# Nombres réels approchés

## Algo & Prog avec R

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

6 avril 2019

Université Côte d'Azur, CNRS, I3S, France
firstname.lastname@univ-cotedazur.fr

# Nombres réels approchés

Ou nombres réels inexacts. On parle de nombres flottants (float).

**Calcul entier et réel en précision finie**
Ils n'ont qu'un nombre limité de chiffres avant et après la "virgule" (le point décimal).

**Donc aucun nombre irrationnel !**

**Approximation de $\pi$**

```
> pi
[1] 3.141593
> sprintf('%.17f',pi)
[1] "3.14159265358979312"
> typeof(pi)
[1] "double"
```

**Approximation de $\sqrt{2}$**

```
> sqrt(2)
[1] 1.414214
> sprintf('%.17f',sqrt(2) ** 2)
[1] "2.00000000000000044"
> sqrt(2) ** 2 == 2
[1] FALSE
```

Mais,
$\pi = 3.141592653589793\mathbf{23}\ldots$

# Nombres rationels

Les nombres rationels peuvent être représenté sous la forme d'une fraction, par exemple $\frac{1}{10}$.

- Le nombre $\frac{1}{10} = (0.1)_{10}$, par exemple, est simple dans le système décimal.
- Mais, il possède une infinité de chiffres après la virgule dans le système binaire !

$$0.0001100110011001100110011001100110011001100110011\ldots$$

Lorsque R affiche une valeur approchée, ce n'est qu'une approximation de la véritable valeur interne de la machine :

```
> 0.1 # quelle est la valeur de 0.1 ?
[1] 0.1 # ceci est une illusion !
```

La fonction `print` ou `printf` permet de voir (en décimal) la véritable représentation en machine de 0.1 qui n'est pas 0.1 mais :

```
> print(0.1,digits=17)
[1] 0.10000000000000001
```

# Représentation des nombres réels

- Les ressources d'un ordinateur étant limitées, on représente seulement un sous-ensemble des réels de cardinal fini.
- Ces éléments sont appelés nombres à virgule flottante.
- Leurs propriétés sont différentes de celles des réels.

## Problèmes et limitations

- les nombres et les calculs sont nécessairement arrondis.
- il y a des erreurs d'arrondi et de précision
- On ne peut plus faire les opérations de façon transparente

**Le zéro n'est plus unique !**

```
> 10^20 + 1 == 10^20
[1] TRUE
> 10^20 + 2 == 10^20
[1] TRUE
```

En math, il existe un unique nombre $y$ tel que $x + y = x$, le zéro !

# Égalité entre nombres flottants

Le calcul sur des nombres approchés étant par définition INEXACT, on évitera sous peine de surprises désagréables de questionner l'ÉGALITÉ en présence de nombres approchés !

```
> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 0.7
[1] TRUE
> 0.1 * 7 == 0.7
[1] FALSE
> 0.1 + 0.1 + 0.1 == 0.3
[1] FALSE
> 0.1 * 3 == 0.3
[1] FALSE
```

Le domaine du calcul approché est TRES difficile, et prévoir à l'avance le nombre exact de décimales correctes lors d'un résultat de calcul reste réservé aux spécialistes d'Analyse Numérique (brrr) ...

**Alors que faire ? Remplacer l'égalité par une précision h**

```
a == b # BAD !
```

```
(a - b) < h # GOOD !
```

# Autres problèmes avec les nombres flottants

## Une boucle infinie ?

```
x <- 1
while( x > 0 ) {
  print(x)
  x <- x / 2
}
```

Est-ce que cette boucle s'arrête ? En math ? En info ?

## Annulation catastrophique $x^2 - y^2$

```
> y <- 2**50
> x <- y + 1
> z1 <- x**2 - y**2 # appliquer directement la formule
> z2 <- (x - y)*(x + y) # appliquer une identité remarquable
> z2 - z1 # Est-ce que les résultats sont identiques ?
[1] 1
```

# Exemple : approximation de $\sqrt{r}$

Par la méthode des tangentes de Newton (1669).

Soit à calculer la racine carrée approchée d'un nombre réel $\texttt{r} > 0$, par exemple $\sqrt{2}$, sans utiliser `sqrt` !

**Newton**
Si $a$ est une approximation de $\sqrt{r}$ alors :

$$b = \frac{1}{2}(a + \frac{r}{a})$$

est une exception encore meilleure ! Pourquoi ? Cf TD.

Nous allons développer cet algorithme en répondant à trois questions :

**ITÉRATION** Comment améliorer l'approximation courante ?

**TERMINAISON** Mon approximation courante `a` est-elle assez bonne ?

**INITIALISATION** Comment initialiser la première approximation ?

# Algorithme d'approximation de $\sqrt{r}$

## ITÉRATION

Pour améliorer l'approximation, il suffit d'appliquer la formule de Newton, qui fait approcher `a` de $\sqrt{r}$ :

```
a = 0.5 * (a + r / a)
```

## TERMINAISON

Mon approximation courante `a` est-elle assez bonne ? Elle est assez bonne lorsque `a` est très proche de $\sqrt{r}$. Notons `h` la variable dénotant la précision, par exemple $\mathtt{h} = 2^{-20}$.

```
abs(a*a - r) < h
```

## INITIALISATION

Comment initialiser l'approximation ? En fait, les maths sous-jacentes à la technique de Newton montrent que n'importe quel réel $\mathtt{a} > 0$ convient :

```
a = 1
```

# Programme d'approximation de $\sqrt{r}$

```r
Racine <- function(r, h = 2**(-10)) {
  a <- 1
  while( abs(a*a -r) >= h) {
    print(a)
    a <- 0.5 * (a + r/a)
  }
  return(a)
}
```

```
> approx <- Racine(r = 2, h = 10**(-10))
[1] 1
[1] 1.5
[1] 1.416667
[1] 1.414216
> print(approx, digit = 15)
[1] 1.41421356237469
> print(sqrt(2), digit = 15)
[1] 1.4142135623731
```

**Observation**
La méthode de Newton converge rapidement vers le résultat.

# Mais d'où vient la formule de Newton ?

D'un simple calcul de tangentes (cf TD) ...

TODO Ajouter figure

# Le nombre $\pi$

- $\pi$ est défini comme le rapport constant entre la circonférence d'un cercle et son diamètre dans le plan euclidien.
- De nos jours, les mathématiciens définissent $\pi$ par l'analyse réelle à l'aide des fonctions trigonométriques elles-mêmes introduites sans référence à la géométrie.
- Le nombre $\pi$ est irrationnel, ce qui signifie qu'on ne peut pas l'écrire comme une fraction.
- Le nombre $\pi$ est transcendant ce qui signifie qu' il n'existe pas de polynôme à coefficients rationnels dont $\pi$ soit une racine.

# Calcul de $\pi$ par la formule de Leibniz

On utilisera la formule de Leibniz issue du développement en série de Taylor en 0 de *arctan*($x$) évalué au point 1 :

$$\sum_{k=0}^{+\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{9} + \frac{1}{11} + \ldots = \frac{\pi}{4}$$

Elle a été découverte en Occident au XVIIe, mais apparaît déjà chez Madhava, mathématicien indien de la province du Kerala, vers 1400.

c.f. Wikipedia

Nous allons développer un algorithme d'approximation de $\pi$.

**ITÉRATION** Comment améliorer l'approximation courante ?
**TERMINAISON** Mon approximation courante est-elle assez bonne ?
Est-ce que le calcul prend trop de temps ?
**INITIALISATION** Comment initialiser la première approximation ?

# Algorithme d'approximation de $\pi$

## ITÉRATION

Pour améliorer l'approximation, étant en possession de la somme acc des $i$ premiers termes, on voudra obtenir la somme des $i+1$ premiers. Il suffira donc d'incrémenter i, puis d'ajouter $\frac{(-1)^i}{2i+1}$ à acc.

```
i <- i + 1
term <- (-1)**i / (2*i + 1)
acc <- acc + term
```

## TERMINAISON

Mon approximation courante a est-elle assez bonne ? Elle est assez bonne lorsque je n'arrive plus à l'améliorer. Notons h la précision.

Est-ce que le calcul prend trop de temps ? Notons n le nombre maximum de termes à calculer.

```
abs(term) < h || i > n
```

## INITIALISATION

```
i <- 0
acc <- 1
```

# Programme d'approximation de $\pi$

```
LeibnizPi <- function(n = 10**4, h = 2^(-20)) {
  i <- 0
  term <- 1
  acc <- 1
  while( (i <= n) && 4*abs(term) > h) {
    i <- i + 1
    term <- (-1)**i / (2*i + 1)
    acc <- acc + term
  }
  return(4*acc)
}
```

```
> LeibnizPi(n = 100, h = 0)
[1] 3.131789
> LeibnizPi(n = 1000, h = 0)
[1] 3.140595
> LeibnizPi(n = 100000, h = 0)
[1] 3.141583
> pi
[1] 3.141593
```

## Observation

# Calcul de $\pi$ : analyse du programme

Calculons le temps nécessaire pour atteindre une précision donnée sans limiter le nombre d'itérations.

```
> system.time(LeibnizPi(n = Inf, h = 10**(-4)))
utilisateur     système      écoulé
      0.006       0.000       0.006
```

- Le temps d'exécution et le nombre d'itérations augmentent linéairement avec la précision.
- La recherche d'une estimation très précise de $\pi$ demande un temps de calcul important.
- En extrapolant ces résultats, il faudrait $5 \times 10^8$*secondes* ($\geq 15$ ans) pour obtenir une estimation de $\pi$ à la précision machine (approximativement 15 décimales).
- Certaines formules convergent beaucoup plus rapidement.

| Précision | Temps (s) |
|---|---:|
| $10^{-4}$ | 0.006 |
| $10^{-5}$ | 0.147 |
| $10^{-6}$ | 0.599 |
| $10^{-7}$ | 5.347 |
| $10^{-8}$ | 52.860 |

## Calcul de $\pi$ : optimisation du programme

- Les multiplications, divisions, et puissances sont plus coûteuse en temps de calcul que les additions et soustractions.
- Exploitons la récurrence pour accélerer les calculs.

```
LeibnizPi2 <- function(n = 10**4, h = 2^(-20)) {
  i <- 0
  term <- 1
  acc <- 1
  h <- h / 4 ## éviter la multiplication du test
  sign <- 1 ## mémoriser le signe du terme
  denom <- 1 ## mémoriser le dénominateur du terme
  while( (i <=n) && abs(term) > h) {
    i <- i + 1
    sign <- -1 * sign # éviter une puissance
    denom <- denom + 2 # éviter une multiplication
    term <- sign / denom
    acc <- acc + term
  }
  return(4*acc)
}
```

# Calcul de $\pi$ : comparaison de programmes

| Précision | Temps (s) |
|---|---|
| `LeibnizPi` | 5.347 |
| `LeibnizPi2` | 4.157 |
| En langage C | 0.007 |

- Les optimisations du programme offrent un gain supérieur à 20%.
- R est donc un langage interprété de haut niveau ce qui se paie au niveau des performances.
- Le langage C, entre autres, est beaucoup plus rapide.
- Le langage C est un langage impératif, généraliste et de bas niveau où chaque instruction du langage est compilée.

**Questions?**

**Retrouvez ce cours sur le site web**

`www.i3s.unice.fr/~malapert/R`