Document de révision pour Méthodes numériques

Gabriel Crépeault-Cauchon

5 avril 2018

Table des matières

1 Simulation

1.1 Génération de nombres aléatoires uniformes

1.1.1 Opérateur modulo

Tout d'abord, il est important de comprendre ce que signifie $b \mod m$

$$a \equiv b \mod m \Leftrightarrow \frac{a-b}{m} \equiv k \quad , k \in \mathbb{Z}$$

Voici un exemple numérique pour mieux comprendre. $b \mod m$ s'écrit `b%%m` dans R avec l'opérateur `%%` :

```
15 %% 6 # le reste pour faire une division entière
```

```
## [1] 3
```

```
15/6 # il reste 2.5 ...
```

```
## [1] 2.5
```

```
floor(15/6) * 6 # 6 rentre 2 fois dans 15 ...
```

```
## [1] 12
```

```
15 - floor(15/6)*6 # il va rester 3
```

```
## [1] 3
```

On peut donc dire que

$$a = b \mod m \Leftrightarrow a = b - \left\lfloor \frac{b}{m} \right\rfloor m$$

1.1.2 Générateur congruentiels linéaires

Un bon générateur de nombres aléatoires (congruentiels) doit répondre à 4 critères pour être efficace :

- Doit produire des nombres distribués approximativement uniformément ;
- Les nombres doivent être indépendants
- Possède une période suffisament longue (*au moins* 2^{60}) ;
- Le générateur doit être facilement reproduisible, lorsqu'on connaît le point de départ (*seed*)

Le générateur congruentiel linéaire est défini par la formule suivante pour générer des nombres $U(0,1)$:

$$x_i = (ax_{i-1} + c) \mod m$$

a le multiplicateur. a doit être une racine primitive de m, soit le plus petit entier k satisfaisant $1 = a^k \mod m$.

c l'incrément

m le module

x_0 l'armorce (*seed*)

L'algorithme

```r
SimCongru <- function(nsim, a = 7^5, c, m = 2^(31) - 1, seed)
{
    # les chiffres par défaut pour a et m sont des choix populaires
    x    <- numeric(nsim + 1)
    x[1] <- seed
    for (i in seq(nsim))
    {
        x[i + 1] <- (a * x[i] + c) %% m
    }
    x[-1] / m   # enlève le seed et on divise par m
                # pour avoir des nombres suivant une U(0,1)
}
# Test
SimCongru(nsim = 10,c = 100, seed = 2345)
```

```
##  [1] 0.01835288 0.45689588 0.04904205 0.24965308 0.91927864 0.31612259
##  [7] 0.07237431 0.39497196 0.29375428 0.12816416
```

1.1.2.1 Maximiser l'efficacité du générateur

On maximise la période du générateur si

- m est un nombre premier
- a est une *racine primitive* de m

1.1.3 Générateur de Excel

Encore aujourd'hui, on est incapable de savoir le fonctionnement exacte du générateur. Si on programme en VBA, on suggère de ne pas utiliser la fonction interne `RND()`, mais plutôt faire appel à la fonction interne `RAND()`, qui est meilleure.

1.1.4 Simulation uniforme dans R

Il existe quelques fonctions, notamment :

```r
runif(15, min = 0, max = 3) # min = a, max = b
```

```
##  [1] 1.8968495 0.2063186 0.2246975 0.6268798 2.4218879 2.8853796 2.9531956
##  [8] 0.6568298 0.9116116 0.3830123 0.6580184 0.7433501 1.9665432 1.6226223
## [15] 0.6091336
```

```r
set.seed(101) # fixe l'amorce. Permet de retrouver la même simulation.
```

1.2 Simulation pour des loi non-uniformes

1.2.1 Rappels sur les transformations de v.a

Il y a 3 techniques utilisées :

1. Avec la fonction de répartition ;
2. avec un changement de variable. Dans le contexte continu, on utilise la méthode du Jacobien (univarié) vu dans le cours ACT-1003 :

(a) **Cas univarié** ($Y = g(X)$)

$$f_Y(y) = f_X(g^{-1}(y))|g^{-1}(y)'| \tag{1}$$

Exemple

Soit $Y = -2\ln(X)$, avec $X \sim U(0,1)$. On veut trouver la distribution de Y

On commence par trouver $g^{-1}(y)$

$$\begin{aligned} Y &= g(X) \\ y &= -2\ln(x) \\ \frac{-y}{2}&\ln(x) \\ g^{-1}(y) &= e^{-\frac{y}{2}} \end{aligned}$$

Puis on trouve $g^{-1}(y)'$,

$$g^{-1}(y)' = -\frac{1}{2}e^{-\frac{y}{2}}$$

Ensuite, on applique la transformation :

$$\begin{aligned} f_Y(y) &= f_x(g^{-1}(y))\left|g^{-1}(y)'\right| \\ &= f_x(e^{-\frac{y}{2}})\left|-\frac{1}{2}e^{-\frac{y}{2}}\right| \\ &= \underbrace{\frac{1}{1-0}}_{X\sim U(0,1)}\ \frac{1}{2}e^{-\frac{y}{2}} \end{aligned}$$

En compensant avec des constantes, on retrouve rapidement la forme d'un Gamma :

$$f_Y(y) = \frac{\left(\frac{1}{2}\right)^1 y^{1-1}e^{-\frac{y}{2}}}{\Gamma(1)}$$

On peut conclure que $Y \sim Gamma(1, \frac{1}{2})$.

(b) **Cas multivarié** (plus précisément, le bivarié). On connait les v.a. X_1 et X_2 et on cherche la fonction conjointe de f_{Y_1,Y_2} où $Y_1 = u_1(x_1, x_2)$ et $Y_2 = u_2(x_1, x_2)$.

$$f_{Y_1,Y_2} = f_{X_1,X_2}(u_1^{-1}(y_1,y_2), u_2^{-1}(y_1,y_2))\left|\begin{matrix} \frac{\partial u_1^{-1}}{\partial y_1} & \frac{\partial u_1^{-1}}{\partial y_2} \\ \frac{\partial u_2^{-1}}{\partial y_1} & \frac{\partial u_2^{-1}}{\partial y_2} \end{matrix}\right| \tag{2}$$

Note : une table des principales règles de dérivation se trouve à l'annexe C à la page 30.

3. en posant la fonction génératrice des moments ou la transformée de Laplace.

1.2.2 Méthode de l'inverse

Cette méthode, basée sur le théorème de la fonction quantile, est très simple à utiliser lorsqu'on peut facilement inverser la fonction de répartition F_X pour obtenir la fonction quantile F_X^{-1}. En effet,

$$F_X(X) \backsim U(0,1)$$

$$F_X^{-1}(U) \backsim X$$

1.2.2.1 avec la fonction quantile

1. Obtenir un nombre u d'une loi $U(0,1)$
2. poser $X = F_X^{-1}(u)$

```
SimExp <- function(n, lam)
{
    # Simulation de nombres aléatoires d'une Exp(Lambda)
    u <- runif(n)
    # On trouve la forme de la fonction quantile
    -log(1 - u) / lam
}
# Test
SimExp(10, 1/40)
```

```
##  [1] 18.621242  1.792565 49.471414 42.881587 11.499589 14.270131 35.166217
##  [8] 16.226634 38.915709 31.571221
```

Note : on aurait pu utiliser directement la fonction intégrée de R `qexp()` et l'évaluer aux points u trouvés.

Contexte discret

Dans un contexte discret, on construit fonction de répartition de probabilité, i.e

$$F_X(x) = \begin{cases} 0,1x & x \leq 10 \\ 1 & x > 10 \end{cases}$$

1.2.2.2 avec `uniroot()`

Si jamais il est trop difficile d'isoler x pour trouver la fonction quantile, on procède par optimisation en posant une fonction égale à 0 (et on cherche les racines) :

```
SimUnirGamma <- function(n, alph, lam)
{
    x <- numeric(n)
    for (i in seq(n))
    {
        # On simule encore un nombre u de U(0,1)
        u <- runif(1)
        # On va poser F_X(x) - u = 0, on va donc trouver la     valeur de x qui se rapproche le plus
        f <- function(x) (pgamma(x, alph, lam) - u)
        x[i] <- uniroot(f, c(0, 100000))$root
    }
    x # on renvoi le vecteur de réalisations
}
```

```r
# Test
SimUnirGamma(10, 3, 1/40)
```

```
##  [1] 202.14661 146.20052 152.21554 234.52207  99.83380 122.43692 177.90207
##  [8]  65.15326  93.16458  29.42252
```

1.2.2.3 avec `optimize()`

`Optimize()` cherche le minimum (ou le maximum) de la fonction évaluée. Alors, on doit poser le même algorithme qu'avec `Uniroot()`, sauf que notre fonction doit être en valeur absolue :

** à faire plus tard**

```r
SimOptimGamma <- function(n, alph, lam)
{
    x <- numeric(n)
    for (i in seq(n))
    {
    # On simule encore un nombre u de U(0,1)
        u <- runif(1)
    # On va poser f = abs(F_X(x) - u), on va donc trouver     la valeur de x qui minimise f
        f <- function(x) pgamma(x, alph, lam) - u
        x[i] <- optimize(f, c(0, 99999999999))$minimum
    }
    x # on renvoi le vecteur de réalisations
}
# Test
SimOptimGamma(10, 3, 1/40)
```

```
##  [1] 1e+11 1e+11 1e+11 1e+11 1e+11 1e+11 1e+11 1e+11 1e+11 1e+11
```

1.2.3 Méthode acceptation-rejet

Lorsqu'on a une fonction de densité particulière, on peut l'englober d'une autre fonction, $cg_Y(y)$. Par la suite, on simule des nombres u d'une loi $U(0,1)$ et on vérifie si le ratio $\frac{f_X(y)}{cg_Y(y)}$ [1] est respecté : si oui, on accepte le point. Sinon, on repart l'algorithme.

1.2.4 Fonctions internes de simulation

Excel

Il existe des fonctions de simulation pour les principales lois de probabilité dans Excel. On peut trouver en annexe A un tableau récapitulatif.

R

Il existe des fonctions de simulation pour les principales lois de probabilité dans R avec le package *actuar*. On peut trouver en annexe B un tableau récapitulatif.

1. Une autre façon de voir cet algorithme est de simuler un nombre uniforme $U(0,1)$, puis de valider si $u \cdot cg_Y(y) \leq f_X(y)$. On accepte les points sous la courbe de $f_X(x)$ et on rejete les autres points.

1.3 Simulation de modèles actuariels

1.3.1 Mélanges discrets (*mixture*)

Il existe déjà une fonction R du package actuar. Toutefois, il est important de comprendre la logique de l'algorithme en arrière :

```
MixtExp <- function(nsim, p, lam1, lam2)
{
    x <- numeric(nsim)
    for (i in seq(nsim))
    {
    u <- runif(1)
    x[i] <- ifelse(u <= p,
                   rexp(1, lam1), # si vrai
                   rexp(1, lam2) # si faux
                   )
    }
    x
}
MixtExp(10, 0.35, 1/40, 1/600)
```

```
##  [1]  586.25098   67.66620   85.99436   12.05970  614.99543  861.55740
##  [7] 1280.60837  281.22749  340.54286   22.44438
```

C'est sur cet algorithme que se base `rmixture()` :

```
rmixture(10, # nombre de simulations
         0.35, # probabilités 1ère loi vs 2e
         expression(rexp(1/40), rexp(1/600)))
```

```
##  [1]   7.0620520  25.1671734  24.8035957   8.7948722   0.5290429
##  [6]  36.5387274  97.4917096 235.6847265 385.5614156 487.8231587
```

```
# Remarque : on doit mettre nos fonction random dans expression()

rmixture(10,
         c(30,4,5), # R la fonction calcule les proportions
         expression(rgamma(2, 1/40),
                    rexp(1/200),
                    rpareto(1.5, 100000)))
```

```
##  [1]  28.69218  85.96322 102.89362 127.67797  29.55280  16.88010 124.46835
##  [8]  41.42751 102.30706  19.55251
```

1.3.2 Mélanges continus et modèles hiéarchiques

Il arrive qu'on doivent travailler avec une distribution qui est conditionné par l'un de ces paramètres, lui-même aléatoire selon une loi connue. À la main, ça peut rapidement devenir compliqué : dans R, il suffit de simuler le même nombre de paramètres que l'on a besoin pour notre distribution principale. Voici un exemple :

$$X|\beta \backsim Beta(1, \beta)$$
$$\beta \backsim poisson(\lambda = 4)$$

Alors,

```
rbeta(10, 1, rpois(10, 4))
```

```
##  [1] 0.11990551 0.03773443 0.10957311 0.22217912 0.36997161 0.26502854
##  [7] 0.01693514 0.13656607 0.52287890 0.83264011
```

1.3.3 Distributions composés

Ces distributions vont revenir dans le cours **ACT-2001**, alors qu'elles sont beaucoup utilisées dans les modèles *fréquence-sévérité*. Encore une fois, une fonction a été créée dans le package `actuar`, voici l'idée en arrière :

```
FreqPois_SecGamma <- function(nsim, lampois, alph, beta)
{
    Sn <- numeric(nsim) # résultat final qu'on va vouloir
    for (i in seq(nsim))
    {
        n <- rpois(1, lampois)
        x <- rgamma(n, alph, beta)
        Sn[i] <- sum(x)
    }
    Sn
}
FreqPois_SecGamma(10, lampois = 8, alph = 2, beta = 1/400)
```

```
##  [1] 11719.869 13459.843  3249.322  7067.283  5892.603  4577.540  2781.835
##  [8]  4263.551  1360.321  7741.510
```

La fonction `rcompound()` (plus générale) ou bien `rcompois()` nous permet de faire le même genre de simulation :

```
library(actuar)
rcompound(n = 5, # nombre de simulations
          model.freq = rbinom(1000, 0.48), # modèle fréquence
          model.sev = rnorm(1000, 15)) # modèle sévérité
```

```
## [1] 505521.9 458950.0 494821.0 462325.6 515891.9
```

```
rcomppois(n = 5, # nombre de simulations
          lambda = 5, # paramètre lambda de la poisson composée
          model.sev = rgamma(5, 1/1000))
```

```
## [1] 40503.47 11287.25 19861.82 26383.86 28554.08
```

1.4 Intégration Monte Carlo

Lorsqu'on veut intégrer une fonction difficile à intégrer papier, on peut l'approximer avec la technique d'intégration Monte Carlo. Il existe d'autres techniques d'approximation qui seront vue plus en détail dans le chapitre 4

1.4.1 Exemple

Disons que l'on veut calculer l'intégrale suivante :

$$\int_4^{15} (x+4)^2 dx \tag{3}$$

On cherche donc à obtenir la valeur θ, qui représente le résultat de l'évaluation de l'intégrale (3).

Objectif : transformer notre fonction $g(x) = h(x)f(x)$ où $f(x)$ est une fonction de densité d'une loi connue. Puisqu'on a 2 bornes finies, on va utiliser la loi uniforme. Alors,

$$\begin{aligned}
\theta &= \int_4^{15} (x+4)^2 dx \\
\theta &= (15-4) \underbrace{\int_4^{15} (x+4)^2 \frac{1}{15-4} dx}_{X \sim U(4,15)} \\
&= (15-4) E[(X+4)^2]
\end{aligned}$$

On peut simuler facilement θ sous la forme suivante :

$$\theta = \frac{11}{n} \sum_{i=1}^{n} (x+4)^2$$

En posant $n = 10\hat{}\{4\}$ dans notre simulation, on obtient le résultat suivant :

```
x <- runif(n, 4, 15)
mean((x+4)^2) * 11 # valeur estimée
```

```
## [1] 2118.112
```

```
quad(function(x) (x+4)^2, 4, 15) # valeur exacte
```

```
## [1] 2115.667
```

Évidemment, notre niveau de précision augmente avec $n \to \infty$.

1.4.1.1 Astuce

Pour reproduire $n = 3$ simulations d'une distribution $m = 3$ fois, on peut utiliser la fonction `replicate()` de R :

```
x <- replicate(5, rnorm(3)) # créé une matrice 3x5 de nombres N(0,1)
# on veut calculer l'espérance de la statistique d'ordre Y_2
mean(apply(x, # matrice sur laquelle on applique la fonction
           2, # 2e dimension (colonne)
           function(x) sort(x)[2] # fonction à appliquer
           ))
```

```
## [1] 0.1780995
```

2 Arithmétique des ordinateurs

2.1 Définitions et notation

2.1.1 Bases de conversion

Un nombre peut être représenté dans plusieurs bases, voici quelques exemples :

Décimal la base 10
Binaire la base 2
Octal la base 8
Hexadécimal la base 16

2.1.2 Unités de mesure

Les ordinateurs fonctionnent en base 2 (binaire), Il y a un vocabulaire à connaître pour les différentes unités (différents du système métrique, car ce n'est pas la même base) :

Unité	Symbole	Capacité
bytes	b	1 b
octet	o	8 b
kilooctet	ko	1 024 o
mégaoctet	Mo	1 048 576 o
gigaoctet	Go	1 073 741 824 o

2.2 Conversion de base

2.2.1 Conversion de la base 10 vers une base b

</> Algorithme

On veut convertir le nombre x_{10} vers la base b, soit x_b :

1. On pose $v = \lfloor x \rfloor$ [a] et $i = 0$ (notre compteur)
2. On répète les étapes suivantes jusqu'à temps que $v = 0$
   (a) $d_i = v \mod b$
   (b) Trouver x_i, le symbole correspondant dans la base b de d_i [b]
   (c) $v = \left\lfloor \frac{v}{b} \right\rfloor$
   (d) $i = i + 1$
3. $x_b = x_m ... x_2 x_1 x_0$

a. Dans le cas où notre nombre à convertir est fractionnaire, on commence avec $v = \lfloor b^n x \rfloor$ et on déplace la virgule de n positions vers la gauche pour n chiffres maximum dans la partie fractionnaire.
b. d_i sera différent de x_i pour des bases supérieures à 10 seulement.

Exemple

On veut convertir le nombre 343,16 en **base 6**, avec 3 chiffres après la virgule.

1. On commence par multiplier le nombre à convertir par b^3 :

$$343,16 \times 6^3 = 74122,56$$

2. On pose $v = \lfloor 74\ 122,56 \rfloor = 74\ 122$ et on applique l'algorithme :

$$\begin{aligned} 74122 \div 6 &= 12353 && \text{reste } 4 \\ 12353 \div 6 &= 2058 && \text{reste } 5 \\ 2058 \div 6 &= 343 && \text{reste } 0 \\ 343 \div 6 &= 57 && \text{reste } 1 \\ 57 \div 6 &= 9 && \text{reste } 3 \\ 9 \div 6 &= 1 && \text{reste } 3 \\ 1 \div 6 &= 0 && \text{reste } 1 \end{aligned}$$

3. en rassemblant les modulos, on obtient le nombre 1 331 054. Il reste à déplacer la virgule vers la gauche de 3 positions, dû à la précision désirée pour représenter le chiffre :

$$341,16_6 = 1\ 331,054$$

Pour pouvoir se pratiquer, on peut utiliser le site web Base Convert pour se valider et passer d'une base à l'autre.

2.2.2 Conversion d'une base b vers la base 10

Algorithme

1. On représente le nombre en base b à convertir en forme *explosée* :

$$x_{10} = x_{m-1} \times b^{m-1} + ... + x_1 \times b^1 + x_0 \times b^0 + \underbrace{x_{-1} \times b^{-1} + x_{-2} \times b^{-2} + ... + x_{-n} \times b^{-n}}_{\text{partie fractionnaire}}$$

2. La conversion est le résultat du calcul ci-haut.

Exemple

On veut convertir le nombre binaire 1000011,100101 en nombre décimal (base 10)

$$\begin{aligned} 1000011,100101 &= (1\times 2^6 + 0\times 2^5 + ... + 1\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + ... + 1\times 2^{-4} \\ &\quad + 0\times 2^{-5} + 1\times 2^{-6})_{10} \\ &= 67,578125 \end{aligned}$$

2.2.3 Conversion vers des bases générales

Les cas les plus fréquents sont les **conversions de jours en minutes ou heures**, ou bien **trouver la position exacte d'un élément dans une matrice** (selon une disposition par ligne ou par colonne).

Voici un exemple simple pour convertir le nombre 91 492 en nombres de jours, heures, minutes et secondes :

TAB. 4.3 – Conversion du nombre décimal 91 492 dans la base générale [365 24 60 60].

i	v	b_i	$\lfloor v/b_i \rfloor$	$v \bmod b_i$	x_i
0	91 492	60	1 524	52	52
1	1 524	60	25	24	24
2	25	24	1	1	1
3	1	365	0	1	1

2.2.3.1 Cas spécifique des tableaux

Pour trouver la position du t^e élément d'une matrice $m \times n$, on procède ainsi :

$$\begin{aligned} (t-1) \div n &= i \text{ reste } x_0 \\ i \div m &= 0 \text{ reste } x_1 \end{aligned}$$

Les coordonnées du t^e élément sont $\left[x_0 + 1, x_1 + 1\right]$

Fonction arrayInd de R : la fonction `arrayInd()` dans R permet d'effectuer ce calcul. Petit exemple rapide avec une matrice 5×10 qu'on veut connaître les coordonnées du 47^e élément (matrice remplie par ligne) :

```
arrayInd(47, c(5,10))
```

```
##      [,1] [,2]
## [1,]    2   10
```

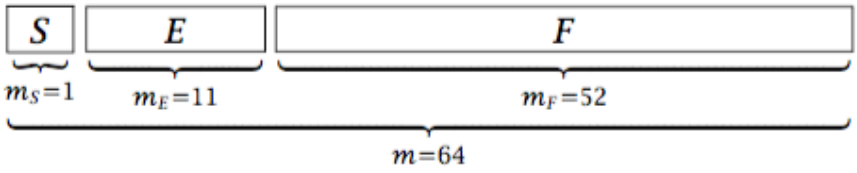

FIG. 4.1 – Représentation schématique d'un nombre en double précision dans la norme IEEE 754.

FIGURE 1 –

2.3 Représentation en virgule flottante

Note : le site web a aussi conçu un Convertisseur vers la représentation IEEE$_{754}$en virgule flottante.

La représentation en virgule flottante se représente avec l'équation suivante :

$$x = (-1)^S \times B^{E-e} \times M \tag{4}$$

où

S byte du signe (*bit fort*) ;

B base de représentation ($B = 2$ selon la norme IEEE$_{754}$) ;

E bytes de l'exposant ;

e décalage de l'exposant

M ensemble de bytes pour la partie fractionnaire (représenté sous la forme 1,F)

Puisque R travaille avec des nombres *double*, on doit savoir les caractéristiques reliées à un nombre en double précision :

- le nombre est stocké dans un mot de $m = 64$ bytes
- un byte est réservé pour le signe ($m_S = 1$)
- 11 bytes sont réservés pour l'exposant ($m_E = 11$)
- et finalement 52 bytes ($m_F = 52$) sont réservés pour la partie fractionnaire d'un nombre.
- le décalage (e) est toujours égal à 1023.
- La valeur que peut prendre l'exposant (non-décalé) va de 0 à 2047 (mais 0 et 2047 sont réservés pour certains cas de le norme IEEE$_{754}$). Alors **l'exposant décalé prend toujours une valeur entre -1022 et 1023**.

Comme il a été vu à la section 2.2, on va devoir faire une conversion de l'équation (4) vers la représentation de la norme IEEE$_{754}$en passant de nombres décimaux vers des nombres binaires :

Méthode de conversion

Soit x en décimal que l'on veut convertir selon la norme IEEE_{754} :

1. On veut trouver la **valeur des bytes de l'exposant** pour s'approcher le plus possible de la valeur de x :

$$\left\lfloor \frac{\log(x)}{\log(2)} \right\rfloor + e = E$$

2. Trouver la valeur de la mantisse $M = 1, F$ en convertissant x en base 2, puis en décalant la virgule de $E - e$ positions :

$$\frac{x_2}{10^{E-e}} = 1, F$$

3. Alors, on obtient le nombre x par :

$$x = (-1)^S \times 2^{E-e} \times 1, F$$

Note : il peut être demandé de convertir la représentation en binaire. Il suffit de respecter le nombre de bytes par élément (S, E ou F) selon la norme IEEE_{754}.

Exemple

On veut convertir le nombre décimal 256,50 en représentation IEEE_{754}

On commence par écrire le nombre 256,50 sous la représentation en virgule flottante :

$$\begin{aligned} 256,50 &= (-1)^0 \times 2^8 \times 1,5 \\ &= (-1)^0 \times 2^{1031-1023} \times 1,5 \end{aligned}$$

Si on convertit vers la représentation IEEE_{754} :

0	10000000111	00...101

2.3.1 Caractéristiques importantes

Il est à noter que l'on peut retrouver les caractéristiques mentionnées ci-dessous avec l'appel de fonction `.Machine` dans R.

1. le plus grand nombre que l'on peut représenter avec la double précision 64-bits est :

$$x_{\max} = 1,797693 \times 10^{308}$$

2. le plus petit nombre que l'on peut représenter est

$$x_{\min} = 2,225074 \times 10^{-308}$$

Variante : le plus petit nombre *dénormalisé* est

$$x_{\max} = 4,940656 \times 10^{-324}$$

3. On a exactement la même étendue pour les nombres négatifs (c'est juste le premier bytes qui est égal à 1)

4. l'epsilon de la machine (plus petite valeur tel que $1 + \epsilon \neq 1$) :

$$\epsilon = 2^{-52} = 2,220446 \times 10^{-16}$$

5. tous les nombres qui sont dans l'intervalle $[1, 1 + \epsilon)$ sont représenté par 1 dans l'ordinateur
6. Le nombre qui vient tout de suite après $x_{\min}$ est $x_{\min} + 2^{-1074}$. À l'autre bout du spectre, le nombre qui vient tout juste avant $x_{\max}$ est $x_{\max} - 2^{971}$
7. (a) Si un calcul résulte en $x < x_{\min}$, alors le système renvoie '-Inf' car il y a un *souspassement de capacité.*
   (b) Si un calcul résulte en $x > x_{\max}$, alors le système renvoie 'Inf' car il y a un *dépassement de capacité.*

2.3.2 Erreur d'arrondi (principes de programmation)

Afin d'éviter des erreurs d'arrondi, il y a 6 principes de programmation qui permettent de diminuer les erreurs d'arrondi :

1. L'addition et la soustraction en virgule flottante ne sont pas associatives. **astuce** : additionner les nombres en ordre croissant de grandeur
2. Éviter de soustraire un petit nombre d'un grand (ou sinon le faire le plus tard possible dans l'opétation)
3. Si on a une somme alternée à calculer, additionner tous les nombres positifs et négatifs séparémment, puis on soustrait l'un à l'autre.
4. Multiplier et diviser des nombres d'un même ordre de grandeur.
5. Chercher des formules mathématiques équivalentes pour nous éviter de calculer de très grands ou très petits nombres.
6. Travailler en échelle logarithmique. Exemple :

$$x \times y = e^{\log x + \log y}$$

2.3.3 Coûts des opérations

Comme on peut le voir dans le tableau ci-dessous, certains types d'opération arithmétiques sont plus coûteuses que d'autres :

TAB. 4.6 – Coût relatif de quelques opérations en virgule flottante

Opération arithmétique	Coût relatif
Addition et soustraction	1,0
Multiplication	1,3
Division	3,0
Racine carrée	4,0
Logarithme	15,4

2.3.4 Codage de caractères

— **Premier standard** : Code américain normalisé pour l'échange d'information (ASCII). Ce code contenait 128 caractères. Le code tient sur 7 bytes, mais un byte additionnel a été ajouté pour l'ajout de 128 caractères additionnels.
— **Nouvelle norme**, une nouvelle norme est apparue, ISO 8859-1. Cet encryptage permet d'accueillir plus de caractères, comme des accents de plusieurs langues.
— **Unicode** : Avec les années, on a voulu intégrer les symbôles asiatiques (il fallait alors des milliers de caractères différents), mais qui fonctionnait sur 8 bytes : L'encodage UTF-8 a été adopté.

3 Optimisation numérique

3.1 Méthode de bissection

3.1.1 Idée de la méthode

On veut trouver le x^* où $f(x^*) = 0$.

3.1.2 Algorithme

```
bissection <- function(FUN, lower, upper, TOL = 1E-6,
                       MAX.ITER = 100, echo = FALSE)
{
    # Cas triviaux où une borne est la solution
    if (identical(FUN(lower), 0))
        return(lower)
    if (identical(FUN(upper), 0))
        return(upper)

    # Bornes de départ inadéquates
    if (FUN(lower) * FUN(upper) > 0)
        stop('FUN(lower) and FUN(upper) must be of opposite signs')

    # On commence les itérations :
    x <- lower
    i <- 1

    repeat
    {
        xt <- x
        x <- (lower + upper)/2
        fx <- FUN(x)

        if (echo) # à afficher seulement si on veut suivre les étapes
            print(c(lower = lower, upper = upper, x = x, f_x = fx))

        # Test pour valider si notre x est assez précis
        if (abs(x - xt)/abs(x) < TOL)
            break

        # test pour valider si on a fait trop d'itérations
        if (MAX.ITER < (i <- i + 1))
            stop('Maximum number of iterations reached
                  without convergence')

        # Positionnement du x trouvé (lower ou upper?)
        if (FUN(lower) * fx > 0)
            lower <- x
        else
            upper <- x
    }
    # Le résultat appparait comme dans les fonctions uniroot et optimize
```

```
    list(root = x, nb.iter = i)
}
```

3.2 Méthode du point-fixe

3.2.1 Idée de la méthode

On veut trouver le x^* où $f(x^*) = x^*$.

3.2.2 Algorithme

```
pointfixe <- function(FUN, start, TOL = 1E-6, MAX.ITER = 100,
                      echo = FALSE)
{
    x <- start

    # option pour suivre chaque itérations
    if (echo)
        expr <- expression(print(xt <- x))
    else
        expr <- expression(xt <- x)

    i <- 0

    repeat
    {
        eval(expr)

        x <- FUN(xt)

        if (abs(x - xt)/abs(x) < TOL)
            break

        if (MAX.ITER < (i <- i + 1))
            stop('Maximum number of iterations reached
                  without convergence')
    }
    list(fixed.point = x, nb.iter = i)
}
```

3.2.3 Alternative

Dans le cas où on cherche x^* où $f(x^*) = 0$, on peut plutôt optimiser la fonction $g(x)$, où $g(x) = x - f(x)$.

3.3 Méthode de Newton-Raphson

3.3.1 Idée de la méthode

Ressemble à la méthode du point-fixe, sauf qu'on défini notre nouveau $\tilde{x}$ dans les itérations en trouvant le point $\hat{x}$ où $f'(\hat{x}) = 0$. En effet,

$$
\begin{aligned}
f'(x) &= \frac{f(\tilde{x}) - 0}{\tilde{x} - \hat{x}} \\
x^* &\approx \hat{x} \\
&\approx \tilde{x} - \frac{f(\tilde{x})}{f'(\tilde{x})}
\end{aligned}
$$

3.3.2 Algorithme

```
nr <- function(FUN, FUNp, start, TOL = 1E-6,
               MAX.ITER = 100, echo = FALSE, ...)
{
    x <- start

    # option pour suivre chaque itérations
    if (echo)
        expr <- expression(print(xt <- x))
    else
        expr <- expression(xt <- x)

    i <- 0

    repeat
    {
        eval(expr)

        x <- xt - FUN(xt, ...)/FUNp(xt, ...)

        if (abs(x - xt)/abs(x) < TOL)
            break

        if (MAX.ITER < (i <- i + 1))
            stop('Maximum number of iterations reached
                  without convergence')
    }
    list(root = x, nb.iter = i)
}
```

3.3.3 Méthode de la sécante

Lorsqu'il n'est pas possible de trouver la dérivée de $f(x)$, alors on peut utiliser une technique modifiée au lieu de *Newton-Raphson* : la méthode de la sécante.

3.3.3.1 Idée de la méthode

```
secante <- function(FUN, x0, x1,
                    TOL = 1E-06, MAX.ITER = 100)
{
    x <- c(x0, x1, 0)
```

```
    i <- 1

    repeat
    {

        x[3] <- x[2] - (FUN(x[2])*(x[2] - x[1])) / (FUN(x[2]) - FUN(x[1]))

        ## test pour voir si on accepte le x trouvé
        if ((abs(x[3] - x[2]) / abs(x[3])) < TOL)
            break

        ## test du nombre d'itérations
        if (i >= MAX.ITER)
            stop("Nombre maximal d'itérations atteint.")

        ## On prépare les variables pour les prochaines itérations
        i <- i + 1
        x[1] <- x[2]
        x[2] <- x[3]
    }

    list(root = x[3], nb.iter = i)
}
```

3.4 Fonctions R pour optimisation

Il existe plusieurs fonctions dans R qui permettent de faire de l'optimisation selon le contexte.

Fonction R	packages	description
`uniroot()`	de base	Permet de trouver la racine d'une fonction ($f(x) = 0$)
`optimize()`	de base	Permet de trouver le maximum ou le minimum d'une fonction $f(x)$
`nlm()`	de base	utile pour estimation des paramètre avec la méthode MLE $f(x)$
`optim()`	de base	fonction tout usage de R pour optimiser. Plusieurs des fonctions d'optimisation sont dérivés de celle-ci.

4 Intégration numérique

4.1 Polynôme d'interpolation de Lagrange

On utilise le polynôme d'interpolation de Lagrange pour approximer des fonctions, un peu comme on le faisait avec le développement de Taylor.

$$P_n(x) = \sum_{k=0}^{n} f(x_k) L_k(x) \tag{5}$$

où

$$L_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}$$

4.1.1 Fonction R pour l'approximation de polynôme

Il existe une fonction R pour développer un polynôme et l'évaluer en un point.

```
library(polynom) # package nécessaire
```

```
##
## Attaching package: 'polynom'
```

```
## The following object is masked from 'package:pracma':
##
##     integral
```

```
x <- c(2, 2.75, 4)                  # noeuds
(P1 <- poly.calc(x[1:2], 1/x[1:2])) # polynôme de degré 1
```

```
## 0.8636364 - 0.1818182*x
```

```
(P2 <- poly.calc(x, 1/x))  # polynôme de degré 2
```

```
## 1.113636 - 0.3977273*x + 0.04545455*x^2
```

4.2 Méthode du point-millieu

$$\int_a^b f(x)dx \approx 2h \sum_{j=0}^{n-1} f(x_{2j+1}) \tag{6}$$

où $h = \frac{b-a}{2n}$ et $x_j = a + jh$.

4.2.1 Code R

```r
PointMillieu <- function(FUN, a, b, n = 2)
{
  # Identifier h
  j <- seq(0, n-1)
  h <- (b-a) / (2 * n)

  ## Vecteur de X_j
  x <- a + (2*j  +1) * h

  ## Calcul du polynôme
  2 * h * sum(FUN(x))
}
PointMillieu(dnorm, a = -1.645, b = 1.645, n = 2)
```

```
## [1] 0.9358462
```

4.3 Méthode du trapèze

La méthode du trapèze se base sur l'approximation de la fonction par un polynôme de Lagrange du 1er degré ($m = 1$). Alors,

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right] \tag{7}$$

où $h = x_1 - x_0$ et $x_j = a + jh$.

4.3.1 Code R

4.4 Méthode de Simpson

La méthode de Simpson se base sur l'approximation de la fonction par un polynôme de Lagrange du 2e degré ($m = 2$). Alors,

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_{2j}) + 4\sum_{j=1}^{n} f(x_{2j-1}) + f(b)\right] \tag{8}$$

où $h = x_1 - x_0$ et $x_j = a + jh$.

4.4.1 Code R

4.5 Méthode de Simpson $\frac{3}{8}$

La méthode de Simpson $\frac{3}{8}$ se base sur l'approximation de la fonction par un polynôme de Lagrange du 3e degré ($m = 3$). Alors,

$$\int_{x_0}^{x_3} f(x)dx \approx \frac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\right] \tag{9}$$

4.5.1 Code R

5 Notions fondamentales en algèbre linéaire

5.1 Définition d'une matrice

Matrice Soit la matrice $A = [a_{ij}]_{m \times n}$:

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{m1} & a_{m2} & ... & a_{mn} \end{bmatrix}$$

matrice symétrique La matrice suivante est un exemple de matrice symétrique, i.e. $a_{ij} = a_{ji}$:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 4 & 1 \end{bmatrix}$$

Triangulaire inférieure Une matrice inférieure (carrée) est constituée de 0 en dessous de la diagonale :

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

Triangulaire supérieure Une matrice supérieure (carrée) est constituée de 0 au-dessus de la diagonale :

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 8 & 3 & 0 \\ 2 & 3 & 9 \end{bmatrix}$$

Matrice diagonale Est à la fois une matrice triangulaire inférieure et supérieure (il y a des zéros partout, sauf en $a_{ii}, \quad \forall i$).

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Matrice identité Matrice souvent notée I_n, c'est une matrice carrée diagonale, où $a_{ii} = 1, \quad \forall i$.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

La matrice identitaire I est beaucoup utilisée lorsqu'on doit inverser une matrice (résolution d'équation).

Matrice transposée il est important de ne pas confondre la matrice inverse et la matrice transposée ! La matrice transposée A^T est la matrice A ré-écrite de façon à ce que $A^T = [a_{ij}]^T = [a_{ji}]$. On définit quelques propriétés de la transposée ici :

1. $(A^T)^T = A$
2. $(A + B)^T = A^T + B^T$
3. $(kA)^T = kA^T$
4. $(AB)^T = B^T A^T$
5. $A^T A$ et AA^T sont symétriques.

5.2 Arithmétique matricielle

5.2.1 Propriétés arithmétiques

Voici une série d'opérations matricielle possibles :

$$A + B = [a_{ij} + b_{ij}] \tag{10}$$
$$A - B = [a_{ij} - b_{ij}] \tag{11}$$
$$cA = [ca_{ij}] \tag{12}$$
$$AB = \left[\sum_{k=1}^{p} a_{ik}b_{kj}\right] \text{ , avec } A = [a_{ip}] \text{ et } B = [b_{pj}] \tag{13}$$
$$A + B = B + A \tag{14}$$
$$A + (B + C) = (A + B) + C \tag{15}$$
$$AB \neq BA \tag{16}$$
$$A(BC) = (AB)C \tag{17}$$
$$A(B + C) = AB + AC \tag{18}$$
$$(B + C)A = BA + CA \tag{19}$$
$$A(B - C) = AB - AC \tag{20}$$
$$(B - C)A = BA - CA \tag{21}$$
$$a(B + C) = aB + aC \quad \text{où } a \text{ est une constante} \tag{22}$$
$$a(B - C) = (aB)C = B(aC) \quad \text{où } a \text{ est une constante} \tag{23}$$
$$a(BC) = aB - aC \quad \text{où } a \text{ est une constante} \tag{24}$$
$$(a + b)C = aC + bC \quad \text{où } a \text{ et } b \text{ sont des constantes} \tag{25}$$
$$(a - b)C = aC - bC \quad \text{où } a \text{ et } b \text{ sont des constantes} \tag{26}$$
$$a(bC) = (ab)C \quad \text{où } a \text{ et } b \text{ sont des constantes} \tag{27}$$
$$\tag{28}$$

Exemple

[Exemple de produit matriciel] Soit les matrice A et B tel que

$$A = \begin{bmatrix} 3 & 4 & 6 \\ 0 & 2 & 10 \end{bmatrix} B = \begin{bmatrix} 3 & 4 \\ 0 & 2 \\ 1 & 5 \end{bmatrix}$$

Alors, le produit matriciel $A \times B$ donnera

$$\begin{bmatrix} 3 & 4 & 6 \\ 0 & 2 & 10 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 0 & 2 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 3 \times 3 + 4 \times 0 + 6 \times 1 & 3 \times 4 + 4 \times 2 + 6 \times 5 \\ 0 \times 3 + 2 \times 0 + 10 \times 1 & 0 \times 4 + 2 \times 2 + 10 \times 5 \end{bmatrix}$$
$$= \begin{bmatrix} 15 & 50 \\ 10 & 54 \end{bmatrix}$$

5.2.2 Trace d'une matrice

On peut facilement calculer la trace d'une matrice carrée, en sommant les coefficients de la diagonale de cette dernière, soit :

$$Tr(A) = \sum_{i=1}^{n} a_{ii} \tag{29}$$

5.3 Déterminant

LE déterminant sert, entre autres, à pouvoir déterminer la matrice inverse A^{-1} d'une matrice A. Voici comment on le calcule :

5.4 Matrice Inverse

Pour isoler les éléments d'une matrice (i.e. la matrice en soit) on ne peut pas passer par une simple division comme on le fait algébriquement : il faut multiplier par la matrice inverse. Pour se faire, il existe des méthodes numériques mais aussi des méthodes qui se font très bien papier (et qui étaient utilisée au Cégep)

A Annexe Fonctions Excel de simulation

Loi	Fonctions Excel (nom anglais)	Fonctions Excel (nom français)
Bêta	BETA.DIST	LOI.BETA.N
	BETA.INV	BETA.INVERSE.N
Binomiale	BINOM.DIST	LOI.BINOMALE.N
	BINOM.INV	LOI.BINOMIALE.INVERSE
Binomiale nég.	NEGBINOM.DIST	LOI.BINOMIALE.NEG.N
Exponentielle	EXPON.DIST	LOI.EXPONENTIELLE.N
F (Fisher)	F.DIST	LOI.F.N
	F.INV	INVERSE.LOI.F.N
Gamma	GAMMA.DIST	LOI.GAMMA.N
	GAMMA.INV	LOI.GAMMA.INVERSE.N
Hypergéom.	HYPGEOM.DIST	LOI.HYPERGEOMETRIQUE.N
Khi carré	CHISQ.DIST	LOI.KHIDEUX
	CHISQ.INV	LOI.KHIDEUX.INVERSE
Log-normale	LOGNORM.DIST	LOI.LOGNORMALE.N
	LOGNORM.INV	LOI.LOGNORMALE.INVERSE.N
Normale	NORM.DIST	LOI.NORMALE.N
	NORM.INV	LOI.NORMALE.INVERSE.N
	NORM.S.DIST	LOI.NORMALE.STANDARD.N
	NORM.S.INV	LOI.NORMALE.STANDARD.INVERSE.N
Poisson	POISSON.DIST	LOI.POISSON.N
t (Student)	T.DIST	LOI.STUDENT.N
	T.INV	LOI.STUDENT.INVERSE.N
Weibull	WEIBULL.DIST	LOI.WEIBULL.N

TAB. 2.1 – Liste des fonctions relatives à des lois de probabilité depuis Excel 2010.

B Fonctions R de simulation

Loi de probabilité	Racine dans R	Noms des paramètres
Bêta	`beta`	`shape1, shape2`
Binomiale	`binom`	`size, prob`
Binomiale négative	`nbinom`	`size, prob ou mu`
Cauchy	`cauchy`	`location, scale`
Exponentielle	`exp`	`rate`
F (Fisher)	`f`	`df1, df2`
Gamma	`gamma`	`shape, rate ou scale`
Géométrique	`geom`	`prob`
Hypergéométrique	`hyper`	`m, n, k`
Khi carré	`chisq`	`df`
Logistique	`logis`	`location, scale`
Log-normale	`lnorm`	`meanlog, sdlog`
Normale	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
t (Student)	`t`	`df`
Uniforme	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

TAB. 2.2 – Lois de probabilité pour lesquelles il existe des fonctions dans le système R de base

B.1 Simulation distribution discrète

On peut simuler une distribution discrète non paramétrique quelquoncque avec la fonction de R `sample()`

```
sample(x = 1:5, # le support de x
       size = 10, # on simule 10 nombres
       replace = T, # tirage avec remplacement
       prob = c(0.2, 0.3, 0.1, 0.25, 0.15)) # masses de probabilité
```

```
##  [1] 3 5 4 2 3 1 2 5 1 2
```

C Règles de dérivation

$$
\begin{aligned}
c' &= 0 \\
(u^n)' &= nu^{n-1} \\
(e^u)' &= e^u u' \\
(a^u)' &= a^u \ln(a) u' \\
(\ln(u))' &\frac{1}{u} u' \\
(\log_a(u))' &= \frac{1}{\ln(a) u} u' \\
(\sin(u))' &= \cos(u) u' \\
(\cos(u))' &= -\sin(u) u' \\
(\tan(u))' &= \sec^2(u) u' \\
(\cot(u))' &= -\csc^2(u) u' \\
(\sec(u))' &= \sec(u)\tan(u) u' \\
(\csc(u))' &= -\csc(u)\cot(u) u' \\
(\arcsin(u))' &= \frac{1}{\sqrt{1-u^2}} u' \\
(\arccos(u))' &= \frac{-1}{\sqrt{1-u^2}} u' \\
(\arctan(u))' &= \frac{1}{1+u^2} u'
\end{aligned}
$$