact, on doit remplacer l'un des entiers par un réel avec un point décimal. Pour avoir 1,5, on peut essayer l'une des possibilités suivantes

```
puts(3.0/2)
puts(3/2.0)
puts(3.0/2.0)
puts(3.to_f/2)
```

Mais dans ce cas, on travaille sur des valeurs approchés. Pour avoir les valeurs exactes, il faut utiliser des fractions (voir à Mathématiques avec Python et Ruby/Fractions en Ruby). Et bien entendu, toute tentative de division par 0 donne un message d'erreur.

Reste

Lorsqu'on divise euclidiennement 13 par 8, le quotient est donc égal à 1. Mais il reste 5. Pour calculer directement ce reste en *Ruby*, on peut utiliser le symbole %:

```
a=13
b=8
r=a%b
puts(r)
```

Cette opération permet de travailler sur les congruences. Remarque: Si b=0, on a le même message d'erreur que lorsqu'on divise par 0.

Primalité

En *Ruby*, l'opération pgcd est infixée. Pour chercher le plus grand entier qui divise à la fois 13572468 et 12345678, on entre

```
a=13572468
b=12345678
g=a.gcd(b)
puts(g)
```

Bien entendu, *a.gcd(b)* et *b.gcd(a)* donnent le même résultat.

De même, le ppcm se calcule de façon analogue avec *a.lcm(b)*.

Lorsqu'un entier n'a pas de diviseurs non triviaux, il est dit *premier*, et on a vu ci-dessus qu'avec *mathn* on peut tester si un nombre est premier. Avec *prime* aussi:

```
require 'prime'
n=2010
puts(n.prime?)
puts(n.prime_division)
```

Et en bonus, la décomposition en facteurs premiers!

Puissances

L'opérateur d'élévation à la puissance se note avec l'astérisque de la multiplication, mais dédoublée:

```
a=4
b=2
puts(a**b)
puts(b**a)
```

pour vérifier que $4^2 = 2^4$(Exercice: Quelles sont les autres solutions de l'équation $a^b = b^a$?)

Remarques:

1. Si l'exposant est négatif, le résultat est une fraction;
2. Si l'exposant est réel, le résultat est réel aussi.

Priorités opératoires

En *Ruby* comme en algèbre, on effectue dans l'ordre

1. Les parenthèses
2. Les fonctions (comme l'élévation à une puissance)
3. Les multiplications et divisions
4. Les additions et soustractions.

Ainsi

```
puts(2+3*5)
```

affiche 17 et non 25: Les opérations ne sont pas effectuées de gauche à droite, mais en suivant les priorités opératoires.

Entiers et itération

Itérateur

La méthode la plus utile d'un nombre entier est sans conteste le bouclage, qui permet de répéter quelque chose de répétitif. Il suffit de dire à *Ruby* ce qu'il doit répéter (entre *do* et *end*) et combien de fois il doit le répéter: Un entier!

Bis repetita

Pour écrire un message très enthousiaste, on peut écrire

```
oui=10
oui.times do puts("Yes!") end
```

Avec un indice

Pour additionner les entiers successifs, on peut le faire avec

```
somme=0
n=10
n.times do |indice| somme+=indice end
puts(somme)
```

Le fait de mettre la variable entre traits verticaux lui donne automatiquement les valeurs entières successives. Mais la somme est affichée égale à 45 alors que 1+2+3+4+5+6+7+8+9+10=55...

Pour savoir d'où vient cette erreur de comptage, on peut essayer

```
n=10
n.times do |indice| puts(indice) end
```

Bingo! Les 10 premiers entiers naturels vont de 0 à 9, pas de 1 à 10.

Pour éviter de commencer par 0, on peut explicitement commencer par 1:

```
(1..10).inject {|somme, indice| somme+indice}
```

Mais cette fois-ci, on ne travaille plus avec un entier mais avec une liste d'entiers. Pour un tel objet, *inject* est une méthode typique de *Ruby* qui permet d'injecter à la liste un bloc d'instructions. Le bloc comprend deux variables locales, la première des deux (*somme*) étant destinée à se faire injecter des doses successives du médicament, la seconde (*indice*) représentant les doses de médicament à injecter l'une après l'autre.

Boucles

Ruby permet aussi de faire de la programmation impérative avec des boucles:

à nombre prédéterminé d'exécutions

Pour additionner les entiers de 1 à 10, on peut aussi faire comme ceci:

```
somme=0
for indice in 1..10 do
    somme+=indice
end
puts(somme)
```

Cette fois-ci on a bien 55.

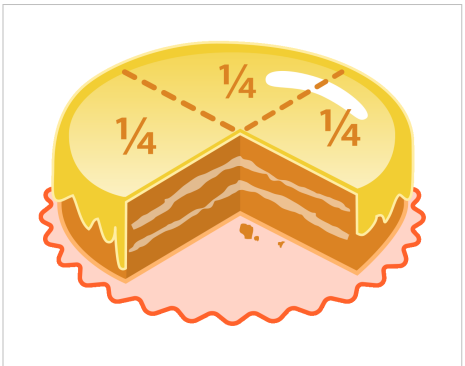
à condition de sortie

La même somme peut aussi être calculée avec cette boucle:

```
somme,indice=0,0
while indice<=10 do
    somme+=indice
    indice=indice.succ
end
puts(somme)
```

Fractions en Ruby

"Dieu fit le nombre entier, le reste est l'oeuvre de l'Homme", disait Leopold Kronecker. Le début du reste ce fut incontestablement les fractions, définies comme quotients d'entiers, et pratiquées bien avant les nombres décimaux.



Obtention d'une fraction

Pour entrer la fraction $\frac{24}{10}$, on peut entrer

```
a=Rational(24,10)
puts(a)
```

qui la simplifie automatiquement. Alternativement, on peut charger *mathn*, ce après quoi le symbole de division donne des fractions au lieu de la division euclidienne:

```
require 'mathn'
a=24/10
puts(a)
```

On peut également obtenir une fraction à partir d'un réel, avec la méthode *to_r* (*r* comme *rational*). Mais la fraction n'est correcte que si son dénominateur est une puissance de 2:

```
a=1.2
b=a.to_r
puts(b)
```

Certes, $\frac{5404319552844595}{4503599627370496} \simeq \frac{6}{5}$mais tout de même...

Propriétés d'une fraction

Numérateur

Pour avoir le numérateur d'une fraction *f*, on entre *f.numerator*:

```
a=Rational(24,10)
puts(a.numerator)
```

Dénominateur

Pour avoir le dénominateur d'une fraction *f*, on entre *f.denominator*:

```
a=Rational(24,10)
puts(a.denominator)
```

Valeur approchée

Pour avoir la valeur approchée d'une fraction, on convertit celle-ci en un réel:

```
a=Rational(24,10)
puts(a.to_f)
```

Opérations sur les fractions

Opérations unaires

Opposé

L'opposé d'un nombre, en particulier d'une fraction, s'obtient en le faisant précéder d'un signe -:

```
a=Rational(2,-3)
puts(-a)
```

Inverse

Pour obtenir l'inverse d'une fraction, on divise 1 par celle-ci:

```
a=Rational(5,4)
puts(1/a)
```

Addition

Pour additionner deux fractions, on met le signe + entre elles, et le résultat est une fraction (même si celle-ci est entière, comme par exemple $\frac{1}{1}$):

```
a=Rational(34,21)
b=Rational(21,13)
puts(a+b)
```

Soustraction

La différence de deux fractions est une fraction:

```
a=Rational(34,21)
b=Rational(21,13)
puts(a-b)
```

Multiplication

Le produit de deux fractions est une fraction:

```
a=Rational(34,21)
b=Rational(21,13)
puts(a*b)
```

Division

Le quotient de deux fractions (à condition que la deuxième ne soit pas nulle) est une fraction:

```
a=Rational(34,21)
b=Rational(21,13)
puts(a/b)
```

Et même le reste euclidien est défini entre fractions, et le résultat est encore une fraction:

```
a=Rational(32,7)
b=Rational(7,2)
puts(a%b)
```

Exemple

On voudrait savoir quel est le rapport des longueurs des tuyaux d'orgue

1. Entre un gros Nasard et un Nasard;
2. Entre un gros Nasard et une grosse Tierce.

Ces rapports sont affichés par *Ruby* sous forme de fractions, même le premier d'entre eux qui est entier (ce qui ne sautait pas aux yeux !):

```
gn=5+Rational(1,3)
n=2+Rational(2,3)
gt=3+Rational(1,5)
puts(gn/n)
puts(gn/gt)
```

Puissance

Une puissance entière (même négative) d'une fraction) est encore une fraction:

```
a=Rational(3,2)
puts(a**12)
puts(a**(-2))
```

Mais si l'exposant est décimal, même si le résultat est une fraction, *Ruby* le considère comme un réel:

```
a=Rational(9,4)
b=a**0.5
puts(b)
puts(b.to_r)
```

Algorithmes

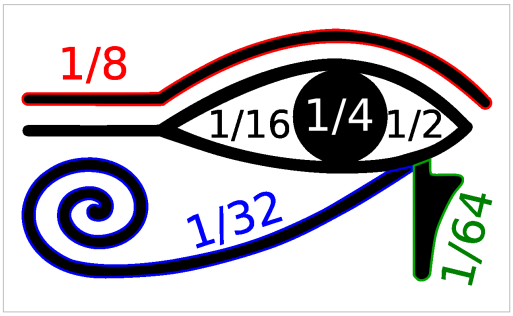
Réduite de Farey

Pour calculer la *médiane* (ou réduite) de Farey de deux fractions *a* et *b*, on définit une fonction *Ruby* de deux variables, qui s'appelle *Farey*:

```
def Farey(a,b)
    n=a.numerator+b.numerator
    d=a.denominator+b.denominator
    return Rational(n,d)
end


a=Rational(3,4)
b=Rational(1,13)
puts(Farey(a,b))
```

Fractions égyptiennes

Pour écrire une fraction à l'égyptienne (comme somme de fractions de numérateur 1), on applique l'algorithme de Fibonacci:




```
def egypt(f)
    e=f.to_i
    f-=e
    liste=[e]
    begin
```

```
        e=Rational(1,(1/f).to_i+1)
        f-=e
        liste.push(e)
    end while f.numerator>1
    liste.push(f)
    return liste
end

require 'mathn'

a=21/13
puts(egypt(a))
```

On peut résumer ce script *Ruby* aux étapes suivantes:

1. On commence par extraire la partie entière de *f*, pour qu'il reste une fraction inférieure à 1;
2. On soustrait à *f* (fraction restante) le plus grand inverse d'entier possible...
3. On s'arrête quand le reste est lui-même un inverse d'entier (autrement dit, on continue tant que son numérateur est plus grand que 1).
4. On ajoute à la liste, la dernière fraction obtenue.

Nombres réels en Ruby

Écriture décimale

Depuis l'apparition de chiffres arabes et de la numération de position, les nombres décimaux sont devenus plus concrets que les fractions: En écrivant $\frac{6}{5}$, on voit deux nombres et on a tendance à oublier que cette écriture désigne un seul nombre (le quotient de 6 par 5). Alors qu'en écrivant ce nombre 1,2 on voit immédiatement qu'il n'y en a qu'un seul !

Decimaux

Un nombre décimal est un nombre dont le développement décimal s'arrête quelque part. Les réels non décimaux sont donc ceux dont le développement décimal est infini, et on peut en construire exprès de cette manière comme le fit Liouville par exemple.

En *Ruby*, certains nombres décimaux ont quand même une infinité de chiffres parce qu'ils sont stockés en machine sous forme binaire et que, sous cette forme, ils ont une infinité de chiffres.

Fractions

Le développement décimal d'une fraction se remarque par le fait qu'un motif finit par se répéter indéfiniment, comme le montrent les exemples suivants:

```
puts(1.0/3)
puts(1.0/9.0)
puts(1/11.0)
puts(1.0/7)
```

Nombres irrationnels

Les premiers nombres irrationnels connus ont été les racines carrées des nombres entiers et le nombre d'or.

```
puts(2**0.5)
puts(Math.sqrt(2))
puts((1+5**0.5)/2)
puts((Math.sqrt(5)+1)/2)
```

D'autres sont *e* et π :

```
puts(Math::E)
puts(Math::PI)
```

Voici comment on peut calculer en *Ruby* la constante de Champernowne:

```
c='0.'
(1..40).collect { |n| c=c+n.to_s }
puts(c.to_f)
puts(c.to_r)
```

Le premier objet qui a été créé ci-dessus est une chaîne de caractères. *Ruby* le sait parce qu'on l'a mise entre guillemets. Initialement elle comprend le début de la représentation décimale de la constante (ou *0* suivi du point décimal). Le deuxième objet créé ci-dessus est une liste d'entiers (allant de 1 à 40). Cet objet est créé au vol, sans lui donner de nom, parce que la seule chose qu'on veuille faire avec lui, est d'appeler sa méthode *collect* qui fonctionne un peu comme le *times* des entiers: Un bloc d'instructions, entre accolades, avec un indice qui parcourt les éléments successifs du tableau, et ... une seule instruction, de concaténation de *n* (une fois transformé en chaîne de caractères avec *to_s*) et de la constante en cours de construction. Ceci fait, la constante est donc une chaîne de caratères, que *Ruby* peut transformer en un réel (*flottant*) ou en une fraction (*rationnel*).

Fonctions

Opérations

Les quatre opérations sont notées +, -, * et /. Dès que l'un des opérandes est écrit avec un point décimal, *Ruby* le reconnaît comme réel et l'opération donne un réel. La division peut même être euclidienne, ce qui permet notamment de calculer la valeur principale d'un angle en radians:

```
puts(100%Math::PI)
```

Le signe - peut aussi être unaire et dans ce cas, représente l'opposé du nombre qui le suit. Pour additionner un nombre *h* à un autre nombre *x*, on peut, au lieu de noter *x=x+h*, écrire *x+=h*.

Pour arrondir *x* à l'entier inférieur, on invoque *x.floor*; pour arrondir à l'entier supérieur, on invoque *x.ceil*. Pour calculer la valeur absolue de *x*, on invoque *x.abs*. Sa racine carrée se note indifféremment

```
r=2**0.5
puts(r)
r=Math.sqrt(2)
puts(r)
```

En effet, l'astérisque dédoublé code l'élévation à un exposant en *Ruby*.

Logarithmes et exponentielles

Logarithmes

Le script ci-dessous calcule et affiche l'image de 0,5 par le logarithme népérien, par le logarithme décimal, par les fonctions réciproques du cosinus hyperbolique, du sinus hyperbolique, de la tangente hyperbolique:

```
puts(Math.log(0.5))
puts(Math.log10(0.5))
puts(Math.acosh(0.5))
puts(Math.asinh(0.5))
puts(Math.atanh(0.5))
```

Exponentielles

Le script ci-dessous calcule et affiche l'image de 2 par l'exponentielle, par le cosinus hyperbolique, par le sinus hyperbolique, puis la tangente hyperbolique:

```
puts(Math.exp(2))
puts(Math.cosh(2))
puts(Math.sinh(2))
puts(Math.tanh(2))
```

Trigonométrie

Pour calculer les cosinus, sinus et tangente d'un radian, on peut faire comme ceci:

```
puts(Math.cos(1))
puts(Math.sin(1))
puts(Math.tan(1))
```

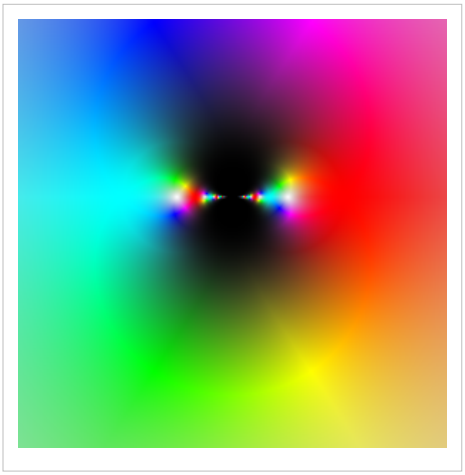
Pour connaître un angle en radians dont le cosinus, le sinus ou la tangente sont connus, on peut mettre un *a* devant la fonction:

```
puts(Math.acos(0.5))
puts(Math.asin(0.5))
puts(Math.atan(0.5))
```

Pour connaître un angle dont les côtés opposé et adjacent sont connus, on peut utiliser *Math.atan(y/x)* ou *Math.atan2(x,y)*. Et même pour calculer $\sqrt{x^2+y^2}$, on peut utiliser *Math.hypot(x,y)*. Par exemple, si on veut connaître les angles et l'hypoténuse d'un triangle rectangle de côtés 12 cm et 5 cm, on peut utiliser ce script:

```
cdr=180/Math::PI
a=12
b=5
puts(Math.atan2(a,b)*cdr)
puts(Math.atan2(b,a)*cdr)
puts(Math.hypot(a,b))
```

Nombres complexes en Ruby



On a inventé les nombres complexes juste parce qu'on voulait que certaines équations, comme $x^2 = -1$, aient une solution (dans le cas présent, notée *i*). C'est typique des mathématiques, ça:

1. On se fait un petit caprice;
2. Pour le satisfaire, on invente une grosse théorie;
3. D'autres gens utilisent cette théorie pour résoudre des problèmes, souvent issus des mathématiques appliquées, et qui n'ont plus rien de capricieux !

Instanciation d'un complexe

Pour créer dans *Ruby* le complexe *x+iy*, on utilise *Complex(x,y)*:

```
a=Complex(4,3)
puts(a)
```

Les nombres *x* et *y* sont juste des nombres: Ils peuvent très bien être des entiers ou des fractions. On appelle entier de Gauss un nombre complexe dont les parties réelle et imaginaire sont des entiers relatifs.

Opérations

La somme, la différence, le produit et le quotient de deux nombres complexes sont des nombres complexes:

```
a=Complex(2,3)
b=Complex(4,3)
puts(a+b)
puts(a-b)
puts(a*b)
puts(a/b)
```

Ces exemples illustrent le fait que la somme, la différence et le produit d'entiers de Gauss sont des entiers de Gauss. Par contre leur quotient exact ne l'est pas forcément. L'exemple de la soustraction montre que même lorsque le résultat d'une opération est réel, *Ruby* le considère quand même comme un complexe (de partie imaginaire nulle).

Ainsi,

```
i=Complex(0,1)
puts(i**2)
puts(i**i)
```

On voit que pour *Ruby*, $i^2 = -1 + 0i$ et non -1. On voit également que i^i est réel. En effet, la puissance se note toujours ** en *Ruby*, et l'exposant n'est pas nécessairement réel.
Il est donc possible de calculer "la" racine carrée d'un complexe *z* avec *z**0.5*. Mais si *7+24i* est le carré de deux complexes, "sa" racine carrée calculée par *Ruby* est *4+3i* et non *-4-3i*. Comment *Ruby* choisit-il parmi ces deux nombres?

De même, -1 a deux racines carrées dans $\mathbb{C}$: *i* et *-i*. *Ruby* n'en reconnaît qu'une, et on se doutait qu'il choisirait *i* plutôt que son opposé. Mais

```
puts((-1)**0.5)
puts(Complex(-1,0)**0.5)
```

Si -1 est réel, il n'a pas de racine carrée du tout (sous-entendu dans $\mathbb{R}$), alors que si -1 est complexe, "sa" racine carrée est proche de *i* mais pas exactement égale à *i* (sa partie réelle étant de l'ordre de 10^{-16}).

Propriétés

Les parties réelle et imaginaire d'un complexe *a* sont des propriétés de celui-ci:

```
a=Complex(4,3)
puts(a.real)
puts(a.imag)
puts(a.conj)
```

Il en est donc de même de son conjugué, qui est un complexe (contrairement à sa partie réelle, qui peut être un réel, mais aussi un entier ou une fraction, selon le cas).

D'autres propriétés d'un complexe sont son module et son argument:

```
a=Complex(4,3)
puts(a.abs)
puts(a.arg)
puts(a.polar)
```

z.polar permet d'avoir d'un coup le module et l'argument d'un complexe. Il permet de résoudre plus rapidement le problème vu dans le chapitre sur les nombres réels: Chercher le plus petit angle (en radians) et l'hypoténuse d'un triangle rectangle dont les côtés mesurent 12 cm et 5 cm:

```
a=12
b=5
z=Complex(a,b)
puts(z.polar)
```

Fonctions

CMath contient des versions complexes des fonctions trigonométriques, exponentielles et logarithmes:

Exponentielles

On peut retrouver l'écriture exponentielle des complexes de module 1 à condition de considérer l'exposant comme imaginaire (ci-dessous, pour vérifier numériquement que $e^{-\frac{\pi}{3}i} = \frac{1}{2} + i\frac{\sqrt{3}}{2}$):

```
require 'cmath'
t=Complex(0,Math::PI/3)
w=CMath.exp(t)
puts(w.real==0.5)
puts(w.real-0.5)
puts(w.imag==Math.sqrt(3)/2)
```

Comme d'habitude, 0,5 n'étant pas stocké très précisément en machine, l'égalité des parties réelles n'est qu'approximative.

Pour calculer les cosinus hyperbolique, sinus hyperbolique et tangente hyperbolique de *4+3i*, on fait ainsi:

```
require 'cmath'
a=Complex(4,3)
puts(CMath.cosh(a))
puts(CMath.sinh(a))
puts(CMath.tanh(a))
```

Logarithmes

Pour calculer les images de *4+3i* par les fonctions logarithme népérien, logarithme décimal, arguments des fonctions trigonométriques hyperboliques, on peut faire ainsi:

```
require 'cmath'
a=Complex(4,3)
puts(CMath.log(a))
puts(CMath.log10(a))
puts(CMath.acosh(a))
puts(CMath.asinh(a))
puts(CMath.atanh(a))
```

Fonctions trigonométriques

directes

Les cosinus, sinus et tangente d'un nombre complexe *z* se calculent avec le module *cmath*:

```
require 'cmath'
z=Complex(4,3)
puts(CMath.cos(z))
puts(CMath.sin(z))
puts(CMath.tan(z))
```

indirectes

Les fonctions trigonométriques inverses se calculent de manière analogue, en mettant juste un *C* devant *Math*:

```
require 'cmath'
z=Complex(4,3)
puts(CMath.acos(z))
puts(CMath.asin(z))
puts(CMath.atan(z))
```

La fonction *arc tangente* se calcule aussi avec deux nombres complexes:

```
require 'cmath'
a=Complex(4,3)
b=Complex(2,1)
puts(CMath.atan2(a,b))
```

Ça doit sûrement servir à quelque chose, mais à quoi?

Quaternions et octonions en Ruby

Complexes

On a vu dans le chapitre précédent que pour Ruby, un nombre complexe z est essentiellement une structure abritant deux réels, accessibles par *z.real* et *z.imag* respectivement. La construction de Cayley-Dickson généralise ce point de vue: En prenant deux complexes a et b, on peut les regrouper dans une nouvelle structure qui est considérée comme un nombre: Un quaternion.

Dans toute la suite, on va profiter de la gestion des fractions offerte par *cmath*, avec

```
require 'cmath'
```

Quaternions

Definition et affichage

Définition

La définition d'un quaternion se fait dans une classe nommée *Quaternion*:

```
class Quaternion

end
```

La première méthode, l'initialisation, crée donc deux variables a et b (qui seront des complexes, mais Ruby ne le sait pas encore):

Initialisation

```
      def initialize(a,b)
            @a,@b = a,b
      end
```

Les nombres complexes a et b seront des propriétés du quaternion:

Propriétés a et b

```
      def a
            @a
      end

      def b
            @b
      end
```

Désormais on accède aux deux complexes a et b d'un quaternion q par $q.a$ et $q.b$.

Affichage

Pour afficher un quaternion *q* avec *puts(q)*, il est nécessaire de redéfinir (une sorte de **surcharge**) sa méthode de conversion en chaîne de caractères (*string*):

```
      def to_s

'('+a.real.to_s+')+('+a.imag.to_s+')i+('+b.real.to_s+')j+('+b.imag.to_s+')k'
      end
```

La notation des points se lit de droite à gauche, par exemple *a.real* veut dire *la partie réelle de a* et *q.a.real*, *la partie réelle du a de q*.

Le quaternion de Ruby ne possède alors que deux propriétés, *a* et *b*, mais on va se rattraper sur les méthodes, qui opèrent sur un quaternion (ou deux):

Fonctions

Module

Le module d'un quaternion est un réel:

```
      def abs
            Math.hypot(@a.abs,@b.abs)
      end
```

Conjugué

Le conjugué d'un quaternion est un quaternion de même module que celui-ci:

```
      def conj
            Quaternion.new(@a.conj,-@b)
      end
```

Opérations

Addition

Pour additionner deux quaternions, on additionne leurs *a* respectifs, et leurs *b* respectifs, et on crée un nouveau quaternion à partir des deux nombres complexes obtenus:

```
      def +(q)
            Quaternion.new(@a+q.a,@b+q.b)
      end
```

Pour calculer et afficher la somme des quaternions *p* et *q*, il suffit alors d'entrer *puts(p+q)*.

Soustraction

La soustraction des quaternions relève d'un principe analogue:

```
      def -(q)
            Quaternion.new(@a-q.a,@b-q.b)
      end
```

Multiplication

Le produit de deux quaternions est plus difficile à définir:

```
      def *(q)
            Quaternion.new(@a*q.a-@b*q.b.conj,@a*q.b+@b*q.a.conj)
      end
```

La multiplication des quaternions n'est pas commutative, comme le montre l'exemple suivant:

```
p=Quaternion.new(Complex(2,1),Complex(3,4))
q=Quaternion.new(Complex(2,5),Complex(-3,-5))
puts(p*q)
puts(q*p)
```

Division

Pour diviser un quaternion par un autre, on peut faire ainsi:

```
      def /(q)
            d=q.abs**2

Quaternion.new((@a*q.a.conj+@b*q.b.conj)/d,(-@a*q.b+@b*q.a)/d)
      end
```

Comme ils ont le même module, le quotient d'un quaternion par son conjugué est égal à 1:

```
p=Quaternion.new(Complex(2,1),Complex(3,4))

puts((p/p.conj).abs)
```

Cet exemple révèle que $\left(-\frac{22}{30}\right)^2+\left(\frac{4}{30}\right)^2+\left(\frac{12}{30}\right)^2+\left(\frac{16}{30}\right)^2=1$, c'est-à-dire que $22^2+4^2+12^2+16^2=484+16+144+256=900=30^2$, qui est une décomposition de 30^2 comme somme de 4 carrés.

Résumé

La classe Quaternion de Ruby tient en entier dans un fichier plutôt léger, au vu de ses possibilités:

```
require 'cmath'

class Quaternion

      def initialize(a,b)
            @a,@b = a,b
      end
```

```
      def a
            @a
      end

      def b
            @b
      end

      def to_s

'('+a.real.to_s+')+('+a.imag.to_s+')i+('+b.real.to_s+')j+('+b.imag.to_s+')k'
      end

      def +(q)
            Quaternion.new(@a+q.a,@b+q.b)
      end

      def -(q)
            Quaternion.new(@a-q.a,@b-q.b)
      end

      def *(q)
            Quaternion.new(@a*q.a-@b*q.b.conj,@a*q.b+@b*q.a.conj)
      end

      def abs
            Math.hypot(@a.abs,@b.abs)
      end

      def conj
            Quaternion.new(@a.conj,-@b)
      end

      def /(q)
            d=q.abs**2

Quaternion.new((@a*q.a.conj+@b*q.b.conj)/d,(-@a*q.b+@b*q.a.conj)/d)
      end

end
```

Si on enregistre ce fichier sous le nom *quaternions.rb*, il suffit d'insérer *require 'quaternions'* pour être en mesure d'effectuer des calculs sur les quaternions.

Octonions

Ce qui est intéressant avec la construction de Cayley-Dickson utilisée ci-dessus pour les quaternions, c'est qu'elle se généralise: En définissant une structure (un objet) comprenant deux quaternions *a* et *b*, on définit un octonion.

Définition et affichage

Définition

Comme pour les quaternions, on décrit l'objet *octonion* dans une classe *Octonion*:

```
class Octonion

      def initialize(a,b)
            @a,@b = a,b
      end

      def a
            @a
      end

      def b
            @b
      end
```

Au passage on définit les propriétés *a* et *b* de l'octonion comme celles du quaternion, sauf que cette fois-ci ce ne sont plus des complexes mais des quaternions. Mais comme Ruby est faiblement typé, cette particularité n'apparaîtra que lorsque *a* ou *b* sera utilisé.

Affichage

Là encore, la méthode *to_s* se définit comme celle des quaternions, mais il y a 8 nombres à afficher au lieu de 4:

```
    def to_s

'('+a.a.real.to_s+')+('+a.a.imag.to_s+')i+('+a.b.real.to_s+')j+('+a.b.imag.to_s+')k+('+b.a.real.to_s+')l+('+b.a.imag.to_s+')li+('+b.b.real.to_s+')lj+('+b.b.imag.to_s+')lk'

    end
```

Pour accéder au premier de ces nombres, que est la partie réelle du *a* de *a*, on note *a.a.real*. Autrement dit, on parcourt un arbre binaire, de profondeur 3.

Fonctions

Les fonctions sur les octonions se définissent presque comme celles sur les quaternions, Cayley-Dickson oblige:

Module

Comme pour les quaternions:

```
      def abs
            Math.hypot(@a.abs,@b.abs)
      end
```

Conjugué

```
      def conj
            Octonion.new(@a.conj,Quaternion.new(0,0)-@b)
      end
```

Opérations

Addition

Comme pour les quaternions, on additionne les octonions composante par composante (*a* avec *o.a*, *b* avec *o.b*):

```
      def +(o)
            Octonion.new(@a+o.a,@b+o.b)
      end
```

Soustraction

```
      def -(o)
            Octonion.new(@a-o.a,@b-o.b)
      end
```

Multiplication

```
      def *(o)
            Octonion.new(@a*o.a-o.b*@b.conj,@a.conj*o.b+o.a*@b)
      end
```

Non seulement la multiplication des octonions n'est pas commutative, elle n'est plus associative non plus:

```
m=Octonion.new(p,q)
n=Octonion.new(q,p)
o=Octonion.new(p,p)
puts((m*n)*o)
puts(m*(n*o))
```

Division

```
     def /(o)
          d=1/o.abs**2

Octonion.new((@a*o.a.conj+o.b*@b.conj)*Quaternion.new(d,0),(Quaternion.new(0,0)-@a.conj*o.b+o.a.conj*@b)*Quaternion.new(d,0))
     end
```

Là encore, le quotient d'un octonion par son conjugué est de module 1:

```
puts(m/m.conj)
puts((m/m.conj).abs)
```

Résumé

L'objet *Octonion* de Ruby est lui aussi, assez léger:

```
class Octonion



      def initialize(a,b)

            @a,@b = a,b

      end



      def a

            @a

      end



      def b

            @b

      end



      def to_s


'('+a.a.real.to_s+')+('+a.a.imag.to_s+')i+('+a.b.real.to_s+')j+('+a.b.imag.to_s+')k+('+b.a.real.to_s+')l+('+b.a.imag.to_s+')li+('+b.b.real.to_s+')lj+('+b.b.imag.to_s+')lk'

      end



      def +(o)

            Octonion.new(@a+o.a,@b+o.b)

      end



      def -(o)

            Octonion.new(@a-o.a,@b-o.b)

      end



      def *(o)

            Octonion.new(@a*o.a-o.b*@b.conj,@a.conj*o.b+o.a*@b)

      end



      def abs

            Math.hypot(@a.abs,@b.abs)

      end



      def conj

            Octonion.new(@a.conj,Quaternion.new(0,0)-@b)

      end



      def /(o)

            d=1/o.abs**2


Octonion.new((@a*o.a.conj+o.b*@b.conj)*Quaternion.new(d,0),(Quaternion.new(0,0)-@a.conj*o.b+o.a.conj*@b)*Quaternion.new(d,0))

      end
```

```
end
```

En l'enregistrant sous le nom *octonions.rb*, il suffit d'écrire

```
require 'octonions'
```

pour être en mesure d'effectuer des calculs sur les octonions en Ruby.

Bibliographie

- En fait, les quaternions existent déjà sous Ruby, à condition de les télécharger: [1]; sur le même site, l'auteur propose aussi des octonions.
- Sur les octonions, le livre de John Baez est une lecture hautement conseillée: [3]

Références

[1] http://www.math.kobe-u.ac.jp/~kodama/tips-ruby-quaternion.html

Ensembles en Ruby

Les évènements sont décrits en probabilité par des ensembles. Si ces ensembles sont finis, *Ruby* les gère.

Construction d'évènements

Évènements certain et impossible

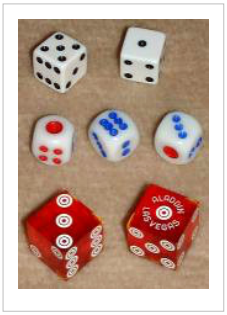
L'évènement impossible, noté $\emptyset$, est entré en *Ruby* avec des crochets vides:

```
impossible=[]
puts(impossible.length())
puts(impossible.empty?)
```

L'ensemble de toutes les issues possibles de l'expérience aléatoire, appelé *univers*, est noté Ω.

Pour savoir si un élément se trouve dans un ensemble, on utilise *include?* comme dans *omega.include?(6)* pour savoir si l'évènement peut donner un 6.

Avec un dé

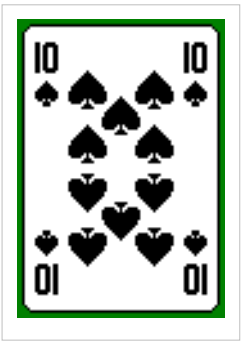


On s'apprête à lancer un dé. Alors l'évènement "le résultat sera plus petit que 5" est décrit par l'ensemble $\{1,2,3,4\}$. De même, l'évènement "le résultat sera pair" est représenté par $\{2,4,6\}$. On construit aisément ces évènements avec la notation ensembliste de *Ruby* qui se fait avec des crochets au lieu des accolades. Mais la définition d'ensembles par description est possible avec *Ruby*:

```
univers=(1..6).to_a
petit=univers.select { |r| r<5 }
pair=univers.select { |r| r%2==0 }
impossible=[]
```

Pour construire l'univers, on peut prend la liste des nombres allant de 1 à 6, et on la transforme en tableau avec *to_a*. Pour avoir les *petits* résultats (moins que 5), on choisit dans l'univers les éléments qui sont inférieurs à 5. *select* est une méthode de l'objet *univers*, qui se crée une variable *r* (entre traits verticaux) et lui fait parcourir les éléments du tableau (car *univers* en est un) et lui fait passer ou non par un filtre. En bref, on *sélectionne* les éléments de l'univers qui sont inférieurs à 5. De même, pour avoir les résultats pairs, on sélectionne les éléments de l'univers dont le quotient par 2 tombe juste (reste nul).

Avec des cartes



Cette fois-ci, on extrait au hasard une carte parmi un jeu de 32 cartes.

On construit l'univers par un *produit cartésien* (des cartes avec Descartes !) entre l'ensemble des valeurs et celui des couleurs:

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
couleurs=['carreau','coeur','pique','trèfle']
valeurs=valeurs.collect { |v| v.to_s }
univers=[]
valeurs.collect { |v| couleurs.collect { |c| univers.push(v+' '+c) }}
puts(univers)
```

La troisième ligne transforme toutes les cartes en chaînes de caractères; en effet certaines d'entre elles étaient des chiffres. La suite du script consiste à créer un univers initialement vide, puis, avec la méthode *collect* du tableau des valeurs de cartes, à placer dans le jeu de cartes, l'une après l'autre, toutes les cartes (il est nécessaire de mettre une boucle à l'intérieur de la première, pour les différentes couleurs associées à chaque valeur de carte).

L'évènement "la carte est une figure" (pas un nombre) se construit en choisissant les cartes dont le début du nom n'est pas un nombre entier (donc se transforme en 0 lorsqu'on le convertit en entier):

```
figure=univers.select { |carte| carte[0..1].to_i==0 }
puts(figure)
```

Et pour construire l'évènement "la carte est un pique", on extrait les cartes de pique du jeu entier:

```
pique=univers.select { |carte| carte[-2..-1]=='ue'}
puts(pique)
```

(on extrait les cartes dont le nom termine par *ue*, puisque seul le mot *pique* se termine ainsi).

Évènements simultanés

Notation

L'évènement "A et B" se note $A \cap B$, et l'opération se note en *Ruby* par une esperluette (*&*).

Avec le dé

```
petit=[1,2,3,4]
pair=[2,4,6]
puts(petit&pair)
```

Avec les cartes

Le script suivant montre que dans un jeu de 32 cartes, il y en a 3 qui sont à la fois des figures et des piques: Les trois figures de pique:

```
puts(figure&pique)
```

Le "ou" inclusif

Notation

De même l'évènement "A ou B" se note $A \cup B$, et en *Ruby*, le symbole *pipe* (trait vertical). *Ruby* enlève automatiquement les doublons.

Avec le dé

```
petit=[1,2,3,4]
pair=[2,4,6]
puts(petit|pair)
```

Avec les cartes

On peut compter les cartes qui sont des figures ou des piques:

```
puts(figure|pique)
puts((figure|pique).size)
```

...mais on peut aussi laisser *Ruby* les compter, il en trouve 17. Ce comptage est à la base des calculs de probabilité.

Contraire

Pour calculer le contraire d'un évènement, on le soustrait à l'univers.

Avec le dé

```
univers=[1,2,3,4,5,6]
petit=[1,2,3,4]
pair=[2,4,6]
puts(univers-petit)
puts(univers-pair)
```

Ce qui montre que le contraire de "pair" est "impair".

Avec les cartes

```
puts(univers-figure)
puts(univers-pique)
```

Probabilités

La probabilité d'un évènement est définie comme le quotient de sa taille (en *Ruby*, *size*) par celle de l'univers.

On peut associer une probabilité à un évènement seul, en baptisant l'univers *$univers* ce qui fait qu'il est une variable globale, et en définissant une probabilité par

```
require 'mathn'
def proba(e)
    return Rational(e.size,$univers.size)
end
```

Mais pour éviter l'usage d'une variable globale, on peut aussi associer une probabilité à l'évènement et à son univers:

Avec le dé

```
require 'mathn'
univers=[1,2,3,4,5,6]
petit=[1,2,3,4]
pair=[2,4,6]
puts(Rational(petit.size,univers.size))
puts(Rational(pair.size,univers.size))


def proba(e,u)
    return Rational(e.size,u.size)
end

p1=proba(petit,univers)+proba(pair,univers)-proba(petit&pair,univers)
puts(p1)
p2=proba(petit|pair,univers)
puts(p1==p2)
```

Avec les cartes

```
require 'mathn'

puts(Rational(figure.size,univers.size))
puts(Rational(pique.size,univers.size))


def proba(e,u)
    return Rational(e.size,u.size)
end

p1=proba(figure,univers)+proba(pique,univers)-proba(figure&pique,univers)
puts(p1)
p2=proba(figure|pique,univers)
puts(p1==p2)
```

Probabilités conditionnelles

Ci-dessus, on a défini les probabilités avec comme paramètre l'univers. En effet, cette variable est globale, donc inaccessible *a priori* dans le corps de la fonction. Ceci permet de remplacer l'univers par un autre évènement, et donc de définir la probabilité conditionnelle. Par exemple, avec le dé:

```
require 'mathn'
univers=[1,2,3,4,5,6]
petit=[1,2,3,4]
pair=[2,4,6]

def proba(e,u)
    return Rational(e.size,u.size)
end

p1=proba(petit&pair,petit)
puts(p1)
p2=proba(petit&pair,pair)
puts(p1==p2)
```

Définitions

On peut alors définir des booléens concernant des évènements, en utilisant les propriétés de leurs probabilités:

```
require 'mathn'

def incompatibles(a,b)
    return proba(a&b)==0
end


def indépendants(a,b)
    return proba(a&b)==proba(a)*proba(b)
```

```
end
```

Nombres pseudo-aléatoires en Ruby

Nombres pseudo-aléatoires

Le traditionnel nombre pseudo-aléatoire compris entre 0 et 1 s'obtient avec

```
puts(rand)
```

Si on veut un nombre entier aléatoire, on peut mettre le nombre d'occurences possibles entre parenthèses après le *rand*. Par exemple, pour lancer un dé à 6 faces, on obtient avec *rand(6)* un nombre entre 0 et 5. Donc pour lancer un dé, on fait

```
puts(rand(6).succ)
```

Pour simuler une variable aléatoire binomiale de paramètres *n* et *p*, on peut utiliser le fait que celle-ci est la somme de *n* variables de Bernoulli indépendantes entre elles. Façon *Ruby*, cela peut se faire avec l'algorithme suivant (basé sur une analogie avec un jeu de pile ou face, où *p* est la probabilité d'avoir *pile* et *n* le nombre de lancers de la pièce):

1. On crée un ensemble de *n* objets, par exemple une liste de nombres *(1..n)*;
2. On extrait de celle-ci les nombres victorieux (ceux pour lesquels une pièce est tombée sur *pile*);
3. On compte les objets retenus.

En *Ruby* cela donne

```
def binomial(n,p)
    return ((1..n).select { |i| rand<p }).size
end


10.times do puts(binomial(8,0.2)) end
```

Lancer de dés

Un dé

Pour voir si le dé est équilibré, on peut le lancer quelques milliers de fois et compter combien de fois chaque face est sortie... ou laisser faire le travail par *Ruby*:

```
effectifs=[0]*6
n=6000
n.times do effectifs[rand(6)]+=1 end
puts(effectifs)
```

Deux dés

Pour lancer deux dés et additionner leurs résultats, on fait comme ci-dessus et on additionne. Seulement le tableau des effectifs est indexé de 0 à 10 (2 de moins que les résultats des lancers):

```
effectifs=[0]*11
n=6000
n.times do effectifs[rand(6)+rand(6)]+=1 end
puts(effectifs)
```

Avec des cartes

Tirer une carte au hasard

Puisque le jeu de cartes est un tableau, il suffit de choisir un indice au hasard pour tirer une carte au hasard. Par prudence, on va faire semblant de ne pas savoir qu'il y a 32 cartes dans le jeu (et qu'elles sont numérotées de 0 à 31). On va tirer 3200 fois une carte d'un jeu de 32 et compter le nombre de fois qu'on a eu un as de pique. Pour cela on va utiliser un compteur du nombre de victoires (*gains*) et on va lui injecter une unité chaque fois que le nom de la carte est *1 pique*:

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
valeurs=valeurs.collect { |v| v.to_s}
couleurs=['carreau','coeur','pique','trèfle']
univers=[]
valeurs.collect{|v| couleurs.collect{|c| univers.push(v+' '+c)}}

n=3200
gains=(1..n).inject {|g,i| g+(univers[rand(univers.size)]=='1 pique' ?
1 : 0) }
puts(gains.to_f/n)
```

Tirer 5 cartes au hasard

Pour constituer une main de 5 cartes, il suffit *a priori* de faire 5 fois l'opération précédente:

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
valeurs=valeurs.collect { |v| v.to_s}
couleurs=['carreau','coeur','pique','trèfle']
univers=[]
valeurs.collect{|v| couleurs.collect{|c| univers.push(v+' '+c)}}

hand=[]
((1..5).to_a).collect{hand.push(univers[rand(univers.size)])}
puts(hand)
```

Seulement il peut arriver qu'on ait deux fois l'as de pique dans la même main ! En effet le script précédent réalise un *tirage avec remise*, pour lequel les calculs de probabilités sont plus faciles, mais irréaliste pour les jeux de cartes et le Loto. Ce dont on a besoin dans le cas présent, c'est d'un *tirage sans remise*, qui se produira en enlevant les cartes au fur et à mesure qu'on les met dans la main de 5 cartes.

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
valeurs=valeurs.collect { |v| v.to_s}
couleurs=['carreau','coeur','pique','trèfle']
univers=[]
valeurs.collect{|v| couleurs.collect{|c| univers.push(v+' '+c)}}

hand=[]
while hand.size<5
    jeu=univers-hand
    carte=jeu[rand(jeu.size)]
    hand.push(carte)
end

puts(hand)
```

Le jeu dont on a extrait les cartes est une variable locale, et à la fin de ce script il y a toujours 32 cartes dans l'univers.

Ensuite on peut compter les carrés d'as parmi 10 000 parties:

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
valeurs=valeurs.collect { |v| v.to_s}
couleurs=['carreau','coeur','pique','trèfle']
univers=[]
valeurs.collect{|v| couleurs.collect{|c| univers.push(v+' '+c)}}

carres=0
10000.times do
    hand=[]
    while hand.size<5
        jeu=univers-hand
        carte=jeu[rand(jeu.size)]
```

```
        hand.push(carte)
    end
    if (hand.select {|c| c[0..1]=='1 '}).size==4
        carres+=1
    end
end

puts(carres.to_f/10000)
```

On construit une liste avec les cartes de *hand* dont le nom commence par un *1* suivi d'un espace (les as), et lorsque la longueur de cette liste est égale à 4, on incrémente le compteur *carres*. À la fin de l'exécution du script, on a donc le nombre de carrés d'as parmi les 10 000 tirages, et en le divisant par 10 000, on a la fréquence de ces carrés d'as. Elle est très faible !

Mélanger un jeu de cartes

En 1741, dans les *Mémoires de l'Académie des Sciences de Berlin*, Leonhard Euler publiait un article titré *Calcul de la probabilité du jeu de rencontre*. Le nom initial du jeu de rencontre était *jeu de treize* parce qu'il se jouait à 13 cartes. Mais Euler généralise le jeu à *n* cartes.

Il le définit ainsi:

Le jeu de rencontre est un jeu de hasard, où deux personnes ayant chacune un entier jeu de cartes, en tirent à la fois une carte après l'autre, jusqu'à ce qu'il arrive, qu'elles rencontrent la même carte: et alors l'une des deux personnes gagne. Or, lorsqu'une telle rencontre n'arrive point du tout, alors c'est l'autre des deux personnes qui gagne. Cela posé, on demande la probabilité, que l'une et l'autre de ces deux personnes aura de gagner.

Dans cet excellent article (16 pages en Français),

Euler montre que

Pourvu donc que le nombre de cartes ne soit pas moindre que 12, l'espérance de A sera toujours à celle de B à peu près comme 12 à 7 ... Ou bien parmi 19 jeux qu'on joue, il y en aura probablement 12 qui font gagner A, et 7 qui feront gagner B.

Pour mélanger un jeu de cartes, on peut construire une main de 32 cartes ! Ensuite on peut répéter l'expérience 1900 fois, et compter combien de fois il y a eu au moins une rencontre (le nombre de rencontres strictement positif), et enfin, par division par 1900, estimer la fréquence de ces rencontres, et la comparer avec le quotient de 12 par 19 donné par Euler:

```
valeurs=[1,7,8,9,10,'Valet','Dame','Roi']
valeurs=valeurs.collect { |v| v.to_s}
couleurs=['carreau','coeur','pique','trèfle']
univers=[]
valeurs.collect{|v| couleurs.collect{|c| univers.push(v+' '+c)}}

gains=0
1900.times do
    hand=[]
    while hand.size<32
        jeu=univers-hand
        carte=jeu[rand(jeu.size)]
        hand.push(carte)
    end
```

```
    rencontres=((0..31).to_a).select { |i| hand[i]==univers[i] }
    if rencontres.size>0
        gains+=1
    end
end

puts(gains.to_f/1900)
puts(12.0/19)
```

Un exemple de résultat sur 1900 jeux:

```
0.634736842105263
```

0.631578947368421

La valeur exacte de la probabilité d'une rencontre est donnée par *Euler*, dont la formule se traduit en *Ruby* par

```
a=(1..32).inject {|p,n| p+(-1)**n/(1..n).inject(1) {|f,k| f*k}}
puts(1-1/a)
```

et qui donne la valeur $\frac{312259436277874488743607770353539 1}{116106858767688587637979490635353 91}$...

Méthode de Monte-Carlo

Pour calculer la valeur approchée de π par la méthode de Monte-Carlo, on crée un nuage de points à coordonnées pseudo-aléatoires, et on compte combien d'entre eux sont à moins d'une unité de distance de l'origine du repère:

```
points=(1..1000000).inject{|p,n| p+(Math.hypot(rand,rand)<1? 1 : 0) }
puts(points.to_f/1000000*4)
```

Le résultat est correct à trois décimales près.

Suites en Ruby

Une suite de nombres (éventuellement complexes) ne peut se représenter en machine parce qu'elle comprend une infinité de termes. Alors on n'en représente qu'une partie sous forme de liste de nombres. Et *Ruby* manipule très bien ce genre d'objets.

Définition de suites

Par fonction

Une suite est une fonction de $\mathbb{N}$ dans $\mathbb{R}$ (ou $\mathbb{C}$ ...). On peut donc facilement calculer les premiers termes de celle-ci en utilisant la méthode *collect* d'une liste d'entiers (approximation finie de $\mathbb{N}$). Par exemple pour vérifier que la suite $u_n = \frac{1}{n}$ tend vers 0, on peut essayer

```
(1..50).collect{|n| puts(1/n.to_f)}
```

Suites récurrentes

Pour une suite récurrente, chaque terme est défini à partir du précédent.

Suite logistique

La suite logistique $u_{n+1} = 4u_n\left(1 - u_n\right)$ est chaotique sur *[0;1]*. Pour le vérifier, on peut faire

```
u=0.1
50.times do
    u=4*u*(1-u)
    puts(u)
end
```

En constatant que $u_0 = 0,1 = \frac{1}{10} \in \mathbb{Q}$, on peut vérifier que, quoique chaotique, cette suite est formée de fractions:

```
require 'mathn'
u=1/10
10.times do
    u=4*u*(1-u)
    puts(u)
end
```

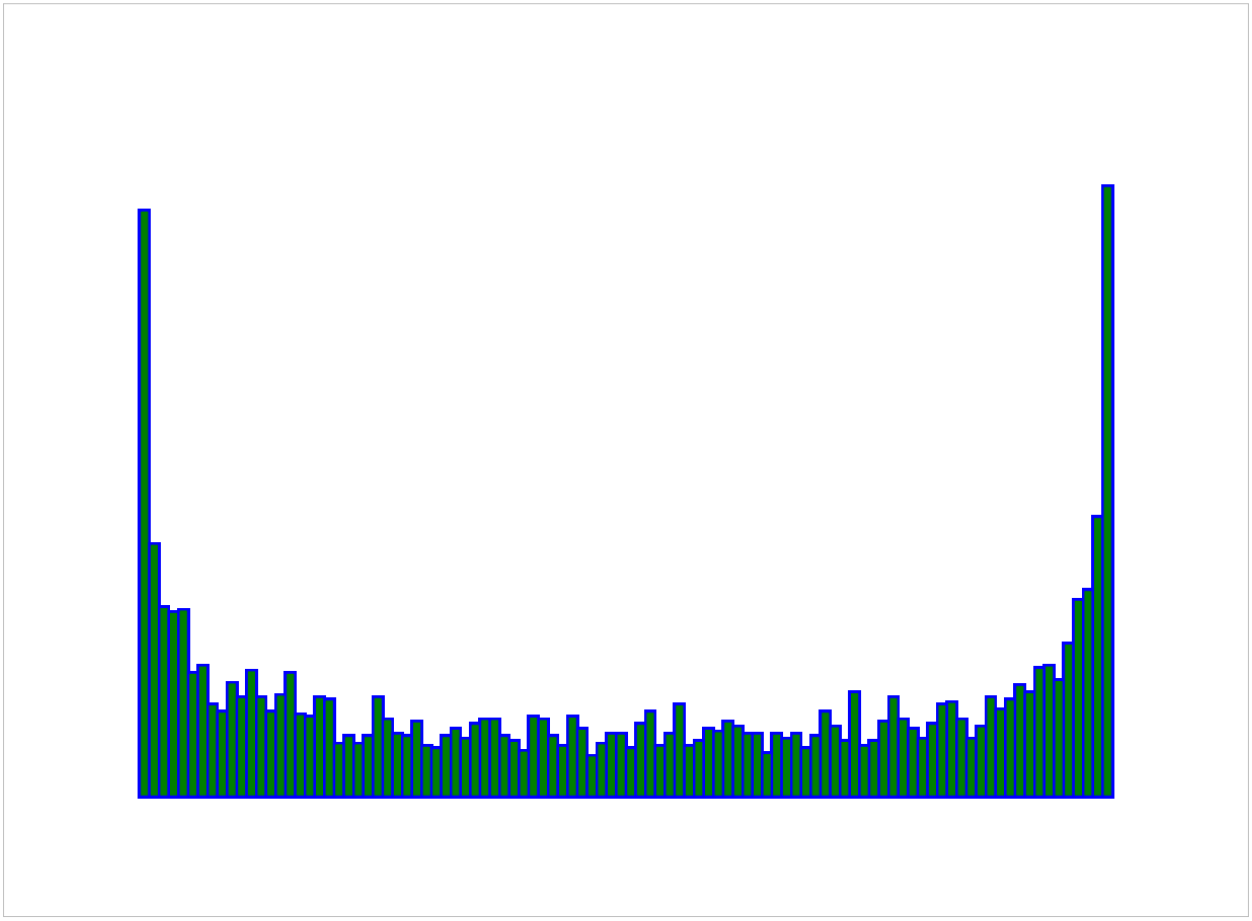
Quoique chaotique, cette suite ne fait pas un bon générateur pseudo-aléatoire, parce que les nombres proches de 0 et 1 sont trop souvent visités. Pour le vérifier graphiquement, on peut dessiner un histogramme des 4000 premières valeurs de la suite, avec l'algorithme suivant:

1. On fait comme ci-dessus, mais au lieu d'afficher *u*, on incrémente l'entrée d'un tableau des effectifs indexée par sa troncature. C'est ce tableau qui va être représenté graphiquement.
2. Ensuite, on représente chaque effectif par un rectangle de largeur 4 pixels et de hauteur l'effectif correspondant. Les rectangles sont bleus, remplis de vert.

Le tout est fait en écrivant les instructions dans le langage svg, engendrées par *Ruby*, dans un fichier *HistogramRuby1.svg* visible ci-dessous. Voici le script au complet:

```
figure=File.open("HistogramRuby1.svg","w")

figure.puts('<?xml version="1.0" encoding="utf-8"?>')

figure.puts('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"')

figure.puts('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">')

figure.puts('<svg xmlns="http://www.w3.org/2000/svg" width="500" height="360">')



effectifs=[0]*100

u=0.1



4000.times do

    u=4*u*(1-u)

    effectifs[(100*u).to_i]+=1

end



(0..99).collect{|n| figure.puts('<rect x="'+(4*n+50).to_s+'" y="'+(320-effectifs[n]).to_s+'" width="4" height="'+effectifs[n].to_s+'" fill="green" stroke="blue" stroke-width="1" />')}



figure.puts('</svg>')

figure.close
```

Et voici le fichier produit par le script:



Suites arithmétiques et géométriques

Les suites arithmétiques et géométriques sont aussi des suites récurrentes.

Suites arithmétiques

Une suite est arithmétique de raison *r* si $\forall n, u_{n+1} = u_n + r$. Cette définition est récurrente.

Par exemple, si on place 2000 € avec des intérêts (simples) correspondant à 3 % du capital de départ, soit 60 €, on peut calculer les valeurs successives du capital pendant 20 ans avec

```
capital=2000.00
interet=capital*3/100.0
20.times do
    capital+=interet
    puts(capital)
end
```

Suites géométriques

Une suite est géométrique de raison *r* si $\forall n, u_{n+1} = u_n \times r$. Les suites géométriques sont donc aussi récurrentes.

Si on place 2000 € à intérêts composés au taux de 2 % par an, l'affichage des valeurs successives du capital (arrondies à l'eurocent près) peut se faire avec

```
capital=2000.00
20.times do
    capital*=1.02
    puts(capital.round(2))
end
```

Sachant que chaque humain a deux parents et que chacun d'entre eux a aussi deux parents, etc. on peut dire que le nombre d'ancêtres à la génération *n* est géométrique de raison 2. Le nombre total d'ancêtres jusqu'à la génération *n* s'obtient par

```
(0..20).inject{|ancetres,generation| ancetres+=2**generation}

puts(ancetres)
puts(2**21)
```

On a presque autant d'ancêtres à la génération 21 qu'à toutes les générations précédentes cumulées!

Ce qui donne envie de vérifier si c'est pareil pour toutes les générations ou si c'est une spécificité de la génération 21:

```
genealogie=[1]*20
(1..20).collect{|i| genealogie[i]=(0..i).inject{|a,g| a+=2**g }}
generations=[1]*20
(1..20).collect{|i| generations[i]=2**i}
test=(1..20).reject{|i| generations[i]==genealogie[i-1]+2}
puts(test.size)
```

On crée une liste des ancêtres jusqu'à la génération *n* comprise (*genealogie*) et une liste (*generations*) des nombres d'ancêtres à la génération *n* seulement. La lise *test* est constituée des entiers *n* pour lesquels le nombre d'ancêtres à la génération *n* n'est **pas** égal au total d'ancêtres jusqu'à la génération *n-1* augmenté de 2 (avec *reject*, on a enlevé les positifs). La longueur de ce *test* est très petite !

Sous *Ruby* on peut aussi calculer des suites géométriques de raison complexe. La somme des termes est alors particulièrement intéressante à étudier (par exemple si la raison vaut *i*).

Suites d'entiers

Suite de Fibonacci

Calcul des termes

La récurrence de la suite de Fibonacci est double, avec $u_{n+1} = u_n + u_{n-1}$. Son calcul pose donc un problème algorithmique, puisqu'il faut trois variables (les deux termes à calculer et une variable *tampon* pour stocker temporairement l'un des deux termes, afin qu'il ne soit pas écrasé par la somme). Ce problème n'existe pas en *Ruby* qui permet les affectations simultanées:

```
a=1
b=1
for n in 1..20 do
    a,b=b,a+b
    puts(b)
end
```

On peut aussi le faire de manière plus *Ruby*:

```
a=1
b=1
(1..20).collect{
    a,b=b,a+b
    puts(b)
}
```

Nombre d'Or

Pour étudier le quotient de deux termes successifs de la suite:

```
a,b=1,1
(1..20).collect{
    a,b=b,a+b
    puts(b.to_f/a.to_f)
}

puts((5**0.5+1)/2)
```

Mais en fait, les nombres de Fibonacci étant entiers, leurs quotients sont des fractions, et cette variante le montre:

```
require 'mathn'

a,b=1,1
(1..20).collect{
    a,b=b,a+b
    puts(b/a)
}
```

On a donc une suite d'approximations rationnelles du nombre d'Or.

Suite de Collatz

Algorithmiquement, la suite de Collatz est intéressante parce que son calcul est basé sur un test de parité, et qu'elle utilise une boucle à condition de sortie:

```
def Collatz(x)
    if x%2==0
        return x/2
    else
        return 3*x+1
    end
end

u=65
while(u>1) do
    u=Collatz(u)
    puts(u)
end
```

Multiples communs

La suite des multiples de 5 et la suite des multiples de 7 sont arithmétiques de raisons respectives 5 et 7. On peut les construire en choisissant les nombres entiers qui sont divisibles respectivement par 5 et par 7:

```
a5=(1..1000).select{|n| n%5==0}
a7=(1..1000).select{|n| n%7==0}
puts(a5&a7)
```

Les multiples communs à 5 et 7 sont les multiples de 35, qui est le ppcm de 5 et 7. Cette construction est à l'origine de la théorie des idéaux par Kummer.

Suites et séries

Une série est une suite dont le terme général est défini par une somme.

Premier exemple

La suite définie par $u_n = \frac{1}{2}+\frac{1}{6}+\frac{1}{12}+\frac{1}{20}+... = \frac{1}{1\times 2}+\frac{1}{2\times 3}+\frac{1}{3\times 4}+\frac{1}{4\times 5}+...+\frac{1}{n(n+1)} = \sum_{k=1}^{n}\frac{1}{k(k+1)}$ tend vers 1, il est relativement aisé de le démontrer, et encore plus facile de le vérifier avec *Ruby*:

```
suite=(1..50).collect{|n|
    (1..n).inject(0){|somme,k|
        somme+=1.0/(k*k.succ)
    }
}

puts(suite)
```

Cette suite est une suite de rationnels:

```
require 'mathn'

suite=(1..20).collect{|n|
    (1..n).inject(0){|somme,k|
        somme+=1/(k*k.succ)
    }
}

puts(suite)
```

Cette variante suggère d'ailleurs une démonstration de la convergence, grâce à l'émission d'une conjecture sur le terme général de la suite...

Deuxième exemple

La suite $u_n = \frac{n}{n^2+1} + \frac{n}{n^2+2} + \frac{n}{n^2+3} + \frac{n}{n^2+4} + ... + \frac{n}{n^2+n} = \sum_{k=1}^{n} \frac{n}{n^2+k}$ converge aussi, bien que ce ne soit pas évident en voyant son expression algébrique.

```
suite=(1..20).collect{|n|
    (1..n).inject(0){|somme,k|
        somme+=n.to_f/(n**2+k)
    }
}

puts(suite)
```

Là encore, la suite est rationnelle:

```
require 'mathn'

suite=(1..20).collect{|n|
    (1..n).inject(0){|somme,k|
        somme+=n/(n**2+k)
    }
}

puts(suite)
```

```
```

Constante d'Euler

On peut la calculer (et vérifier la lenteur de la convergence) avec

```
suite=(1..50).collect{|n|
    (1..n).inject(0){|s,k| s+=1.0/k}-Math.log(n)
}

puts(suite)
```

Applications

Méthode de Heron

Pour calculer $\sqrt{5}$avec la méthode de Heron, on utilise la suite itérée $u_{n+1}=\frac{u_n+\frac{5}{u_n}}{2}$:

```
u=1.0
50.times do
    u=(u+5.0/u)/2.0
    puts(u)
end
```

Mais encore une fois, cette suite qui converge vers $\sqrt{5}$est formée de fractions. On a donc une suite d'approximations rationnelles de $\sqrt{5}$:

```
require 'mathn'

u=1
50.times do
    u=(u+5/u)/2
    puts(u)
end
```

On en déduit des approximations rationnelles du nombre d'Or $\frac{\sqrt{5}+1}{2}$:

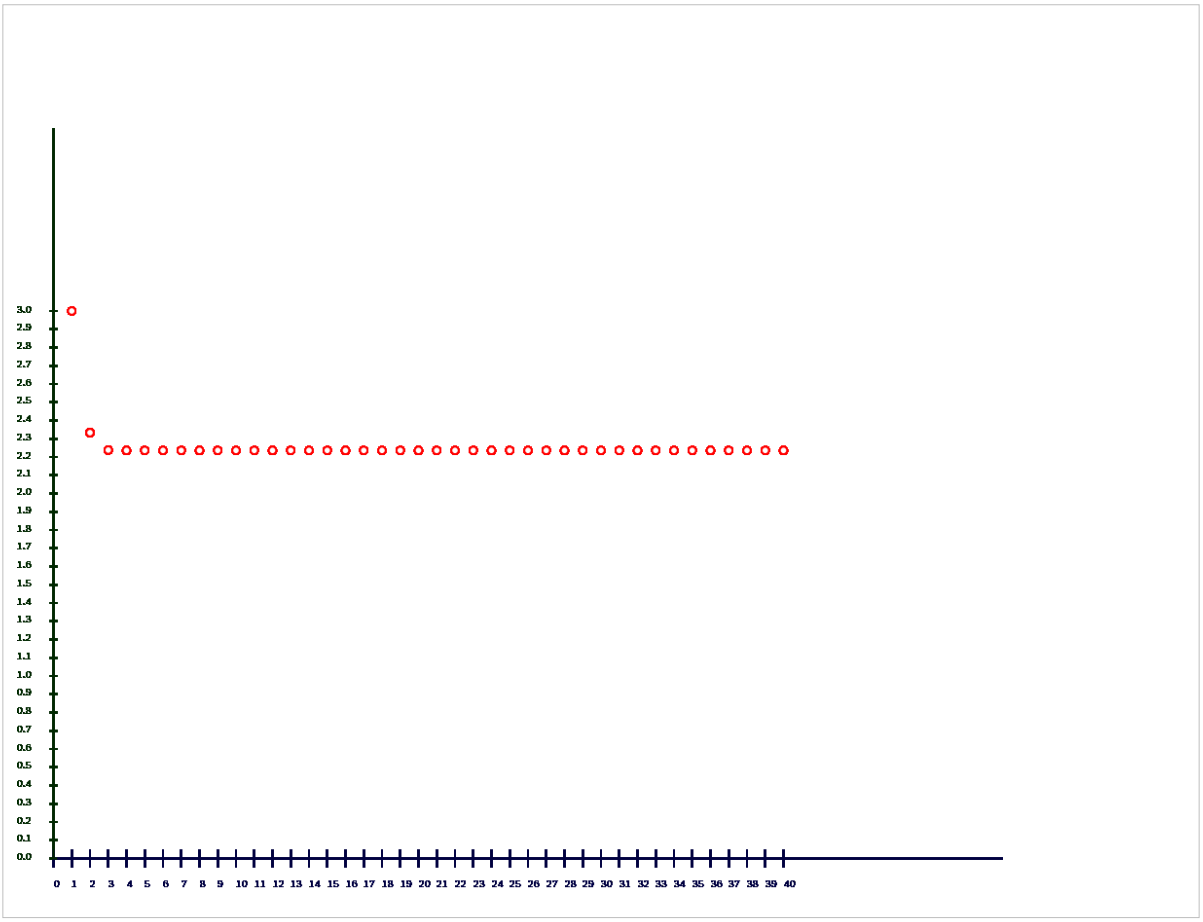
```
require 'mathn'

u=1
10.times do
    u=(u+5/u)/2
    puts((u+1)/2)
end
```

En comparant avec les quotients de nombres de Fibonacci successifs, on voit que la méthode de Heron converge beaucoup plus vite. Cette convergence peut se montrer en représentant graphiquement la suite, ce qu'on peut faire en plaçant des points dans un fichier *svg*:

```
figure=File.open("SuiteRuby01.svg","w")

figure.puts('<?xml version="1.0" encoding="utf-8"?>')

figure.puts('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"')
```

```
figure.puts('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">')

figure.puts('<svg xmlns="http://www.w3.org/2000/svg" width="640" height="480">')



figure.puts('<line x1="20.0" y1="460.0" x2="540.0" y2="460.0" style="stroke:rgb(0,0,64);stroke-width:1"/>')

((0..400).select {|x| x%10==0}).collect { |x|

figure.print('<text x="'+(x+20).to_s+'" y="475.0" style="font-size:6;fill:rgb(0,0,64);font-weight:normal">'+(x/10).to_s+'</text>\n'+'<line x1="'+(x+20).to_s+'" y1="455" x2="'+(x+20).to_s+'" y2="465" style="stroke:rgb(0,0,64);stroke-width:1"/>\n')}

figure.puts('<line x1="20.0" y1="460.0" x2="20.0" y2="60.0" style="stroke:rgb(0,40,0);stroke-width:1"/>')

((0..300).select {|x| x%10==0}).collect { |x|

figure.print('<text x="0" y="'+(460-x).to_s+'" style="font-size:6;fill:rgb(0,40,0);font-weight:normal">'+(x/100.0).to_s+'</text>\n'+'<line x1="18" y1="'+(460-x).to_s+'" x2="22" y2="'+(460-x).to_s+'" style="stroke:rgb(0,40,0);stroke-width:1"/>\n')}



u=1.0

n=0

40.times do

    n+=1

    u=(u+5.0/u)/2.0

    figure.puts('<circle cx="'+(20+10*n).to_s+'" cy="'+(460-100*u).to_s+'" r="2" fill="white" stroke="red" stroke-width="1" />')

end



figure.puts('</svg>')

figure.close
```

Le fichier produit par ce script s'appelle *SuiteRuby01.svg*. Le voici:



Formule de l'arc tangente

```
p=(1..100).collect{|n|
    4*(0..n).inject(0){|s,k|
        s+=(-1)**k/(2*k+1.0)
    }
}

puts(p)
```

Comme on le voit, la suite converge très lentement.

Encore une fois, les termes de la suite sont rationnels, ce qui donne une suite de fractions approchant π :

```
require 'mathn'

p=(1..10).collect{|n|
    4*(0..n).inject(0){|s,k|
        s+=(-1)**k/(2*k+1)
    }
}

puts(p)
```

Fonctions en Ruby

En *Ruby*, une fonction s'appelle une méthode, mais du point de vue mathématique, ce n'est guère qu'une question de vocabulaire.

Exemple

Sujet Bac STG CGRH Métropole-Réunion Septembre 2007

Extrait du sujet de l'exercice 3:

Une entreprise produit des appareils électroménagers. Le coût horaire de production de x appareils est donné en euros par :

$C(x) = x^2 + 50x + 100$ pour $5 \leqslant x \leqslant 40$

Et plus bas, dans le même exercice,

le coût moyen est défini par

Le coût moyen de production d'un objet est égal à $f(x) = \frac{C(x)}{x}$ pour x appartenant à [5 ; 40].

Définition des méthodes

Donc pour *Ruby*, *C* et *f* seront des méthodes, dont l'antécédent s'appellera *x* et dont l'image sera envoyée par *return*:

```
def C(x)
    return x**2+50*x+100.0
end



def f(x)
    return C(x)/x
end
```

Bien entendu, on pouvait aussi définir *f* directement par

```
def f(x)
    return x+50+100.0/x
end
```

Tableau de valeurs

Suite de l'énoncé du Bac STG 2007

Par la suite, on demande de reproduire et compléter le tableau suivant, arrondi au centième d'euro:

x	5	10	20	30	40
f(x)					

Tableau de valeurs

Certes, avec une boucle, on peut calculer plein de valeurs de *f(x)* différentes:

```
for x in 5..40 do
    puts("l'image de #{x} par f est #{f(x)}")
end
```

mais on a trop de valeurs de *x* pour remplir le tableau. Une meilleure variante sera donc

```
for x in [5,10,20,30,40] do
    puts("l'image de #{x} par f est #{f(x)}")
end
```

qui est déjà bien plus léger à exploiter.

Plus typiquement *Ruby*:

```
[5,10,20,30,40].collect {|x| puts(f(x))}
```

Image d'un ensemble par une fonction

Pour que la fonction *f* ne s'applique plus seulement à un réel, mais à tous les nombres du tableau, on utilise *map* (une méthode de l'objet *tableau*):

```
[5,10,20,30,40].map { |x| puts(f(x)) }
```

Cette notion d'image d'un ensemble par une fonction est à la base de pratiquement toute la géométrie: Alors que la symétrie centrale est définie comme une transformation qui associe un point à un point, on l'applique très vite à des ensembles de points comme dans l'expression *la symétrique d'une droite par rapport à un point*.

Arrondis

Enfin, pour avoir les arrondis au centime près:

```
[5,10,20,30,40].collect {|x| puts(f(x).round(2))}
```

qui remplit le tableau presque immédiatement.

Représentation graphique

Ruby n'étant pas très doué (pour le moment) en dessin, on va utiliser ses talents littéraires pour fabriquer un fichier au format svg. Ce fichier sera fabriqué comme une chaîne de caractères, et inscrit dans un fichier nommé *FonctionRuby01.svg* posté ci-dessous. Le fichier sera créé en mode écriture (*w*) avec

```
figure=File.open("FonctionRuby01.svg","w")
```

Création de la figure

On commence par écrire dans le fichier (qui, on l'a vu ci-dessus, s'appelle *figure*), ce qu'il faut pour dire que c'est un fichier *svg* (l'entête):

```
figure.puts('<?xml version="1.0" encoding="utf-8"?>')
figure.puts('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"')
figure.puts('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">')
figure.puts('<svg xmlns="http://www.w3.org/2000/svg" width="640" height="480">')
```

Le fichier fera donc 640 pixels de large et 480 pixels de haut.

Axes

Les axes et leurs graduations seront des traits, créés par des boucles puisque le travail des graduations est répétitif et que les boucles sont faites pour tout ce qui est répétitif.

Des abscisses

L'axe des abscisses et ses graduations sera bleu:

```
figure.puts('<line x1="20.0" y1="460.0" x2="540.0" y2="460.0" style="stroke:rgb(0,0,64);stroke-width:1"/>')

((0..500).select {|x| x%100==0}).collect { |x|

figure.print('<text x="'+(x+20).to_s+'" y="475.0" style="font-size:16;fill:rgb(0,0,64);font-weight:normal">'+(x/10).to_s+'</text>\n'+'<line x1="'+(x+20).to_s+'" y1="455" x2="'+(x+20).to_s+'" y2="465" style="stroke:rgb(0,0,64);stroke-width:1"/>\n')}

((0..500).select {|x| x%50==0}).collect { |x| figure.print('<line x1="'+(x+20).to_s+'" y1="456" x2="'+(x+20).to_s+'" y2="464" style="stroke:rgb(0,0,64);stroke-width:1"/>\n')}

((0..500).select {|x| x%10==0}).collect { |x| figure.print('<line x1="'+(x+20).to_s+'" y1="458" x2="'+(x+20).to_s+'" y2="462" style="stroke:rgb(0,0,64);stroke-width:1"/>\n')}
```

Des ordonnées

L'axe des ordonnées est en vert:

```
figure.puts('<line x1="20.0" y1="460.0" x2="20.0" y2="40.0" style="stroke:rgb(0,40,0);stroke-width:1"/>')

((0..400).select {|y| y%40==0}).collect { |y|

figure.print('<text x="0" y="'+(460-y).to_s+'" style="font-size:16;fill:rgb(0,40,0);font-weight:normal">'+(y/4).to_s+'</text>\n'+'<line x1="15" y1="'+(460-y).to_s+'" x2="25" y2="'+(460-y).to_s+'"  style="stroke:rgb(0,40,0);stroke-width:1"/>\n')}

((0..400).select {|y| y%20==0}).collect { |y| figure.print('<line x1="16" y1="'+(460-y).to_s+'" x2="24" y2="'+(460-y).to_s+'"  style="stroke:rgb(0,40,0);stroke-width:1"/>\n')}

((0..400).select {|y| y%4==0}).collect { |y| figure.print('<line x1="18" y1="'+(460-y).to_s+'" x2="22" y2="'+(460-y).to_s+'"  style="stroke:rgb(0,40,0);stroke-width:1"/>\n')}
```

Courbe

On va représenter graphiquement la fonction en rouge, sous forme d'un polygone à 350 côtés:

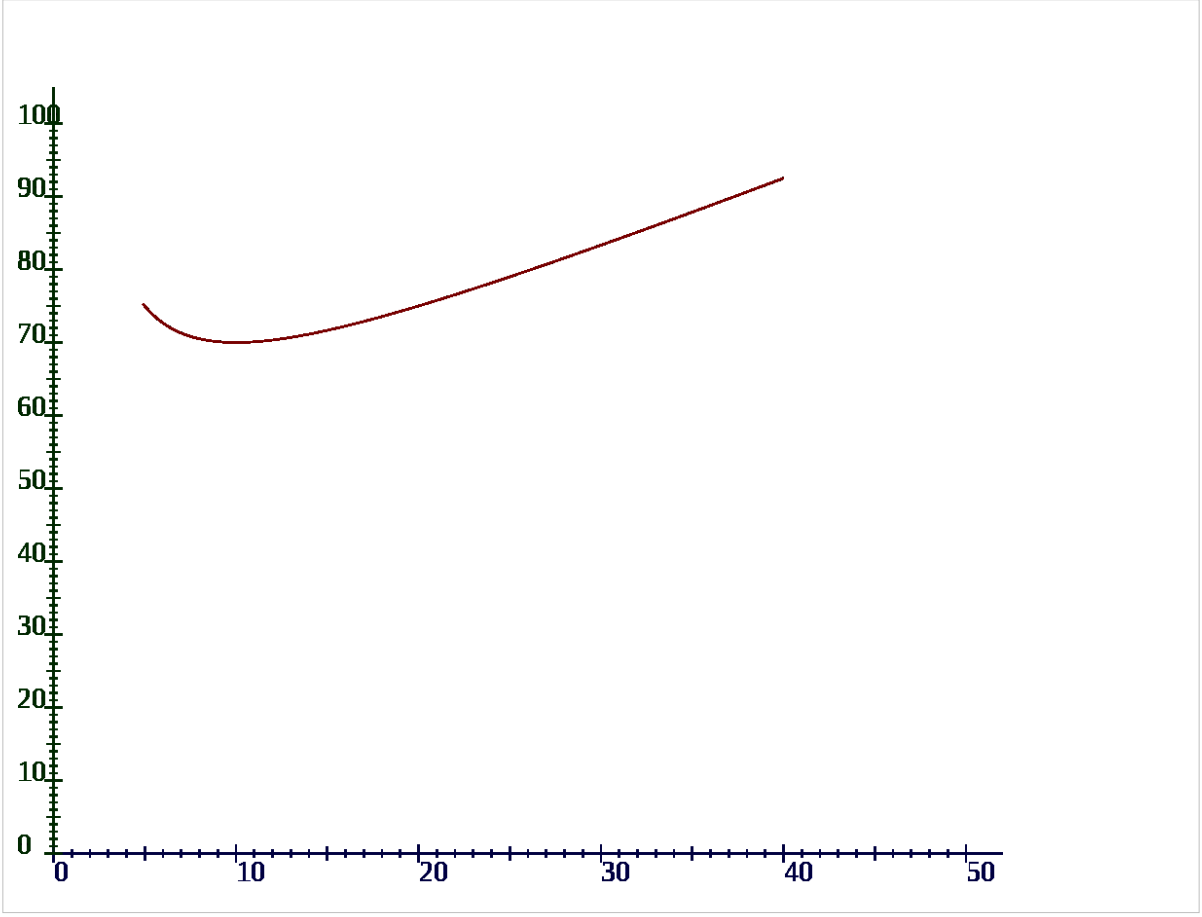
```
(50..400).collect {|x| figure.print('<line x1="'+(19+x).to_s+'" y1="'+(460-4*f((x-1).to_f/10)).to_s+'" x2="'+(20+x).to_s+'" y2="'+(460-4*f(x.to_f/10)).to_s+'"  style="stroke:rgb(120,0,0);stroke-width:1"/>\n')}
```

Résultat

Pour que la figure soit reconnue, il reste encore à fermer la balise *svg*, ouvert au tout début de la figure, puis à fermer le fichier pour que *Ruby* le libère:

```
figure.puts('</svg>')


figure.close
```

L'exécution du script ci-dessus produit le fichier suivant:



Certes, le fichier *Ruby* a l'air encore plus compliqué que le fichier *svg* mais il est possible de créer des méthodes *axes* et *trait* pour en simplifier la lecture.

Analyse numérique en Ruby

Fonction

Les algorithmes ci-dessous seront appliqués à la fonction $f : x \mapsto x^2 - 5$. On va donc commencer par créer une *méthode* pour cela:

```
def f(x)
    return x**2-5
end
```

Résolution numérique d'une équation

Pour chercher à 10^{-14}près un antécédent de 0 par *f*, on peut utiliser la méthode de dichotomie:

```
def zerof(a,b)
    if f(a)*f(b)>0 then
        puts('Pas de solution unique entre '+a.to_s+' et '+b.to_s+'.')
    else
        while ((a-b).abs>1e-14)
            m=(a+b)/2.0
            if f(m)*f(a)>0 then
                a=m
            else
                b=m
            end
        end
    end
    return m
end



puts(zerof(1,3))
```

Le script affiche une solution parce que f(1) est négatif et f(3) positif. Sinon on aurait un message d'erreur.

Calcul approché d'un nombre dérivé

On approche la tangente par une sécante. On utilise une méthode centrée:

```
def NDerf(x)
    h=1e-10
    return (f(x+h)-f(x-h))/(2*h)
end
```

```
puts(NDerf(2))
```

On voit que $f'(2) \simeq 4$

Calcul approché d'une intégrale

La méthode des rectangles consiste à approcher $\int_a^b f(t)\,dt$ par la somme des aires des rectangles de largeur *h* et de hauteur *f(a+nh)* pour *a+nh* allant de *a* à *b*. On choisit *N* assez grand (ici 1 000 000) pour que *h* soit petit et l'approximation bonne:

```
def Nintf(a,b)
    h=(b-a).to_f/1e6
    return (1..1000000).inject{|s,i| s+=h*f(a+h*i)}
end

puts(Nintf(0,2))
```

Points en Ruby

L'objet *Point* est une bonne manière d'aborder la programmation objet. En géométrie repérée, un point est constitué de deux nombres, son abscisse et son ordonnée.

Voici l'énoncé de l'exercice:

Dans un repère orthonormé, on considère $A(-1;3)$, $B(5;1)$et $C(1;5)$. Calculer les distances AB, AC et BC et en déduire la nature du triangle ABC. Puis en déduire les coordonnées du centre de son cercle circonscrit, et le rayon de celui-ci.

Classe

Pour que *Ruby* possède un objet *Point*, il suffit de le définir, sous la forme d'une *classe*:

```
class Point

    def initialize(x,y)
        @x, @y = x, y
    end

end
```

Dorénavant, chaque fois qu'on crée un point par *Point.new(x,y)*, celui-ci possédera les coordonnées *x* et *y* qui sont pour l'instant ses seules propriétés (des variables stockées temporairement dans l'objet).

Coordonnées

Cependant pour accéder depuis l'extérieur aux coordonnées du point, il faut les redéfinir comme des méthodes *Ruby* (parce que dans *Ruby*, tout est méthode).

Abscisse

Il suffit de dire que la méthode *x* renvoit le nombre *x*:

```
    def x
        @x
    end
```

(à l'intérieur de la classe)

Ordonnée

Idem pour *y*:

```
    def y
        @y
    end
```

Dorénavant, l'abscisse de *P* s'appelle *P.x* et son ordonnée, *P.y*.

Affichage

Pour afficher un objet, il faut utiliser la conversion *to_s* que *Ruby* propose. Dans le cas présent, puisqu'on a inventé un nouvel objet, on doit redéfinir cette conversion en chaîne de caractères en mettant les coordonnées entre parenthèses, séparées par un point-virgule:

```
    def to_s
        '('+@x.to_s+';'+@y.to_s+')'
    end
```

Pour afficher un point *M*, on peut faire

```
puts(M.to_s)
```

mais aussi

```
puts(M)
```

puisque *Ruby* se charge automatiquement de la conversion *to_s*.

Deux points

Le plus simple avec deux points, c'est le milieu, parce que c'est un objet de même type (un point):

Milieu

```
    def milieu(q)
        Point.new((@x+q.x)/2,(@y+q.y)/2)
    end
```

La syntaxe est typique de *Ruby*: On parle de "milieu avec q" en invoquant

```
puts(p.milieu(q))
```

Vecteur

Un peu hors sujet ici (on en reparlera dans le chapitre qui leur est consacré), le vecteur $\overrightarrow{AB}$ est bel et bien associé à deux points: son origine *A* et son extrémité *B*. Et ses coordonnées se calculent à partir de celles de *A* et de *B*:

```
    def vecteur(q)
        Vecteur.new(q.x-@x,q.y-@y)
    end
```

Ce qui oblige, soit à placer la classe vecteur dans le même fichier, soit à l'importer avec

```
require 'vector'
```

si on a enregistré ladite classe dans un fichier *vector.rb*.

Là encore, on parle de méthode *vecteur jusqu'à q* pour un point *p*.

Distance

La distance jusqu'à *q* est un nombre, mais associé à deux points:

```
    def distance(q)
        (self.vecteur(q)).norme
    end
```

Pour faire le plus simple possible, on a là encore utilisé le fichier des vecteurs sous la forme de sa méthode *norme*: La distance AB est la norme du vecteur $\overrightarrow{AB}$, qu'on calcule avec la fonction *hypot* de *Ruby* (voir au chapitre suivant comment on l'utilise).
Pour calculer la distance entre *p* et *q*, on entre

```
puts(p.distance(q))
```

ou, au choix,

```
puts(q.distance(p))
```

Application au problème

Pour récapituler, la classe *Point* en entier est décrite ici:

```
class Point

    def initialize(x,y)
        @x, @y = x, y
    end

    def x
        @x
    end

    def y
        @y
    end

    def to_s
        '('+@x.to_s+';'+@y.to_s+')'
    end

    def milieu(q)
        Point.new((@x+q.x)/2,(@y+q.y)/2)
    end

    def vecteur(q)
        Vecteur.new(q.x-@x,q.y-@y)
    end

    def distance(q)
        (self.vecteur(q)).norme
    end

end
```

C'est tout!

On commence par créer trois points, les sommets du triangle:

```
a=Point.new(-1,3)
b=Point.new(5,1)
c=Point.new(1,5)
```

Nature de ABC

Pour voir si ABC est isocèle, on peut afficher les longueurs de ses côtés:

```
puts(a.distance(b))
puts(a.distance(c))
puts(b.distance(c))
```

mais on n'apprend pas grand-chose (sinon qu'il n'est pas isocèle). Mais on peut chercher s'il est rectangle avec la réciproque du théorème de Pythagore:

```
puts(a.distance(b)**2)
puts(a.distance(c)**2+b.distance(c)**2)
```

C'est clair, le triangle ABC est rectangle en C.

Centre du cercle

Il en résulte alors que le cercle circonscrit a pour diamètre *[AB]*, donc pour centre le milieu *M* de *[AB]* et pour rayon la moitié de AB:

```
m=a.milieu(b)

puts(m)
```

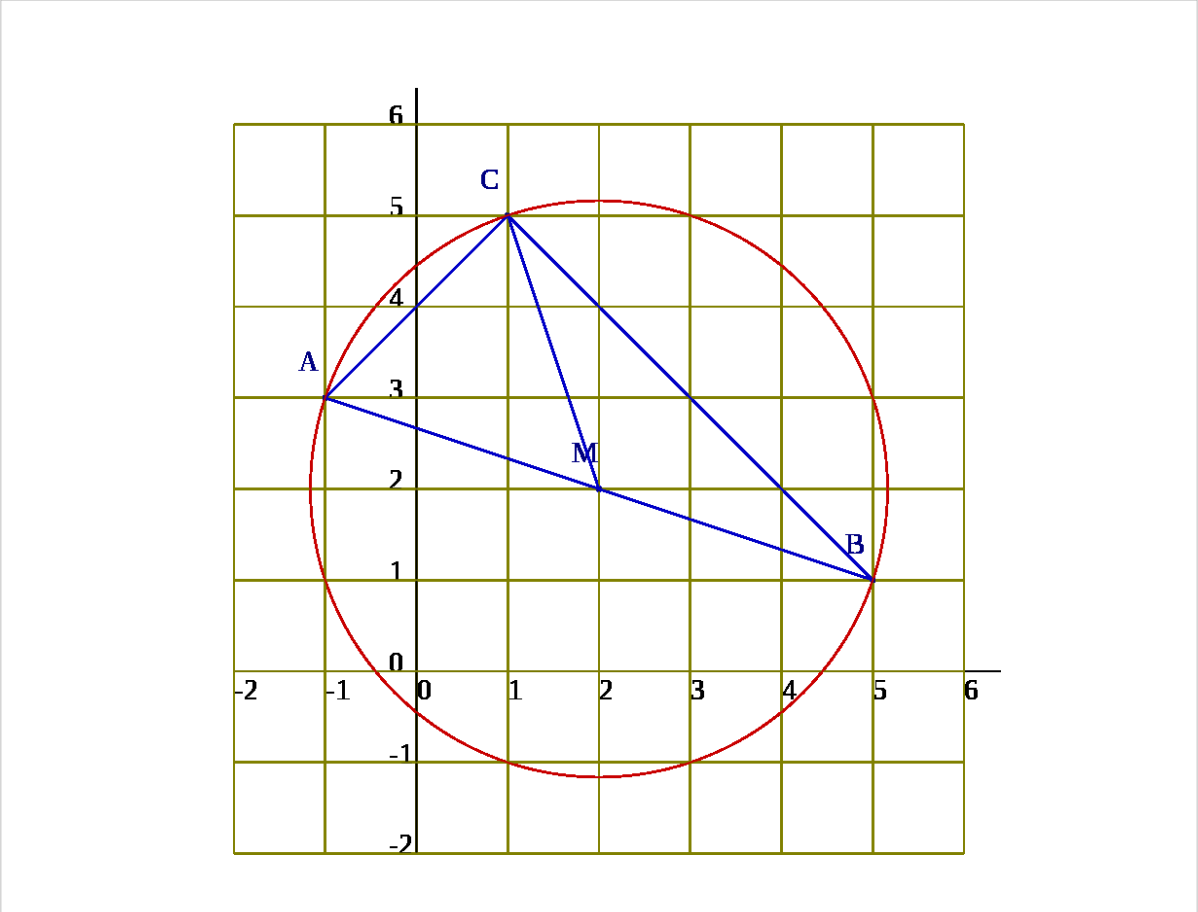
Rayon du cercle

Le rayon peut se calculer en divisant par 2 la distance AB, ou mieux, en vérifiant que les trois rayons MA, MB et MC ont la même longueur à la précision permise par *Ruby*:

```
puts(m.distance(a))
puts(m.distance(b))
puts(m.distance(c))
```

Figure

On peut résumer le tout en modifiant le script ci-dessus pour qu'il crée des éléments *svg*, dont le résultat est visible ci-dessous:



Vecteurs en Ruby

Un vecteur peut être défini par ses deux coordonnées ou avec deux points (son origine et son extrémité). La seconde technique a été abordée dans le chapitre précédent; on utilisera donc ici la définition par les coordonnées:

Définition

Le vecteur sera donc ici une classe *Vecteur*:

```
class Vecteur

    def initialize(x,y)
        @x, @y = x, y
    end

end
```

Créer un vecteur, c'est donc lui donner une abscisse et une ordonnée. Comme on l'a vu dans le chapitre précédent, on peut maintenant en créer en *Ruby* avec

```
v=Vecteur.new(2,3)
```

Coordonnées

On gère les coordonnées et l'affichage comme avec les points; il y a beaucoup de ressemblance entre les vecteurs et les points, ce sont les méthodes qui ne seront pas les mêmes.

Abscisse

```
    def x
        @x
    end
```

Pour lire ou modifier l'abscisse d'un vecteur *u*, on invoque *u.x*.

Ordonnée

```
    def y
        @y
    end
```

Affichage

```
    def to_s
        '('+@x.to_s+';'+@y.to_s+')'
    end
```

Il suffit pour afficher un vecteur *u*, de faire

```
puts(u)
```

Norme

La norme d'un vecteur se calcule avec le théorème de Pythagore:

```
    def norme
        Math.hypot(@x,@y)
    end
```

On a utilisé la norme d'un vecteur pour calculer des distances au chapitre précédent.

Opérations

Il n'est pas d'usage de calculer le milieu de deux vecteurs, mais par contre, on n'additionne pas les points d'habitude (sauf avec GeoGebra). Mais les vecteurs, eux, on les additionne:

Somme

La somme de deux vecteurs est définie par la somme des coordonnées:

```
    def + (u)
        Vecteur.new(@x+u.x,@y+u.y)
    end
```

La notation est infixée ce qui fait que la somme de deux vecteurs u et v se note tout simplement $u+v$.

Produits

Il y a deux multiplications intéressantes:

Par un nombre

En multipliant un vecteur par un nombre, on obtient un vecteur:

```
    def * (r)
        Vecteur.new(@x*r,@y*r)
    end
```

Seulement on est obligé de mettre le nombre après le vecteur ($u*3$ pour avoir le triple de u, alors que d'habitude on fait plutôt le contraire: $3u$), et ce produit est moins intéressant que le suivant:

Par un vecteur

En multipliant un vecteur par un vecteur, on obtient un nombre. Comme les nombres sont disposés comme les barreaux d'une échelle, on appelle cette multiplication, le produit scalaire des deux vecteurs:

```
    def * (u)
        @x*u.x+@y*u.y
    end
```

On écrit $u*v$ pour avoir le produit scalaire de u par v.

Tests

De colinéarité

Pour savoir si deux vecteurs sont colinéaires, on compare deux produits:

```
    def colin(u)
        @x*u.y==@y*u.x
    end
```

Pour savoir si *u* et *v* sont colinéaires, on entre

```
puts(u.colin(v))
```

qui donnera *true* ou *false* selon que les vecteurs sont, ou non, colinéaires.

D'orthogonalité

Pour savoir si deux vecteurs sont orthogonaux, on compare leur produit scalaire à 0:

```
    def ortho(u)
        self * u ==0
    end
```

Exemple

Dans l'exemple du chapitre précédent, le produit scalaire permet plus rapidement de vérifier que le triangle ABC est rectangle:

```
a=Point.new(-1,3)
b=Point.new(5,1)
c=Point.new(1,5)


u=c.vecteur(a)
v=c.vecteur(b)


puts(u*v)
puts(u.ortho(v))
```

Droites en Ruby

La droite peut être définie à partir d'une de ses équations, mais aussi à partir de deux points. Et comme on a vu précédemment comment on peut créer en *Ruby* un objet *point*, on va voir comment on peut s'en servir pour gérer des droites sous *Ruby*.

Définition

Là encore, on va définir une classe *Droite* possédant, lors de son instanciation, deux points:

```
class Droite

    def initialize(a,b)
        @a,@b=a,b
    end
```

Vecteurs

Vecteur directeur

```
    def directeur
        @a.vecteur(@b)
    end
```

On obtient le vecteur directeur de *d* par *d.directeur*

Alignement

Pour savoir si le point *m* est sur la droite d, on peut rajouter ce test:

```
    def IsOnLine(d)
        vecteur(d.a).colin(d.directeur)
    end
```

mais on le rajoute dans l'objet *Point*, puisque c'est une propriété du point...

Vecteur normal

Le vecteur normal s'obtient en choisissant ses coordonnées pour que le produit scalaire avec le vecteur directeur soit nul:

```
    def normal
        Vecteur.new(-self.directeur.y,self.directeur.x)
    end
```

Le vecteur normal s'obtient avec *d.normal* et permet facilement d'avoir l'équation cartésienne ci-dessous.

Équations

Équation cartésienne

```
    def cartesienne

'('+self.normal.x.to_s+')x+('+self.normal.y.to_s+')y='+(self.normal.x*@a.x+self.normal.y*@a.y).to_s
    end
```

Pour afficher l'équation cartésienne de *d*, on entre

```
puts(d.cartesienne)
```

Équation réduite

Comme il y a des divisions à effectuer, on a intérêt à faire appel à *mathn':*

```
require 'mathn'
```

Coefficient directeur

```
    def cd
        self.directeur.y/self.directeur.x
    end
```

Ordonnée à l'origine

```
    def oalo
        @a.y-self.cd*@a.x
    end
```

Équation

L'équation réduite se définit par

```
    def reduite
        'y='+self.cd.to_s+'x+('+self.oalo.to_s+')'
    end
```

et s'obtient par *d.reduite*.

Comparaison de deux droites

Parallélisme

Deux droites sont parallèles lorsque leurs vecteurs directeurs sont colinéaires. Mais aussi (sous réserve qu'elles en aient) lorsqu'elles ont le même coefficient directeur:

```
    def parallele(d)
        self.cd==d.cd
    end
```

Pour savoir si deux droites *d1* et *d2* sont parallèles, on fait

```
puts(d1.parallele(d2))
```

Perpendicularité

Deux droites sont perpendiculaires si et seulement si leurs vecteurs normaux sont orthogonaux:

```
    def perpendiculaire(d)
        self.normal.ortho(d.normal)
    end
```

On aurait aussi pu chercher si le produit de leurs coefficients directeurs est égal à -1.

Intersection

Pour calculer les coordonnées du point d'intersection de deux droites, on résout un système.

Exemple

Dans l'exemple des chapitres précédents, on peut regarder si les deux droites (CA) et (CB) sont perpendiculaires:

```
a=Point.new(-1,3)
b=Point.new(5,1)
c=Point.new(1,5)


d1=Droite.new(c,a)
d2=Droite.new(c,b)
puts(d1.perpendiculaire(d2))
```

Résolution de systèmes en Ruby

Un petit exercice

Au village de Trokhafairtrade dans le Swazibwana occidental, on pratique un mélange de commerce équitable et de troc. Ainsi, un habitant a acquis 2 youkoulélés d'Hawaii contre 3 xylophones et 1 € et un écotouriste a acquis un xylophone et un youkoulélé pour la modique somme de 8 €. On demande le prix, en euros, d'un xylophone et le prix d'un youkoulélé sur le marché de Trokhafairtrade.

En notant *x* le prix d'un *xylophone* et *y* celui d'un *youkoulélé*, l'énoncé se traduit algébriquement par *2y=3x+1* et *x+y=8*.

On va donc voir comment résoudre le système de deux équations à deux inconnues suivant:

$$\left\{\begin{array}{rcl} 3x-2y & = & -1 \\ x+y & = & 8 \end{array}\right.$$

Méthode itérative

Dans le cas présent, il se trouve que *x* et *y* sont entiers naturels. Si on sait que c'est le cas, on peut les chercher par tâtonnement avec une boucle sur *x* et sur *y*. On va successivement fabriquer un tableau bidimensionnel avec les entiers de 0 à 100 (deux premières lignes) puis regarder quels couples de ce tableau vérifient à la fois les deux conditions données par le système:

```
total=[[]]
(0..100).each{|x| (0..100).each{|y| total.push([x,y])}}
solutions=total.select{|c| 3*c[0].to_f-2*c[1].to_f==-1 and
c[0].to_f+c[1].to_f==8}
puts(solutions)
```

Méthode algébrique

Le système $\left\{\begin{array}{rcl} 3x-2y & = & -1 \\ x+y & = & 8 \end{array}\right.$ peut aussi s'écrire matriciellement $\begin{pmatrix} 3 & -2 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}=\begin{pmatrix} -1 \\ 8 \end{pmatrix}$ soit $AX=B$ avec $A=\begin{pmatrix} 3 & -2 \\ 1 & 1 \end{pmatrix}$ et $B=\begin{pmatrix} -1 \\ 8 \end{pmatrix}$. Alors sa solution $X=\begin{pmatrix} x \\ y \end{pmatrix}$ s'obtient par le calcul matriciel $X=A^{-1}B$, ce qui se fait directement avec le module *matrix* de *Ruby*:

```
require 'matrix'
require 'mathn'

A=Matrix[[3,-2],[1,1]]
B=Matrix[[-1],[8]]
solution=A.inverse*B
puts(solution)
```

En ayant choisi *mathn* avec, les solutions s'écrivent automatiquement sous forme de fractions si elles ne sont pas entières.

Triplets pythagoriciens en Ruby

L'énoncé du problème est simple, sa solution avec *Ruby* aussi:

Énoncé

Trouver tous les triplets pythagoriciens (x,y,z) tels que $x+y+z=1000$; autrement dit, on demande les triangles rectangles de périmètre 1000 dont les côtés sont entiers.

```
(1..1000).collect{|y| (1..y).collect{|x| z=Math.hypot(x,y) ;
puts(x,y,z) if x+y+z==1000}}
```

On apprend qu'il n'y a qu'un triplet de somme 1000.

Systèmes congruentiels en Ruby

Rallye mathématique de la Réunion

Sujet 2005, Exercice 2

Énoncé

Pour organiser une grande manifestation sportive, le professeur d'éducation physique doit rassembler sur le stade un important groupe d'élèves. Le nombre d'élèves est compris entre 2 800 et 2 900. Il en profite pour leur faire remarquer que, regroupés par 2, puis par 3, puis par 4, puis par 5, puis par 6, il en reste toujours 1 ; mais, ô miracle, en se regroupant par 7, il ne reste personne.

On demande combien d'élèves il y a au total.

On va filtrer les solutions avec les critères donnés par l'énoncé, l'un après l'autre.

Intervalle

les nombres à tester sont tous les nombres entre 2800 et 2900, il y en a donc 101:

```
solutions=(2800..2900).to_a

puts(solutions.size)
```

Par 2

On ne va finalement garder que les nombres qui, pris 2 par 2, laissent 1 (soit tels que le reste de la division par 2 donne 1; on rappelle que ce reste se note par le symbole *pourcent*).

```
solutions=solutions.select{|n| n%2==1}

puts(solutions.size)
```

Ah! Il n'en reste plus que 50 à tester!

Par 3

Parmi ces 50, on va garder uniquement ceux qui, pris 3 par 3 aussi, laissent 1:

```
solutions=solutions.select{|n| n%3==1}

puts(solutions.size)
```

De 50, on est passé à 17.

Par 4

Ensuite on garde ceux des 17 qui, divisés par 4, laissent 1:

```
solutions=solutions.select{|n| n%4==1}

puts(solutions.size)
```

Plus que 8...

Par 5

On continue l'épuration:

```
solutions=solutions.select{|n| n%5==1}

puts(solutions)
```

Plus que deux candidats possibles !

Par 6

```
solutions=solutions.select{|n| n%6==1}

puts(solutions)
```

Pas de changement, toujours deux candidats.

Par 7

Maintenant on veut garder les nombres divisibles par 7. Le reste euclidien devra donc être nul:

```
solutions=solutions.select{|n| n%7==0}

puts(solutions)
```

Et hop! On a le nombre d'élèves!

Sujet 2007, Exercice 3

Énoncé

Chaque semaine, Jean ramasse entre 40 et 200 œufs qu'il va vendre au marché.

Ce soir, veille de marché, il est perplexe.

• S'il met ses œufs dans des emballages de 6, il en reste 2.

• S'il utilise des emballages de 10, il en reste encore 2.

• Il me faudrait, dit-il, des emballages de 8 pour tout contenir exactement.

On demande combien il y a d'œufs en tout.

Même exercice que celui d'au-dessus:

Intervalle

```
solutions=(40..200).to_a

puts(solutions.size)
```

Par 6

```
solutions=solutions.select{|n| n%6==2}

puts(solutions.size)
```

Plus que 27 nombres à tester.

Par 10

```
solutions=solutions.select{|n| n%10==2}

puts(solutions)
```

Plus que 5 nombres à tester.

Par 8

On veut que le reste dans la division par 8 soit nul:

```
solutions=solutions.select{|n| n%8==0}

puts(solutions)
```

Un seul nombre a réussi le parcours du combattant jusqu'au bout: C'est la solution au problème.

Freudenthal sous Ruby

Le problème de Freudenthal est intéressant à traiter en *Ruby* parce qu'il peut se résoudre en manipulant des tableaux et ensembles et que pour *Ruby*, les tableaux peuvent se manipuler comme des ensembles et *vice-versa*. Le problème est d'ailleurs intéressant en soi parce qu'il porte sur la logique épistémique. En voici l'énoncé traduit du Néerlandais à l'Anglais puis au Français:

Énoncé

On choisit au hasard deux entiers x et y strictement supérieurs à 1, x étant le plus petit des deux, et on donne à Sam leur somme qui est inférieure à 100, et à Polly leur produit. Après un temps de réflexion suffisamment long, le dialogue suivant se déroule entre Sam et Polly:

- Polly: Je ne sais pas qui sont x et y.
- Sam: Je savais que tu ne savais pas!
- Polly: Alors je sais qui sont x et y.
- Sam: Alors je sais aussi!

Le but du problème est donc d'utiliser les connaissances qu'on a sur les connaissances de Polly et Sam pour savoir si on peut savoir quels sont *x* et *y*.

Il y a de nombreuses méthodes pour y arriver, et *Ruby* est en quelque sorte un langage "multiparadigme" où chacun peut utiliser sa propre logique (épistémique ou non) pour aborder le problème.

Digression arithmétique

Si on avait confié un nombre premier à Polly, elle n'aurait pas avoué son impuissance lors de sa première affirmation. Bien que ce ne soit nullement nécessaire pour résoudre ce problème, *Ruby* sait depuis la version 1.9 manipuler des nombres premiers. Avec

```
puts(Prime.prime?(p))
```

on peut savoir si *p* est premier, et avec

```
puts(Prime.prime_division(n))
```

on décompose *n* en facteurs premiers.

Les produits à partir des sommes

Plutôt que de manipuler des couples d'entiers *x* et *y* possibles, on va, comme dans la version Python, manipuler l'ensemble des produits possibles. Donc on aura besoin de l'outil permettant, à partir d'une somme *s* donnée à Sam, d'obtenir la liste *prod(s)* des produits donnés à Polly qui correspondent aux mêmes *x* et *y*:

```
def prod(n)
    t=[]
    (2..n-2).each{|x| t.push(x*(n-x))}
    return t.uniq
end
```

On peut décrire l'algorithme ci-dessus ainsi:

1. On crée un tableau vide *t*
2. Pour chaque nombre entre 2 et n-2, noté provisoirement *x*, on rajoute le produit de *x* par son complément à *n* dans le tableau *t*
3. Quand on a fini, on transforme *t* en ensemble, en enlevant les doublons.

Première affirmation

L'affirmation apporte une information: Le produit donné à Polly peut s'obtenir de plusieurs manières, sinon Polly connaîtrait les facteurs. Pour exploiter cette information, on va commencer par fabriquer l'énorme liste des produits possibles, puis ne garder que ceux qui apparaissent au moins deux fois dans la liste:

On ramasse les *y* entre 3 et 100; pour chacun d'entre eux on ramasse (*collect*) les *x* entre 2 et *y-1*; si la somme *x+y* est inférieure ou égale à 100, on rajoute le produit dans le tableau *produit*. Ensuite on constitue pour chaque *p* de ce tableau, l'ensemble des *k* égaux à *p* dans le tableau. Si cet ensemble contient un seul élément, on l'enlève (*reject*) du tableau des produits:

```
produits=[]
(3..100).collect{|y| (2..y-1).collect{|x| if x+y<=100 then produits.push(x*y) end}}



polly=produits.reject{|p| (produits.select{|k| k==p}).size<2}
puts(polly.size)
```

Ah oui! Il y en a beaucoup qui restent!

Deuxième affirmation

Si Sam sait que Polly ne sait pas, c'est parce que quelle que soit la décomposition en somme d'entiers de celui qu'on lui a dicté, le produit correspondant est dans la liste précédente. Sinon elle eût pu savoir qui sont *x* et *y*, pour ce que Sam en sait! Sam ne va donc garder que les sommes n pour lesquelles la liste prod(n) calculée avec la fonction ci-dessus ne contient que des éléments de la liste *polly*, donc si leur intersection est égale à *prod(n)* (en effet $A \subset B \Leftrightarrow A \cap B = A$ dans l'algèbre de Boole des ensembles):

```
sam=(4..100).select{|s| prod(s)&polly==prod(s)}
puts(sam)
```

Il reste plutôt peu de sommes possibles:

11	17	23	27	29	35	37	41	47	53

On sait ce que Sam sait, mais Sam dit "Ça me suffit pas encore!", et nous aussi!

Troisième affirmation

Si Polly sait maintenant quels sont *x* et *y*, c'est que parmi les sommes ci-dessus, il n'y a pas que des doublons (produits communs à plusieurs sommes). Il y a un produit propre à une des sommes de Sam ci-dessus. Pour en savoir plus, Polly va constituer pour chacun des 10 nombres *s* ci-dessus, la liste de ses produits *prod(s)*, puis chercher tous les nombres communs à au moins deux de ces 10 listes (les doublons). *Ruby* peut en faire de même, mais après avoir aplati le tableau des *prod(s)* (qui est un tableau à deux dimensions), et en plaçant dans la liste des doublons, tous les nombres qui apparaissent plus d'une fois dans le tableau des produits:

```
produits=sam.map{|s| prod(s)}
listeprods=produits.flatten

doublons=listeprods.select{|p| (listeprods.select{|k| k==p}).size>1}
```

Ensuite Polly enlève à chaque liste de produits des sommes de *sam*, les doublons (par une soustraction d'ensembles), et en comptant le nombre d'éléments de chacun des ensembles obtenus,

```
produits.each{|p| puts((p-doublons).size)}
```

Polly constate effectivement la présence d'un et un seul singleton:

11	**17**	**23**	**27**	**29**	**35**	**37**	**41**	**47**	**53**
3	1	3	9	9	10	7	13	13	18

Quatrième affirmation

Puisque Sam sait aussi sa somme, on sait que sa somme est 17 et le produit de Polly, 52:

```
puts(sam[1])
puts(prod(17)-doublons)
```

À la recherche des x et y perdus

Maintenant qu'on connaît la somme et le produit de *x* et *y*, il reste à déterminer ceux-ci, ce qui peut se faire en résolvant l'équation du second degré $x^2 - 17x + 52 = 0$ou par une double boucle:

```
(3..100).collect{|y| (2..y-1).collect{|x| if x+y==17 and x*y==52 then
puts(x,y) end}}
```

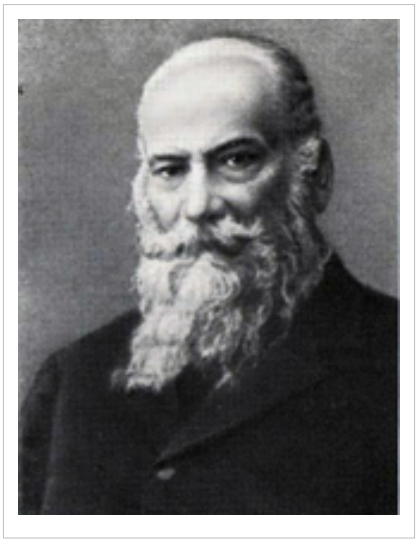
Les inconnues *x* et *y* sont maintenant connues!

Joukovski et Ruby

L'écoulement d'un fluide autour d'un cylindre est aisé à calculer (par exemple pour étudier l'effet Magnus), et toute transformation conforme permet d'en déduire l'écoulement autour du transformé du cercle (la trace du cylindre). En particulier lorsque le transformé a la forme d'une aile d'avion, ce qui est le cas avec la transformée de Joukovski. Ce calcul sera effectué par *Ruby* grâce à ses nombres complexes, puis affiché en fabriquant une figure svg. En fait:

1. On utilise l'inverse de la transformée de Joukovski pour calculer l'écoulement autour du cercle unité.
2. On agrandit légèrement ce cercle pour qu'il ait une transformée de Joukovski plus intéressante;
3. On applique au nouveau cercle la transformée de Joukovski: On a l'écoulement autour de l'aile.

C'est la méthode utilisée par Nikolaï Joukovski au début du vingtième siècle pour les premières études sur l'aéronautique.



Du cercle au segment

Pour calculer l'écoulement autour du cercle unité, il suffit de trouver une transformation conforme qui transforme un segment (dont l'écoulement est trivial) en le cercle unité. Or l'image d'un point quelconque e^{it} du cercle unité par la transformée de Joukovski $z \mapsto z + \frac{1}{z}$est $e^{it}+e^{-it}=2\cos(t)$: La transformation de Joukovski J(z) transforme le cercle unité en le segment [-2;2].

Et l'écoulement autour du segment est le plus simple possible, puisque c'est comme si le segment n'était pas là: Des droites verticales représentent les équipotentielles (ou isobares), et des droites verticales représentent les lignes de courant (ou écoulement) qui montrent la vitesse du vent. La transformation conforme qui produit cette grille est la transformation identique *id(z)=z*.

Du segment au cercle

Transformée inverse

Il suffit donc d'appliquer à cet écoulement l'inverse J^{-1} de la transformée de Joukovski, qui envoie le segment [-2;2] sur le cercle unité. Or J^{-1} est multiforme. On utilisera donc la fonction *fplus* valant $f_{plus}(z) = \frac{z+\sqrt{z^2-4}}{2}$sur

la moitié droite de l'image, et la fonction *fmoins* valant $f_{moins}(z) = \frac{z - \sqrt{z^2 - 4}}{2}$ sur la moitié gauche. Ces fonctions sont compl

```
require 'cmath'

def fplus(z)
      return (z+CMath.sqrt(z**2-4))/2
end
def fmoins(z)
      return (z-CMath.sqrt(z**2-4))/2
end
R=100
```

Le paramètre *R* est un facteur d'échelle permettant de zoomer sans avoir à refaire tout le script.

La figure est enregistrée dans un fichier au format *svg* appelé *JoukovskiRuby1.svg*:

```
figure=File.open("JoukovskiRuby1.svg","w")
figure.puts('<?xml version="1.0" encoding="utf-8"?>')
figure.puts('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"')
figure.puts('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">')
figure.puts('<svg xmlns="http://www.w3.org/2000/svg" width="640" height="480">')
```

La figure occupera donc 640 pixels de large et 480 pixels de haut, et s'appellera *figure* tout simplement.

Tracé des isobares

Côté droit de la figure

Pour tracer les isobares, on prend un point d'affixe x-2i et on le transforme par J^{-1}. Puis on commence un *path* de *svg* par ce point. Ensuite on transforme des points situés sur la même droite verticale que le premier point choisi, et on les ajoute au *path* (une chaîne de caractères appelée *ligne*). La courbe obtenue (le *path*) sera en jaune et est inscrite à la fin dans le fichier:

```
#moitié droite des isobares
(0..25).collect { |i|
    z=fplus(Complex(i/10.0,-2.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (-20..20).collect { |j|
            z=fplus(Complex(i/10.0,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="yellow" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}
```

Côté gauche de la figure

Les isobares de gauche se dessinent de la même manière, en remplaçant *fplus* par *fmoins*:

```
#moitié gauche des isobares
(-25..0).collect { |i|
    z=fmoins(Complex(i/10.0,-2.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (-20..20).collect { |j|
            z=fmoins(Complex(i/10.0,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="yellow" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}
```

Tracé des lignes de courant

Pour les mettre en exergue, on fera les lignes de courant en rouge.

Côté gauche de la figure

Il suffit d'intervertir les rôles de *i* (abscisse) et *j* (ordonnée):

```
#moitié gauche de l'écoulement
(-25..25).collect { |j|
    z=fmoins(Complex(-3.0,j/10.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (-30..0).collect { |i|
            z=fmoins(Complex(i/10.0-0.000001,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="red" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}
```

Le décalage des abscisses permet d'éviter le passage par l'origine, où le changement de feuillet de la transformation donnerait des résultats graphiquement bizarres.

Côté droit de la figure

Même principe, en remplaçant *fmoins* par *fplus*:

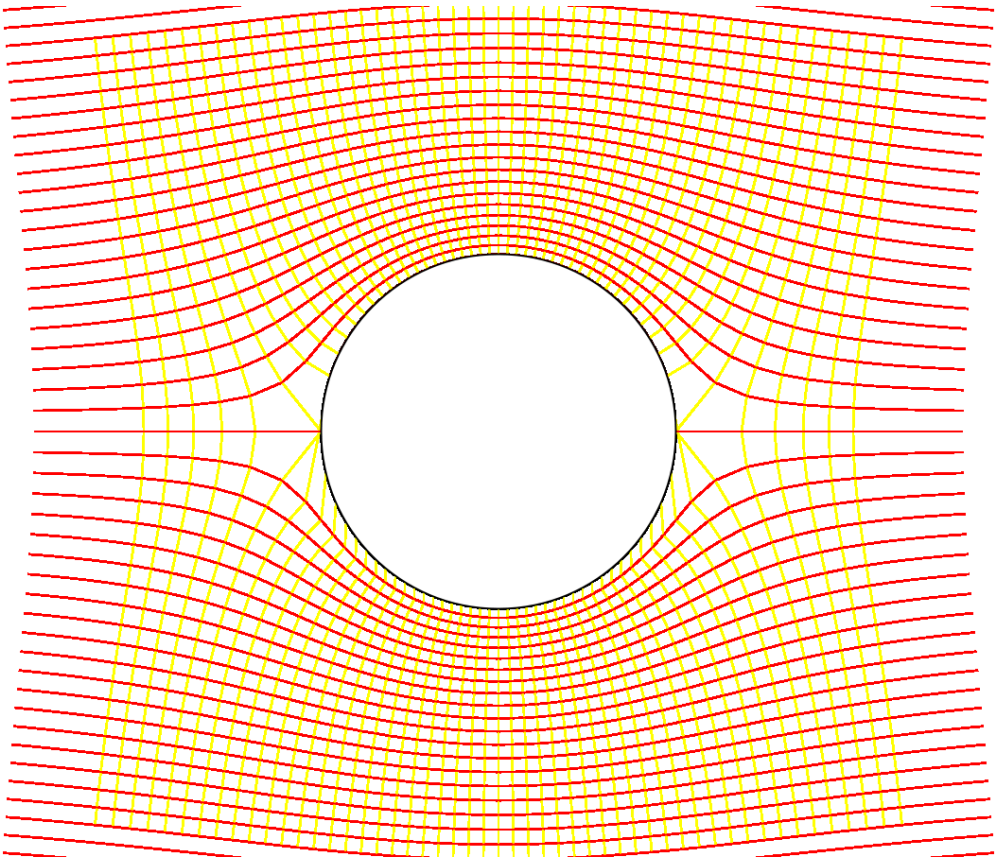
```
#moitié droite de l'écoulement
(-25..25).collect { |j|
    z=fplus(Complex(0,j/10.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (0..30).collect { |i|
            z=fplus(Complex(i/10.0,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="red" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}
```

Tracé du cercle

Pour ajouter le cercle lui-même sur la figure, on peut utiliser l'objet *circle* de *svg*; on en profite pour fermer la bannière *svg* et clore le fichier:

```
figure.puts('<circle cx="320" cy="240" r="'+R.to_s+'" fill="white" stroke="black" stroke-width="1" />')
figure.puts('</svg>')
figure.close
```

C'est tout ce qu'il faut pour obtenir l'écoulement que voici:



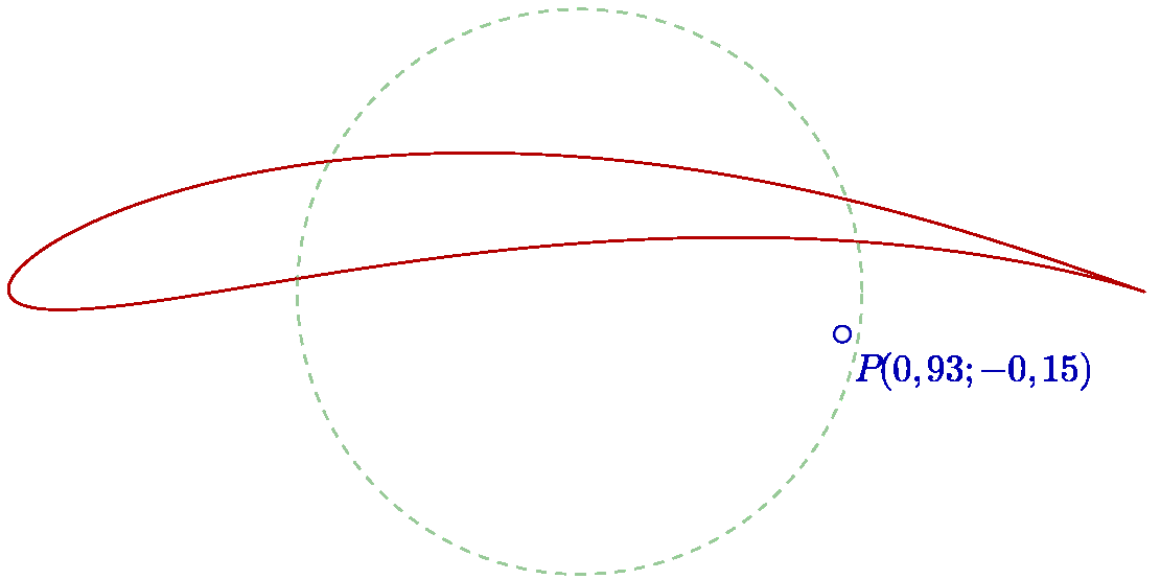
Le défaut des isobares en-dessous du cercle (la figure aurait dû être symétrique par rapport à l'axe des abscisses) est lié au changement de feuillet de J^{-1}. Pour y remédier, on peut remplacer la moitié du bas par la symétrique de la moitié du haut, ce qui double la longueur du script (une fonction par quadrant). La correction est laissée en exercice.

Conception de l'aile d'avion

Si on applique à la figure ci-dessus (avec le cercle unité), la transformation J(z), on obtient l'écoulement autour de [-2;2] qui est trivial. Mais on peut modifier le cercle pour qu'il passe toujours par le point d'affixe 1, et la transformée de Joukovski du nouveau cercle aura toujours un point de rebroussement d'affixe 2, mais peut avoir une forme de lemniscate, ou de profil d'aile d'avion, selon le paramètre choisi.

Tracé du profil

Une fonction affine complexe laisse le point d'affixe 1 invariant, si elle est de la forme *fa(z)=P(z-1)+1*. Le meilleur moyen de choisir la valeur du coefficient directeur *P* est d'implémenter *fa* et J avec un logiciel de géométrie dynamique (par exemple, CaRMetal qui exporte au format svg), et de bouger le point d'affixe *P* avec la souris jusqu'à ce que l'image du cercle par J(fa(z)) ait la forme voulue:



Le choix de P=0,93+0,15i paraît satisfaisant.

Calcul de l'écoulement

Pour la fonction de Joukovski, on ajoute un très petit nombre à z pour que le nombre zéro passe entre les mailles du filet, ce qui évite le risque d'avoir une erreur de division par zéro. À part ça, il suffit d'appliquer J(fa(z)) au cercle unité, et de remplacer le cercle final par son image par J(fa(z)) (pour dessiner l'aile). Ce qui donne le script suivant:

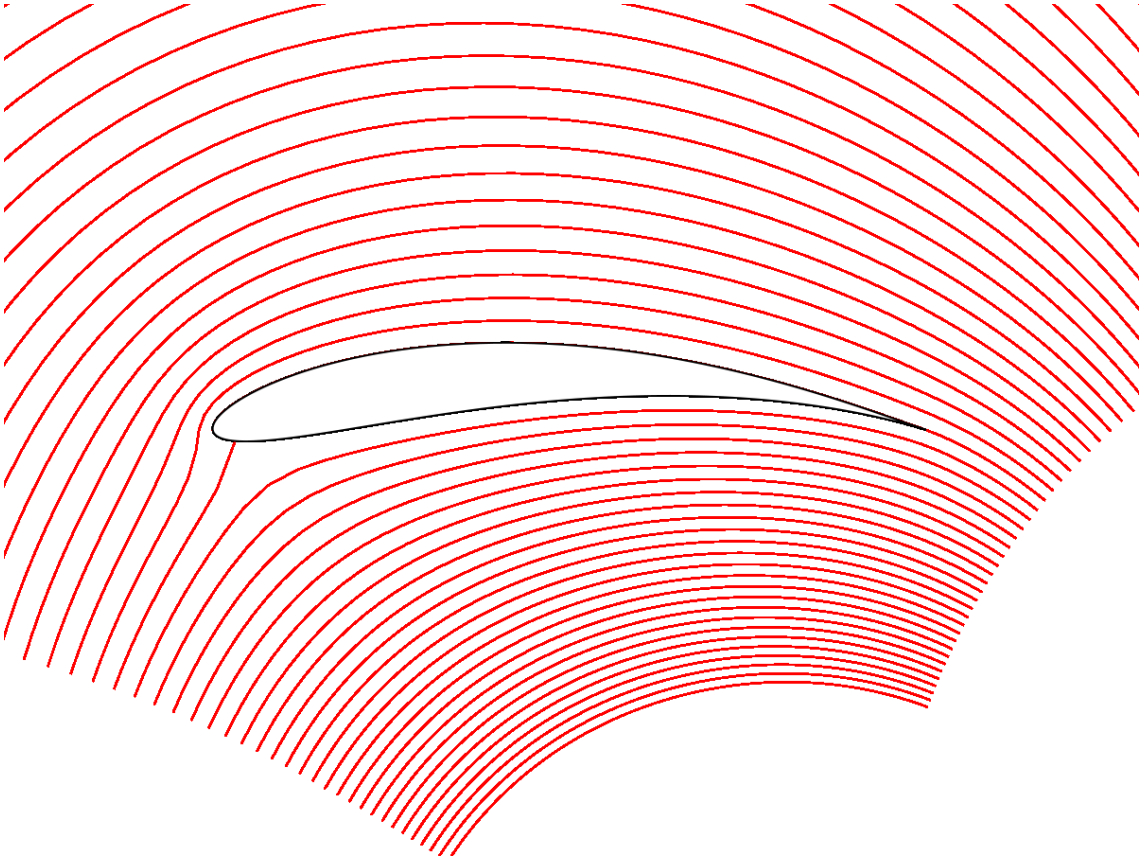
```
require 'cmath'

def fa(z)
      return Complex(0.93,-0.15)*(z-1)+1
end

def Joukovski(z)
      return z+1/(z+0.0000001)
end

def fplus(z)
      return Joukovski(fa((z+CMath.sqrt(z**2-4))/2))
end
def fmoins(z)
      return Joukovski(fa((z-CMath.sqrt(z**2-4))/2))
end
R=100


figure=File.open("JoukovskiRuby2.svg","w")
```

```
figure.puts('<?xml version="1.0" encoding="utf-8"?>')
figure.puts('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"')
figure.puts('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">')
figure.puts('<svg xmlns="http://www.w3.org/2000/svg" width="640" height="480">')
#moitié gauche de l'écoulement
(-25..25).collect { |j|
    z=fplus(Complex(-3.0,j/10.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (-30..0).collect { |i|
            z=fplus(Complex(i/10.0-0.000001,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="red" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}
#moitié droite de l'écoulement
(-25..25).collect { |j|
    z=fmoins(Complex(0,j/10.0))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne='<path d="M'
      ligne+=xa.to_s+' '+ya.to_s
      (0..30).collect { |i|
            z=fmoins(Complex(i/10.0,j/10.0))
            xa=z.real*R+320
            ya=240-z.imag*R
            ligne+=' L'+xa.to_s+' '+ya.to_s

      }
      ligne+='" stroke="red" stroke-width="1" fill="none"/>'
      figure.puts(ligne)
}

ligne='<path d="M'+(320+2*R).to_s+' 240'
(0..200).collect { |i|
      t=Math::PI*i/100
      z=Joukovski(fa(Complex(Math.cos(t),Math.sin(t))))
      xa=z.real*R+320
      ya=240-z.imag*R
      ligne+=' L'+xa.to_s+' '+ya.to_s
}
```

```
ligne+='" stroke="black" stroke-width="1" fill="white"/>'
figure.puts(ligne)

figure.puts('</svg>')

figure.close
```

Figure obtenue

L'exécution du script produit la figure suivante:



Le fait que les lignes de courant sont plus courbées sur le dessus que sur le dessous entraîne par la force de Bernoulli, une dépression sur le dessus de l'aide, qui est à l'origine de la portance: L'avion vole!

Sources et contributeurs de l'article

Mathématiques avec Python et Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=420138 *Contributeurs*: 2A01:E34:EE01:3420:225:FF:FE4A:D3CB, Alain Busser, Catintp, JackPotte, Savant-fou, Sub

Nombres entiers en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=315030 *Contributeurs*: Alain Busser, Melnofil

Fractions en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=361755 *Contributeurs*: Alain Busser, Savant-fou

Nombres réels en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=318453 *Contributeurs*: Alain Busser

Nombres complexes en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=367683 *Contributeurs*: Alain Busser, 1 modifications anonymes

Quaternions et octonions en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=347415 *Contributeurs*: Alain Busser

Ensembles en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=315017 *Contributeurs*: Alain Busser

Nombres pseudoaléatoires en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403109 *Contributeurs*: Alain Busser

Simulation avec Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=315049 *Contributeurs*: Alain Busser

Statistique inférentielle avec Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403108 *Contributeurs*: 2001:660:6602:B:2677:3FF:FE32:A524, Alain Busser

Suites en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=336640 *Contributeurs*: Alain Busser, 2 modifications anonymes

Fonctions en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403107 *Contributeurs*: Alain Busser

Analyse numérique en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=427049 *Contributeurs*: Alain Busser, 1 modifications anonymes

Points en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403106 *Contributeurs*: Alain Busser

Vecteurs en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=315058 *Contributeurs*: Alain Busser

Droites en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=315015 *Contributeurs*: Alain Busser

Une tortue qui accélère la résolution de problèmes *Source*: http://fr.wikibooks.org/w/index.php?oldid=350580 *Contributeurs*: Alain Busser

Résolution de systèmes en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403105 *Contributeurs*: Alain Busser, 3 modifications anonymes

Triplets pythagoriciens en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403104 *Contributeurs*: Alain Busser

Systèmes congruentiels en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403103 *Contributeurs*: Alain Busser

Freudenthal en Python *Source*: http://fr.wikibooks.org/w/index.php?oldid=403102 *Contributeurs*: Alain Busser

Nombres entiers en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315031 *Contributeurs*: Alain Busser

Fractions en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315022 *Contributeurs*: Alain Busser

Nombres réels en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=351777 *Contributeurs*: Alain Busser

Nombres complexes en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315026 *Contributeurs*: Alain Busser

Quaternions et octonions en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=351943 *Contributeurs*: Alain Busser

Ensembles en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315018 *Contributeurs*: Alain Busser

Nombres pseudo-aléatoires en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403110 *Contributeurs*: Alain Busser

Suites en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315053 *Contributeurs*: Alain Busser

Fonctions en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403101 *Contributeurs*: Alain Busser

Analyse numérique en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315244 *Contributeurs*: Alain Busser

Points en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403100 *Contributeurs*: Alain Busser, 1 modifications anonymes

Vecteurs en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315059 *Contributeurs*: Alain Busser

Droites en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=315245 *Contributeurs*: Alain Busser

Résolution de systèmes en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403099 *Contributeurs*: Alain Busser

Triplets pythagoriciens en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403098 *Contributeurs*: Alain Busser

Systèmes congruentiels en Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403097 *Contributeurs*: Alain Busser

Freudenthal sous Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=403096 *Contributeurs*: Alain Busser

Joukovski et Ruby *Source*: http://fr.wikibooks.org/w/index.php?oldid=356867 *Contributeurs*: Alain Busser

Source des images, licences et contributeurs

Fichier:Janus-Vatican.JPG *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Janus-Vatican.JPG *Licence*: Public Domain *Contributeurs*: Fubar Obfusco

Fichier:Python logo and wordmark.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Python_logo_and_wordmark.svg *Licence*: GNU General Public License *Contributeurs*: www.python.org

Fichier:Ruby-logo-notext.png *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Ruby-logo-notext.png *Licence*: Creative Commons Attribution-Sharealike 3.0 *Contributeurs*: Tom Schaub

Fichier:Sheep_herding,_Arkansas.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Sheep_herding,_Arkansas.jpg *Licence*: Public Domain *Contributeurs*: Ken Hammond

Fichier:Cake_quarters.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Cake_quarters.svg *Licence*: Public Domain *Contributeurs*: Acdx, R. S. Shaw

Fichier:Oudjat.SVG *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Oudjat.SVG *Licence*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributeurs*: User:BenduKiwi

File:Sin1perz.png *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Sin1perz.png *Licence*: Creative Commons Attribution-Sharealike 3.0 *Contributeurs*: Kovzol

Fichier:Dice01.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Dice01.jpg *Licence*: Public Domain *Contributeurs*: Original uploader was (Automated conversion) at en.wikipedia (Original text : LDC)

Fichier:Carte_française_pique_10.gif *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Carte_française_pique_10.gif *Licence*: GNU Free Documentation License *Contributeurs*: Docu, François Haffner

Fichier:Carte_française_carreau_12.gif *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Carte_française_carreau_12.gif *Licence*: GNU Free Documentation License *Contributeurs*: Docu, François Haffner

Fichier:Votingwomen.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Votingwomen.jpg *Licence*: inconnu *Contributeurs*: JackyR, Jorva, Ranveig, TwoWings, ~Pyb, 1 modifications anonymes

Fichier:rencontres.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Rencontres.svg *Licence*: inconnu *Contributeurs*: Alain Busser

Fichier:GeomRuby1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:GeomRuby1.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:Galtonpy1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Galtonpy1.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:Bundesarchiv Bild 183-1990-0703-023, Dresden, Rauschgiftspürhund im Einsatz.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Bundesarchiv_Bild_183-1990-0703-023,_Dresden,_Rauschgiftspürhund_im_Einsatz.jpg *Licence*: Creative Commons Attribution-Sharealike 3.0 Germany *Contributeurs*: Anka Friedrich, BlackIceNRW, Burts, DynaMoToR, Hystrix, Miyagawa, Pitke, Ras67, SunOfErat

Fichier:Kochpy1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Kochpy1.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:Kochpy2.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Kochpy2.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:systemkig1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Systemkig1.svg *Licence*: inconnu *Contributeurs*: Alain Busser

Fichier:Freerange_eggs.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Freerange_eggs.jpg *Licence*: inconnu *Contributeurs*: Apdency, Bdk, EEG4Hung, EugeneZelenko, Ies, Kri, Lobo, Notafish, Ranveig, Zaphod, 2 modifications anonymes

Fichier:UF_Forensic_Science.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:UF_Forensic_Science.jpg *Licence*: Creative Commons Attribution-Sharealike 3.0 *Contributeurs*: Apollidon

Fichier:Sin1perz.png *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Sin1perz.png *Licence*: Creative Commons Attribution-Sharealike 3.0 *Contributeurs*: Kovzol

Fichier:Eine_Kopeke_-_Zar_Alexander_I_1821.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:Eine_Kopeke_-_Zar_Alexander_I_1821.jpg *Licence*: Public Domain *Contributeurs*: unbekannt. Original uploader was DALIBRI at de.wikipedia

Fichier:HistogramRuby1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:HistogramRuby1.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:SuiteRuby01.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:SuiteRuby01.svg *Licence*: inconnu *Contributeurs*: Alain Busser

Fichier:FonctionRuby01.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:FonctionRuby01.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:NikolaiYegorovichZhukovsky.jpg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:NikolaiYegorovichZhukovsky.jpg *Licence*: Public Domain *Contributeurs*: Original uploader was Xchgall at ru.wikipedia

Fichier:JoukovskiRuby1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:JoukovskiRuby1.svg *Licence*: Creative Commons Attribution-Sharealike 3.0 *Contributeurs*: User:Alain Busser

Fichier:JoukovskiCaRMetal1.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:JoukovskiCaRMetal1.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Fichier:JoukovskiRuby2.svg *Source*: http://fr.wikibooks.org/w/index.php?title=Fichier:JoukovskiRuby2.svg *Licence*: inconnu *Contributeurs*: Alain Busser, TouzaxA

Licence