Validation des logiciels de calcul numérique probabilistes

Michaël Baudin

EDF R&D
6, quai Watier,78401 Chatou
michael.baudin@edf.fr

Octobre 2013

Sommaire

Valider, c'est quoi ?

On valide quoi ? Une *fonction* de calcul :

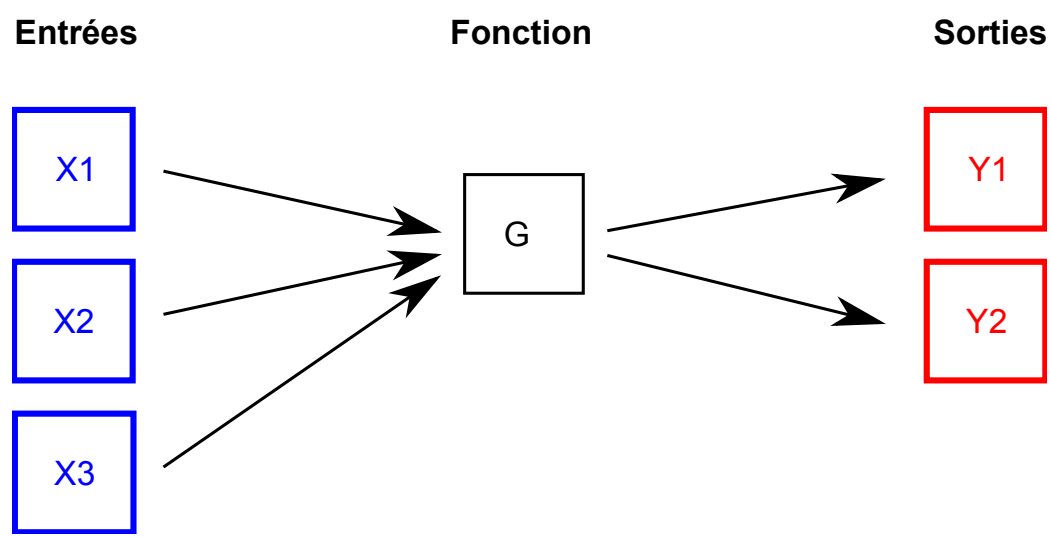
(sortie calculée) (est une fonction de) (entrée fournie)

On valide comment ? Par comparaison :

(sortie calculée) (correspond à) (sortie attendue)

Fonction de calcul - exemples :

- $G(x) = \sin(x)$ où $x \in \mathbb{R}$
- $G(x_1, x_2) = x_1 + x_2$ où $x_1, x_2 \in \mathbb{R}$
- $G(\mathbf{x}) = \text{Aster}(\mathbf{x})$ où $\mathbf{x} = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ et n est grand.



Certes, et après ?

Objectif :

1. Valider les résultats de sortie d'un logiciel ...
2. ... de calcul numérique probabiliste.

Partie "logiciel" pure en annexe :

- Testabilité
  - Améliorer la testabilité d'une fonction,
  - la notion de privé/public pour gérer les tests
- Combinatoire
  - Limiter le nombre de combinaisons d'options à tester,
  - choisir ses expériences selon un plan.

Généralités *relativement* intuitives : annexe (sur question).

Tester un logiciel de calcul

Partie calcul numérique :

1. Comparer deux réels en théorie :
   - comment est représenté un réel en machine : l'erreur de représentation des nombres flottants,
   - comment une (petite) erreur en entrée X se transforme en (parfois grande) erreur sur la sortie Y : erreur et conditionnement.
2. Comparer deux réels en pratique :
   - comment utiliser une librairie d'assertions numérique pour réaliser un test,
   - comment tester une fonction probabiliste.

Pas du tout intuitif : c'est le corps de la présentation.

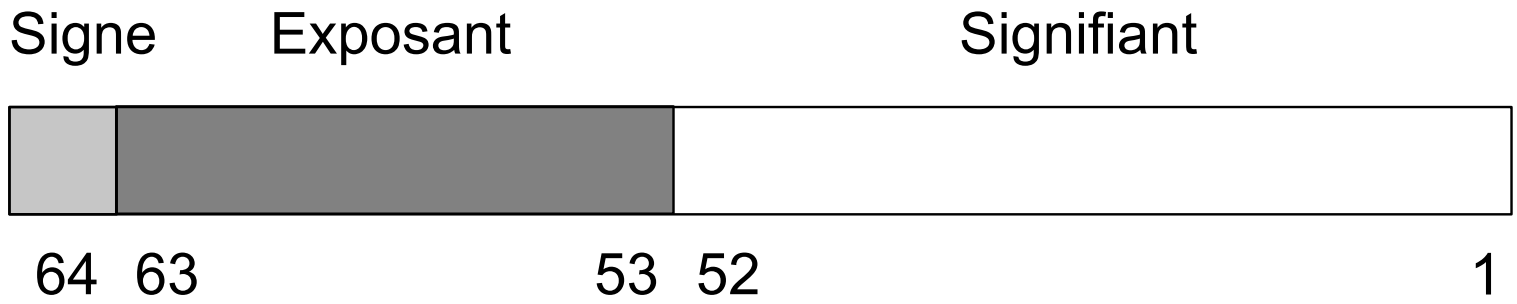
Flottants

Flottants

La différence entre $\mathbb{R}$ et $\mathcal{F}$.

Nombres flottants 64 bits

Un nombre à virgule flottante binaire de type IEEE-754 64 bits :

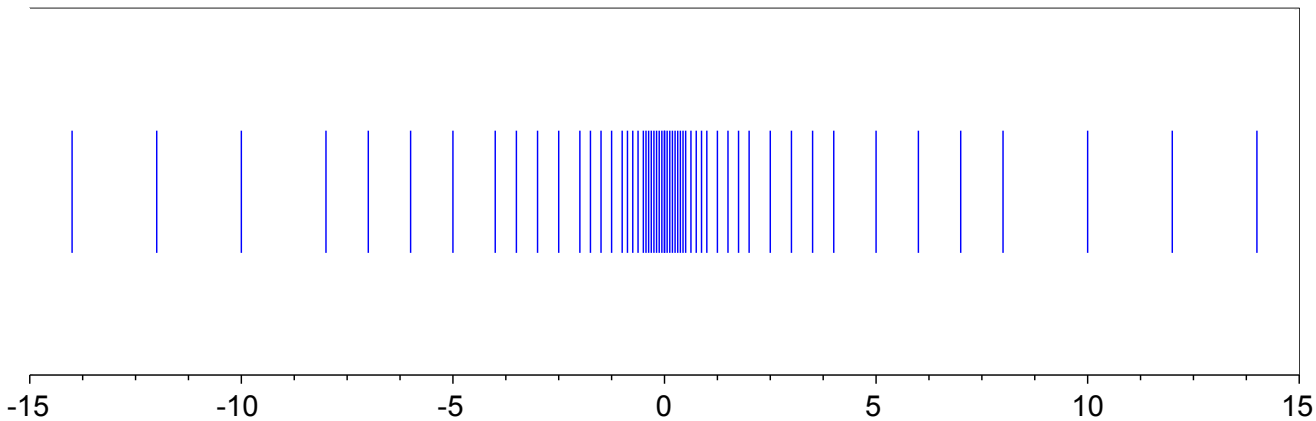


Les "doubles" :

- 1 bit de signe
- 11 bits d'exposant
- 53 bits de signifiant (dont 1 implicite)

Nombres flottants : principes

Tous les flottants dans un système "jouet".



Analogie - Mesure de longueurs sur une règle graduée :

- qui permet de mesurer des nombres négatifs
- qui a une longueur finie
- dont les graduations sont plus resserées autour de zéro

Nombres flottants : principes

Limitations de principe :

- $\mathbb{R}$ est continu, mais les flottants sont discrets
- $\mathbb{R}$ est infini, mais les flottants sont en nombre finis

Limitations techniques :

- Exposant limité : limitation de l'ordre de grandeur (l'amplitude)
- Signifiant limité : limitation de la précision

Système flottant

Définition 1

(*Système à virgule flottante*) Un système à virgule flottante est défini par les quatres entiers β, p, e_{min} et e_{max} où

- $\beta \in \mathbb{N}$ est la base et satisfait $\beta \geq 2$,
- $p \in \mathbb{N}$ est la précision et satisfait $p \geq 2$,
- $e_{min}, e_{max} \in \mathbb{N}$ sont les exposants extrêmes et sont tels que

$$e_{min} < 0 < e_{max}.$$

Exemple 2

Exemple : Les "doubles" IEEE :

$$\beta = 2, \quad p = 53, \quad e_{min} = -1022, \quad e_{max} = 1023.$$

Nombres flottants

Définition 3

(*Nombre à virgule flottante*) Un nombre à virgule flottante x est un réel $x \in \mathbb{R}$ tel qu'il existe (m, e) tels que :

$$x = m \cdot \beta^{e-p+1}, \tag{1}$$

où $m \in \mathbb{Z}$ est la partie intégrale et satisfait

$$|m| < \beta^p, \tag{2}$$

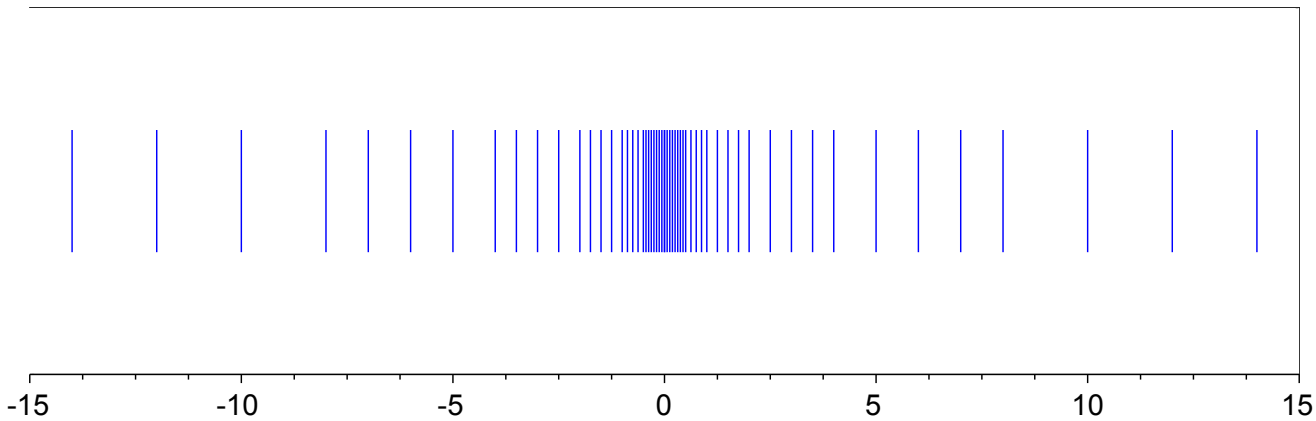
$e \in \mathbb{Z}$ est l'exposant et satisfait

$$e_{min} \leq e \leq e_{max}. \tag{3}$$

Note : la partie intégrale m peut être négative.

Nombres flottants

Tous les flottants (normalisés et dénormalisés) dans le système "jouet" : $(\beta, p, e_{min}, e_{max}) = (2, 3, -2, 3)$.

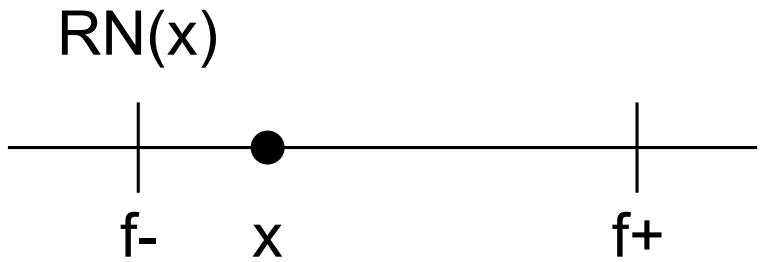


La liste (en gras, les dénormalisés) : -14, -12, -10, -8, -7, -6, -5, -4, -3.5, -3, -2.5, -2, -1.75, -1.5, -1.25, -1, -0.875, -0.75, -0.625, -0.5, -0.4375, -0.375, -0.3125, -0.25, **-0.1875**, **-0.125**, **-0.0625**, 0., **0.0625**, **0.125**, **0.1875**, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7, 8, 10, 12, 14.

Modes d'arrondi

Mode d'arrondi par défaut :

- Si x est dans l'intervalle $[-\Omega, \Omega]$, on utilise RN(x) : arrondi au plus proche (round-to-nearest)



Q : connaissez-vous des réels qui ne sont pas des flottants ?

Erreur d'arrondi

On note $fl(x)$ la représentation flottante de x.

Théorème 4

(Précision machine) Soit $x \in \mathbb{R}$. Supposons que le système flottant est arrondir-au-plus-proche. Si x est dans l'intervalle normalisé, alors

$$fl(x) = x(1+\delta), \qquad |\delta| \leq u = \frac{1}{2}\beta^{1-p}.$$

où u est la précision machine. Si x est dans l'intervalle dénormalisé, alors :

$$|fl(x) - x| \leq \beta^{e_{min}-p+1}.$$

Exercice : preuve.
En anglais : u est le "unit roundoff".
Q : voyez-vous la différence entre les deux en termes d'erreur ?

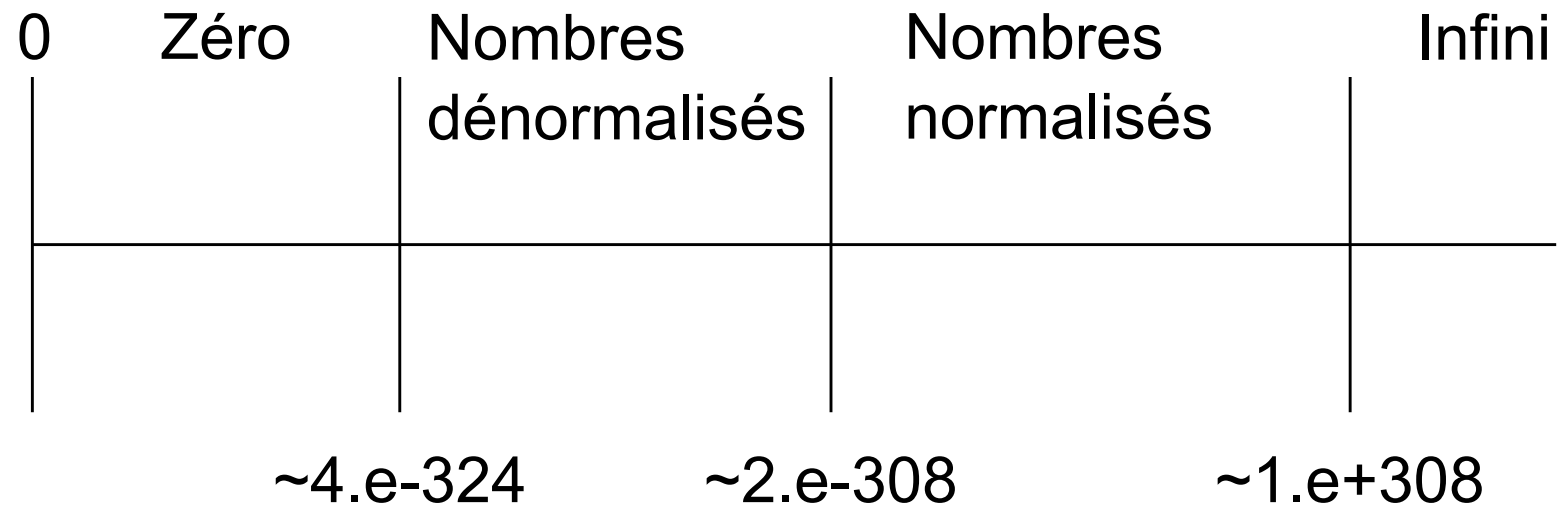
Les doubles IEEE

Les doubles IEEE 754.

Base β	2
Precision p	53
Exposant	11
Exposant minimum e_{min}	-1022
Exposant maximum e_{max}	1023
Plus grand normal Ω	$\left(2 - 2^{-52}\right) \cdot 2^{1023} \approx 1.8 \times 10^{308}$
Plus petit normal > 0 μ	$2^{-1022} \approx 2.22 \times 10^{-308}$
Plus petit dénormalisé > 0 α	$2^{-1022-53+1} \approx 4.94 \times 10^{-324}$
Epsilon Machine ϵ_M	$2^{-52} \approx 2.220 \times 10^{-16}$
Précision machine u	$2^{-53} \approx 1.110 \times 10^{-16}$

Les doubles IEEE

Les doubles positifs, avec une échelle "artistique".



Erreur

Erreur

Comment calculer une erreur absolue, relative, le nombre de chiffres significatifs, l'erreur forward/backward et le conditionnement.

Erreur relative, absolue

Définition 5

(*Erreur absolue, relative*) Soit :

- un nombre réel : x,
- son approximation : $\hat{x}$.

Alors, l'erreur absolue est :

$$E_{abs}(x, \hat{x}) = |x - \hat{x}|$$

et l'erreur relative est :

$$E_{rel}(x, \hat{x}) = \frac{|x - \hat{x}|}{|x|}$$

si $x \neq 0$.

Erreur relative, absolue

Exemple 6

	x	$\hat{x}$	E_{abs}	E_{rel}
A	1.	2.	1.	1.
B	1.	1.000001	9.99999999918e-07	9.99999999918e-07
C	1.e100	1.000001e100	9.99999999909e+93	9.99999999909e-07
D	0.	1.e-100	1e-100	inf

Question - Analyse :

	Relative	Absolue
Grande	?	?
Petite	?	?

Erreur relative, absolue

Exemple 7

	x	$\hat{x}$	E_{abs}	E_{rel}
A	1.	2.	1.	1.
B	1.	1.000001	9.99999999918e-07	9.99999999918e-07
C	1.e100	1.000001e100	9.99999999909e+93	9.99999999909e-07
D	0.	1.e-100	1e-100	inf

Analyse :

	Relative	Absolue
Grande	A, D	A, C
Petite	B, C	B, D

Erreur relative, absolue

En général :

- Si $x \neq 0$, utiliser l'erreur relative.
- Si $x = 0$, utiliser l'erreur absolue.

Si $\mathbf{x} \in \mathbb{R}^n$ est un vecteur, on peut utiliser l'erreur relative composante-par-composante :

$$\max_{i=1,\ldots,n} \left| \frac{x_i - \hat{x}_i}{x_i} \right|$$

Nombre de chiffres corrects

Définition 8

(*Nombre de chiffres corrects en base β - la log-erreur relative*) La Log-Erreur relative en base β est définie par :

$$LRE_\beta(x, \hat{x}) = -\log_\beta(E_{rel}(x, \hat{x}))$$

Avec des doubles :

	$\beta = 2$	$\beta = 10$
LRE min	0	0
LRE max	53	15.95

Précision : exemples de tests

Calcul : distributions.

Données de référence : Mathematica 5.2, ELV (Knüsel, 2003).

Microsoft's performance on correcting errors in Excel's statistical distributions

Distribution	Excel 97	Excel 2000	Excel 2002	Excel 2003	Excel 2007
Binomial	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Hypergeometric	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Poisson	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Normal	Flaws reported	Not fixed	Not fixed	Fixed	
Inv. normal	Flaws reported	Not fixed	Poor fix	Poor fix	Not fixed
Inv. chi-square	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Inv. t	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Inv. F	Flaws reported	Not fixed	Not fixed	Poor fix	Not fixed
Gamma				Flaws reported	Not fixed
Inv. beta				Flaws reported	Not fixed

Inverse standard normal distribution with parameter (p)

p	EXACT	ELV Ed.2	EXCEL 97/2K	EXCEL XP	EXCEL 2003/2007	CALC 2.3.0	GNUMERIC 1.7.11
5E−1	0	Exact	Exact	5.47142E−10	−1.39214E−16	Exact	Exact
1E−1	−1.28155	Exact	Exact	Exact	Exact	Exact	Exact
1E−2	−2.32635	Exact	−2.32634	Exact	Exact	Exact	Exact
1E−3	−3.09023	Exact	−3.09024	−3.09025	Exact	Exact	Exact
1E−4	−3.71902	Exact	−3.71947	−3.71909	Exact	Exact	Exact
1E−5	−4.26489	Exact	−4.26546	−4.26504	Exact	Exact	Exact
1E−6	−4.75342	Exact	−4.76837	−4.75367	Exact	Exact	Exact
1E−7	−5.19934	Exact	−5000000	−5.19969	Exact	Exact	Exact
1E−15	−7.94135	Exact	−5000000	−7.93597	Exact	Exact	Exact
1E−16	−8.22208	Exact	−5000000	−8.29366	Exact	Exact	Exact
1E−100	−21.2735	Exact	−5000000	−8.29366	Exact	Exact	Exact
1E−197	−29.9763	No solution	−5000000	−8.29366	Exact	Exact	Exact
1E−198	−30.0529	No solution	−5000000	−8.29366	−30	Exact	Exact
1E−300	−37.0471	No solution	−5000000	−8.29366	−30	Exact	Exact

Ref. : Yalta (2008)

Précision : exemples de tests

Calcul : régression linéaire.
Données de référence : (American) National Institute of Standards and Technology (NIST), Statistical Reference Data sets (StRD).

LREs for StRD regression data sets

Software package	Excel 2000/XP	Excel 2003^	JMP 5.0 Fit *Y* by *X*	JMP 5.0 Fit Model	Minitab 14.0^	Minitab 14.0 (worksheet)[a]	R 1.9.1	SAS 9.1^	SAS ORTHOREG 9.1	Splus 6.2^	SPSS 12.0^	Stata 8.1^	StatCrunch 3.0
Min LREs for beta coefficients													
Lower difficulty													
Norris	12.1	12.0	12.2	13.3	12.2	12.2	12.5	11.9	12.5	11.9	12.3	12.7	7.6
Pontius	11.2	^12.0	11.2	11.8	11.5	11.5	12.7	11.5	12.2	12.4	12.5	^12.2	7.4
Average difficulty													
NoInt1	15.0	15.0	14.7	14.7	15.0	15.0	14.3	^15.0	15.0	14.3	^15.0	14.7	NS[e]
NoInt2	15.0	15.0	15.0	15.0	15.0	15.0	14.9	15.0	15.0	15.0	15.0	15.0	NS[e]
High difficulty													
Filip	0.0	^^7.2	NS[f]	NS[f]	6.9	6.9	1.3	NS[g]	NS[g]	^7.4	NS[g]	NS[g]	1.3
Longley	7.4	^^13.4	NS[h]	8.3	12.7	12.7	13.0	8.6	13.6	12.9	12.1	11.6	7.2
Wampler1	6.6	^^9.9	8.0	7.0	9.6	9.6	9.8	6.6	9.7	9.6	NS[g]	^7.2	15.0
Wampler2	9.6	^^13.4	10.6	9.5	12.7	12.7	13.5	9.6	13.5	12.9	NS[g]	9.7	15.0
Wampler3	6.6	^^10.1	8.0	7.0	9.3	9.3	9.2	6.6	9.6	^9.3	NS[g]	^6.8	15.0
Wampler4	6.6	^8.1	8.0	7.0	8.7	8.7	7.5	6.6	8.2	^7.8	NS[g]	6.5	15.0
Wampler5	6.6	6.1	8.0	7.0	6.8	6.8	5.5	6.6	6.2	^5.8	NS[g]	6.5	6.1

Ref. : extrait de Keelinga, Pavurb (2007) (le tableau original contenait d'autres comparaisons).

Erreur forward, backward

- On considère une fonction G, un réel d'entrée x et on veut calculer

$$y = G(x).$$

- Soit $\hat{y}$ une approximation de y :

$$\hat{y} \approx G(x)$$

- Comment mesurer la "qualité" de $\hat{y}$?

Erreur forward, backward

Mesure possible : l'erreur est faible si

$$E_{rel}(y, \hat{y}) = \frac{|y - \hat{y}|}{|y|} \approx u$$

Autre mesure : pour quelle donnée d'entrée avons nous résolu le problème ?
"Quelle perturbation de l'entrée x est nécessaire pour obtenir exactement $\hat{y}$?
Pour quelle valeur de Δx avons-nous :

$$\hat{y} = G(x + \Delta x)$$

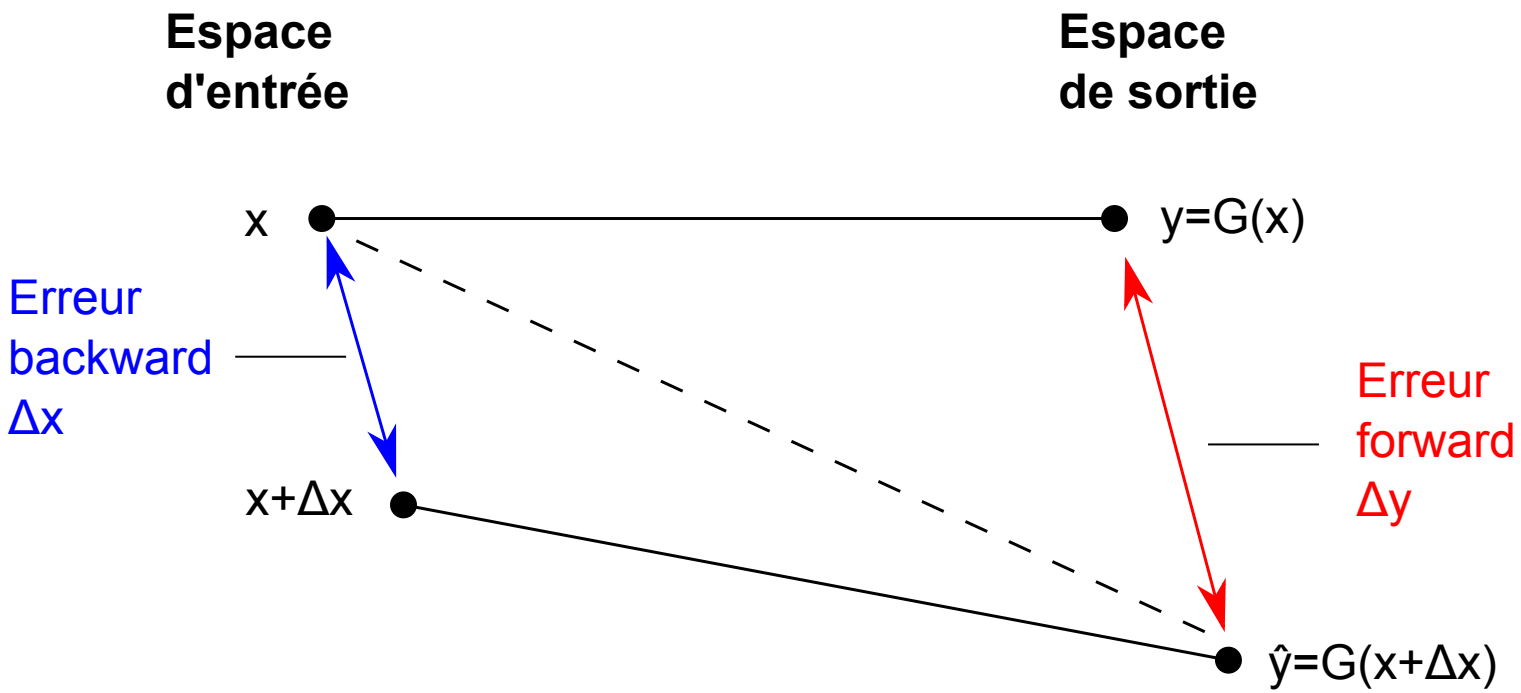
Si il y a plusieurs Δx, on prend le plus petit.

Erreur forward, backward

Deux types d'erreur :

- Erreur forward : $E_{rel}(y, \hat{y})$ ou $E_{abs}(y, \hat{y})$
- Erreur backward : $|\Delta x|/|x|$ (relative) ou $|\Delta x|$ (absolue)

Erreurs forward et backward pour $y = G(x)$.
Solide : exact. Pointillé : calculé.



Conditionnement

Définition 9

(*Conditionnement dans* $\mathbb{R}$) Soit G une fonction continûment dérivable telle que G'' est bornée. On suppose que $G(x)$ n'est pas nul. Le conditionnement de G est :

$$K_G(x) = \lim_{\Delta x \to 0} \left| \frac{E_{rel}(y, \hat{y})}{E_{rel}(x, \hat{x})} \right|$$

avec

$$\hat{x} = x + \Delta x, \qquad y = G(x) \text{ et } \hat{y} = G(x + \Delta x).$$

Le coefficient multiplicatif K_G *mesure le rapport entre l'erreur relative sur* y *et l'erreur relative sur* x*.*

Conditionnement

Quand Δx est petit, on a

$$E_{rel}(y, \hat{y}) \approx K_G(x) \times E_{rel}(x, \hat{x})$$

donc :

$$LRE_{10}(y, \hat{y}) \approx LRE_{10}(x, \hat{x}) - \log_{10}(K_G(x)).$$

Conclusion : $\log_{10}(K_G(x))$ est le nombre de chiffres décimaux perdus dans y par rapport à x à cause du conditionnement dans G.

Théorème 10

(Conditionnement dans $\mathbb{R}$) Sous les mêmes conditions, le conditionnement de G est :

$$K_G(x) = \left| \frac{xG'(x)}{G(x)} \right|.$$

Conditionnement

Règle générale :

$$\text{erreur forward} \lesssim \text{conditionnement} \times \text{erreur backward}$$

Donc :

$$\text{erreur backward petite} \Rightarrow \text{erreur forward petite}$$

mais pas le contraire.

Conditionnement : exemple de log1p

Exemple 11

$$G(x) = \log(x)$$

Le conditionnement est :

$$K_{\log}(x) = \left| \frac{1}{\log(x)} \right|,$$

qui est grand pour $x \approx 1$ car $\log(1) = 0$.
Or $\log(1 + \Delta x) \approx 0$, lorsque Δx est petit.
Donc un petit changement autour de $x = 1$ provoque sur $\log(x)$:

- un petit changement absolu
- un grand changement relatif

Le rôle de la fonction $\mathrm{log1p}(x) = \log(1 + x)$ est de résoudre ce problème de précision.

Conditionnement : exemple de log1p

Application de log1p pour les probabilités.
Calcul de la fonction de répartition inverse (quantile) de la loi exponentielle de moyenne μ.
La fonction de répartition est :

$$p = F(x) = 1 - e^{-\frac{x}{\mu}}$$

La fonction de répartition inverse est :

$$x = F^{-1}(p) = -\mu \log(1 - p)$$

Conditionnement : exemple de log1p

Calcul de $F^{-1}(p)$: pour $\mu = 1$ et $p = 10^{-20}$, le quantile exact est $x = 10^{-20}$.

Naïf :

```
def expinv(p,mu):
    if p==1.0:
        x=float("inf")
    else:
        x=-mu*log(1.0-p)
    return x

>>> expinv(1.e-20,1.0)
-0.0
```

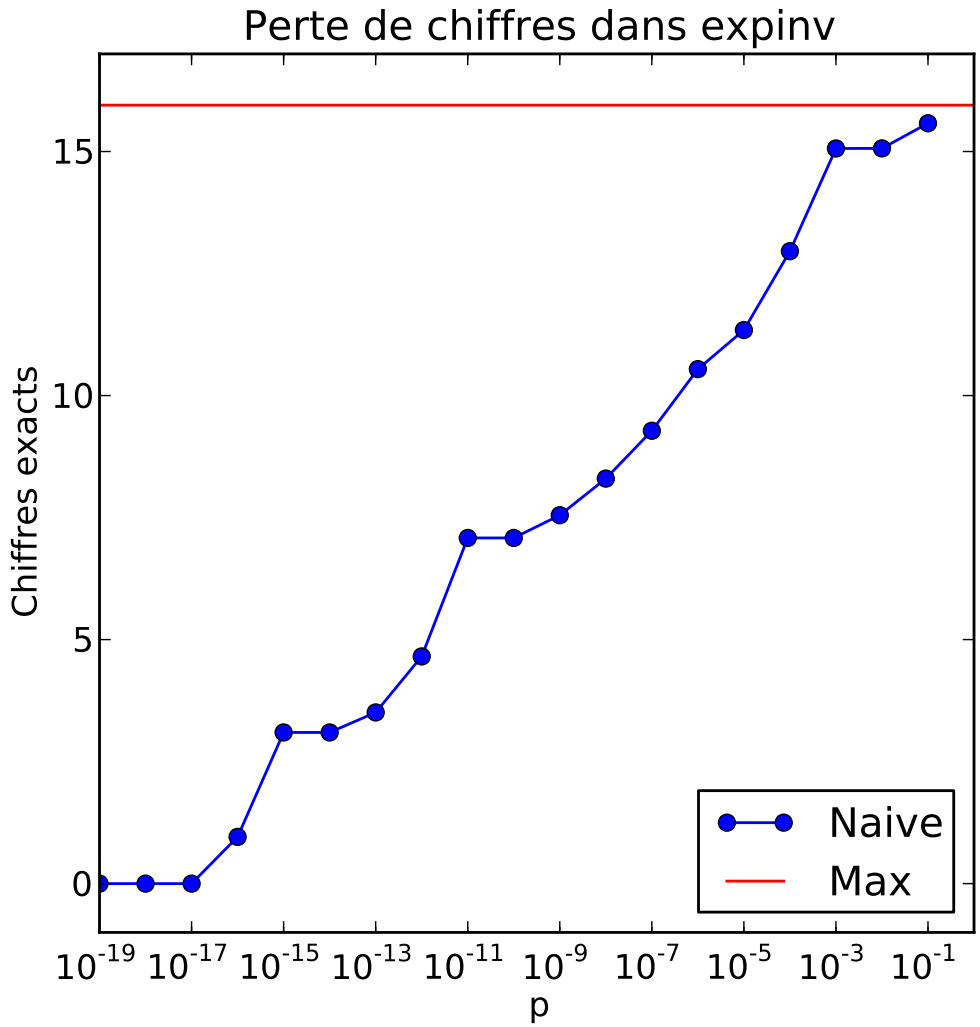
Robuste :

```
def expinvMieux(p,mu):
    if p==1.0:
        x=float("inf")
    else:
        x=-mu*log1p(-p)
    return x

>>> expinvMieux(1.e-20,1.0)
1e-20
```

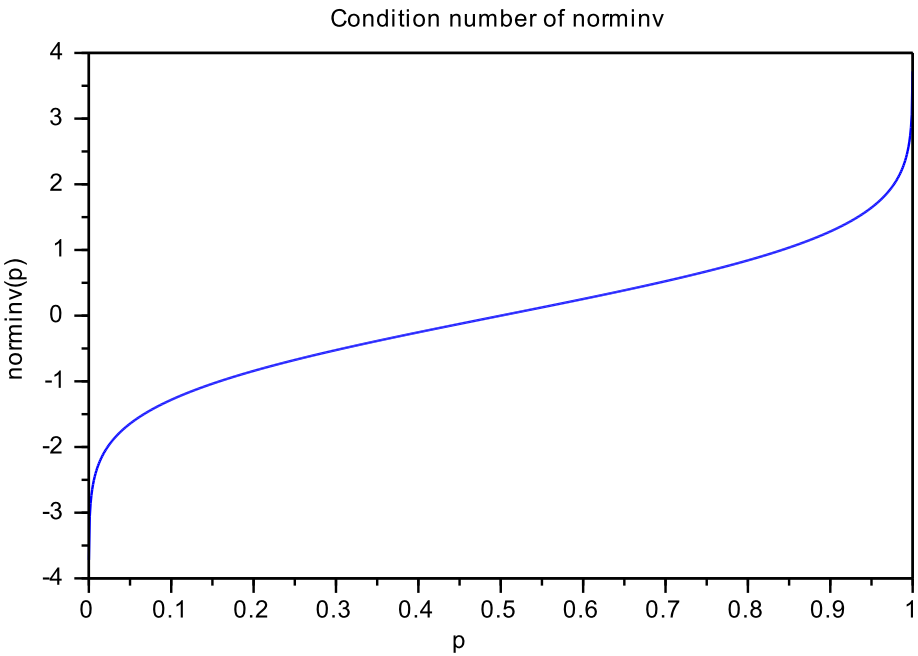
Conditionnement : exemple de log1p

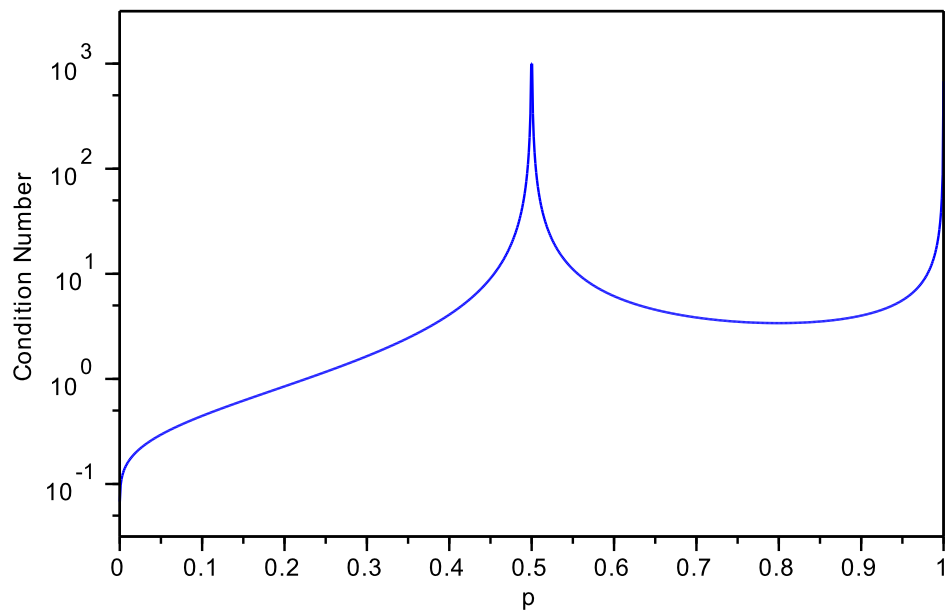
"Mais je ne traite pas des probabilité aussi faibles !"
Certes, il reste que pour des probabilités décroissantes (10^{-1}, 10^{-2}, 10^{-3}, etc...), il y a une perte progressive des chiffres corrects.
Calcul de $F(x)$: utiliser expm1 (sinon cancellation).



Conditionnement : quantiles loi normale

La fonction norminv (quantile de la loi normale) est mal conditionnée en $p = 0.5$ et $p = 1$.





Ref. Edelman (2010)

Tests

Tests

Utiliser une librairie d'assertions.
Comment tester un calcul probabiliste ?

Assertions : pourquoi ?

Pourquoi utiliser une librairie d'assertions ?

- Pour tester le comportement d'une fonction.
- Pour structurer les tests unitaires.
- Faciliter le test de calculs numériques, en particulier, issus d'algorithmes de calcul (par exemple itératifs)
- Préciser la notion de valeurs numériques "presque égales", c'est à dire telles que l'erreur relative est "petite".

Démarrage

Comme exemple, nous considérons les fonctions `assert` de Scilab.
Mais tous les languages ont implémenté le même concept :

- Matlab : Eddins (2009)
- C++ : Cppunit
- Java : Junit
- Python : PyUnit
- Fortran : Flibs/ftnunit (Sourceforge)

L'idée principale : `assert_checktrue`

Exemple dans Scilab : les fonctions `assert`.
La fonction `assert_checktrue` vérifie qu'une matrice de booléens est vraie. On suppose que la fonction `G` prend un réel et renvoit un booléen.

```
x=2
computed=G(x)
assert_checktrue ( computed )
```

Deux cas :

1. Si (au moins) une des entrées est fausse, une erreur est générée : le code s'arrête.
2. Sinon, la fonction ne fait rien.

L'idée principale : `assert_checkequal`

La fonction `assert_checkequal` vérifie que deux valeurs sont égales.

```
x=2
computed=G(x)
expected=3
assert_checkequal ( computed , expected )
```

Deux cas :

1. Si les deux valeurs ne sont pas égales, une erreur est générée : le code s'arrête.
2. Sinon, la fonction ne fait rien.

L'idée principale : `assert_checkalmostequal`

La fonction `assert_checkalmostequal` vérifie qu'une valeur calculée est proche d'une valeur attendue.
Dans l'exemple suivant, on vérifie que 1.23456 est proche de 1.23457 avec une erreur relative de 10^{-4} :

```
assert_checkalmostequal ( 1.23456 , 1.23457 , 1.e-4 )
```

test_run

```
-->test_run("development_tools|assert")
TMPDIR = C:\Users\C61372\AppData\Local\Temp\SCI_TMP_4924_
01/01-[development_tools|assert] :
01/11-[development_tools|assert] checkalmostequal..passed
02/11-[development_tools|assert] checkequal........passed
03/11-[development_tools|assert] checkerror........passed
04/11-[development_tools|assert] checkfalse........passed
05/11-[development_tools|assert] checkfilesequal...passed
06/11-[development_tools|assert] checktrue.........passed
07/11-[development_tools|assert] comparecomplex....passed
08/11-[development_tools|assert] computedigits.....passed
09/11-[development_tools|assert] cond2reltol.......passed
10/11-[development_tools|assert] cond2reqdigits....passed
11/11-[development_tools|assert] generror..........passed
---------------------------------------------------------------
Summary
tests                 11  -  100 %
passed                11  -  100 %
failed                 0  -    0 %
skipped                0
length                 43.95  sec
---------------------------------------------------------------
```

Focus sur `assert_checkalmostequal`

Séquences d'appel possibles :

```
assert_checkalmostequal(computed,expected)
assert_checkalmostequal(computed,expected,reltol)
assert_checkalmostequal(computed,expected,reltol,abstol)
assert_checkalmostequal(computed,expected,reltol,abstol,..
    comptype)
```

où

- `reltol` : l'erreur relative (défaut : $\sqrt{\epsilon_M}$)
- `abstol` : l'erreur relative (défaut : 0)
- `comptype` : type de norme pour la comparaison. Pour une norme matricielle : "matrix", pour une comparaison élément-par-élément : "element" (défaut : "element")

Ajuster la tolérance

Comment tenir compte du conditionnement pour ajuster la tolérance lors du test d'une fonction élémentaire ?
Le conditionnement de G est :

$$K_G(x) = \lim_{\Delta x \to 0} \left| \frac{E_{rel}(y, \hat{y})}{E_{rel}(x, \hat{x})} \right| = \left| \frac{xG'(x)}{G(x)} \right|.$$

avec

$$\hat{x} = x + \Delta x, \qquad y = G(x) \text{ et } \hat{y} = G(x + \Delta x).$$

Souhaite donc :

$$|G(x + \Delta x) - G(x)| \approx K_G(x) \frac{|\Delta x|}{|x|} |G(x)|$$

où Δx est la distance entre x et son flottant le plus proche.
Ref. : Brorson, Edelman, Moskowitz, 2012

Ajuster la tolérance

Or, on peut démontrer que, pour deux flottants x et y voisins, on a :

$$|x - y| \leq \epsilon_M |x|$$

où $y = x^+$ ou bien $y = x^-$ et

$$\epsilon = 2^{1-p} = 2^{-52} \approx 2.220 \times 10^{-16},$$

si on utilise des doubles.
Cela implique

$$\frac{|\Delta x|}{|x|} \leq \epsilon_M .$$

Ajuster la tolérance

On demande donc :

$$\frac{|G_{calcul} - G_{exact}|}{|G_{exact}|} \leq C K_G(x) \epsilon_M$$

où C est une petite constante.
Si

$$G_{exact} \neq 0,$$

on peut donc utiliser l'erreur relative :

$$reltol = C K_G(x) \epsilon_M.$$

Sauf si il s'agit d'une fausse singularité de $K_G(x)$, si

$$G_{exact} = 0,$$

alors l'erreur absolue :

$$abstol = C K_G(x) \epsilon_M |G_{exact}| = 0$$

ne peut pas être utilisée.

Ajuster la tolérance

Ajuster C :

- Augmenter C augmente la tolérance : C doit être le plus petit possible.
- En pratique, commencer par $C = 0$ car certaines fonctions sont telles que, si x est un flottant, alors G_{calcul} est le flottant le plus proche de la valeur exacte $G(x)$. Par exemple : $+, -, *, /$ et $\sqrt{x}$.
- Puis, prendre $C = 1, 2, 3, ...$ dans cet ordre jusqu'à faire passer tous les tests.
- Etre obligé de prendre $C \geq 10$ signale une implémentation sous-optimale.

Tester un calcul probabiliste

Difficulté : nous travaillons sur des calculs probabilistes, et non pas sur des calculs déterministes.
Hypothèse :

$$Y = G(X)$$

avec X une variable aléatoire.
Conséquence : Y est une variable aléatoire que l'on souhaite tester.

Tester un calcul probabiliste

D'abord, on devrait utiliser des algorithmes de qualité :

- un générateur de nombres pseudo-aléatoires uniformes de qualité,
- des algorithmes de générations de nombres non uniformes de qualité,
- des algorithmes de calcul de fonctions de distribution de qualité,
- des techniques de résolution de systèmes linéaires de qualité (par exemple, pivot de Gauss avec permutation des lignes utilisant l'interface LAPACK),
- etc...

Quel sens a une validation, si on sait par avance que les algorithmes utilisés sont mauvais ?

Tester un calcul probabiliste

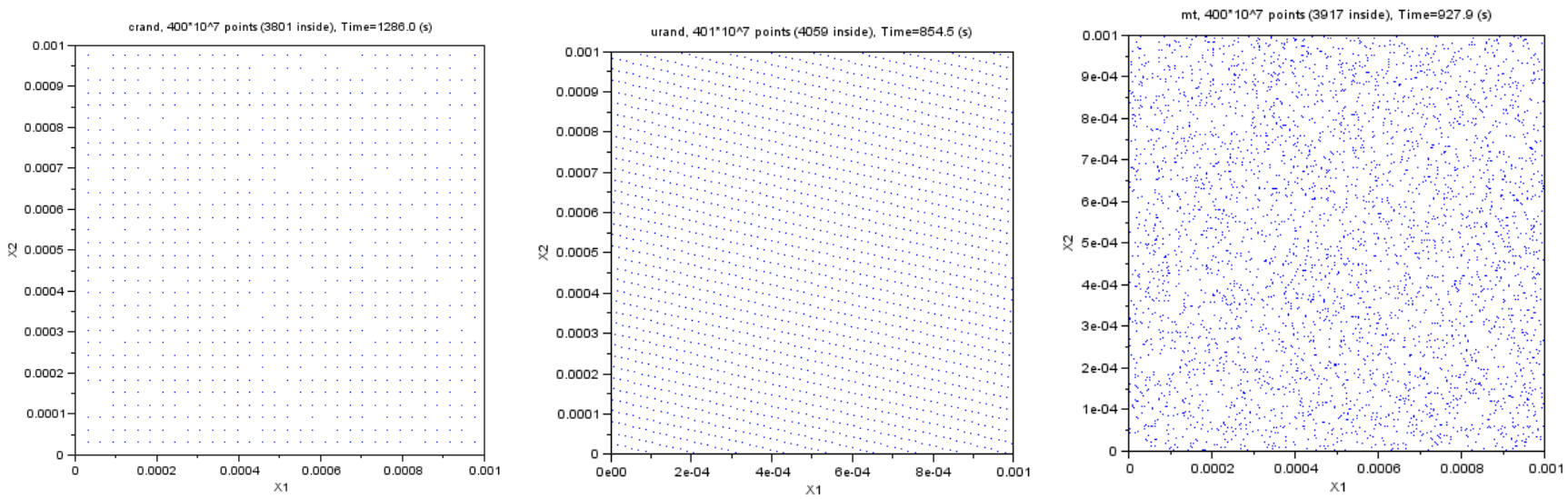
Générateurs Pseudo-Aléatoires Uniformes

- Dans la fonction `rand()` du language C.
- Dans la fonction `grand` de Scilab : URAND
  - Type de générateur : linéaire congruentiel.
  - Auteurs : M. Malcolm, C. Moler (1973).
  - Période : $2^{31} \approx 2.1 \times 10^9$
- Dans la fonction `grand` de Scilab : Mersenne-Twister (par défaut)
  - Type de générateur : Linear Feedback Shift Register.
  - Code : MT19937 par M. Matsumoto, T. Nishimura (1998).
  - Période : $2^{19937} \approx 10^{6001}$.

Tester un calcul probabiliste

Exemple : Je génère 400×10^7 nombres uniformes pseudo-aléatoires dans le carré unité en dimension 2, puis je ne garde que les nombres dans l'intervalle $[0, t]^2$, avec $t = 0.001$ (nécessite ≈ 15 min).

Tester un calcul probabiliste



Moralité : dans Scilab, le statisticien n'utilise pas `rand`, mais il utilise `grand`.

Tester un calcul probabiliste

Exemple : estimer une moyenne empirique avec

$$M_n = \frac{1}{n} \sum_{i=1}^{n} y_i$$

où

$$y_i = G(x_i), \quad i = 1, 2, \ldots, n,$$

et X suit une distribution donnée.
Comment tester le résultat ?

Tester un calcul probabiliste

Test :

```
// 1000 valeurs U(0,1) :
x=grand(1000,1,"unf",0,1)
y=G(x)
computed=mean(y)
expected=0.5
reltol = TODO // <- ?
assert_checkalmostequal(computed,expected,reltol)
```

Questions :

- Comment déterminer `expected` ?
- Comment déterminer `reltol` ?
- Dois-je prendre une erreur absolue ou relative ?
- Et si je relance une simulation ?
- Quelle erreur dûe au conditionnement ?
- Quelle erreur dûe à l'estimateur ?

Comment déterminer `expected` ?

1. Par un calcul exact. Exemple : on connaît la loi de $Y = G(X)$ dans certains cas. Erreur relative : ϵ_M.
2. Par un calcul avec une autre méthode. Exemple : calcul d'intégrale multidimensionnelle par quadrature tensorisée. Erreur relative : dépend de la méthode.
3. Par un calcul Monte-Carlo avec *beaucoup* de simulations. Problème : combien de simulations et quelle est l'erreur qui en résulte ? (voir plus loin)

Dans tous les cas, la référence `expected` est associée à une erreur qui doit être connue au moment du test.

Et si je relance une simulation ?

Si je relance une simulation, le test ne va plus passer ?
En pratique, les nombres sont (souvent) pseudo-aléatoires, et se calculent selon un algorithme déterministe :

$$u_{n+1} = f(u_n), \quad n = 1, 2, \ldots.$$

La suite est entièrement déterminée par la graîne u_0 (en anglais "seed") du générateur :

```
grand("setsd",0)
```

En général, la graîne par défaut est une constante du simulateur : pour pouvoir débugger facilement et reproduire exactement les mêmes nombres à chaque simulation.
Sinon, on peut la fixer arbitrairement.

Tester un calcul probabiliste

Test :

```
grand("setsd",0) // <- Nouveau
x=grand(1000,1,"unf",0,1)
y=G(x)
computed=mean(y)
expected=0.5
reltol = TODO // <- ?
assert_checkalmostequal(computed,expected,reltol)
```

Question subsidiaire : comment rester relativement indépendant du générateur lui-même ?
Réponse : dans la suite.

Moyenne - partie 1 : analyse du conditionnement

Considérons la somme :

$$S(y_1, y_2, \ldots, y_n) = y_1 + y_2 + \ldots + y_n.$$

Son conditionnement est :

$$C(y_1, y_2, \ldots, y_n) = \frac{|y_1| + |y_2| + \ldots + |y_n|}{|y_1 + y_2 + \ldots + y_n|}.$$

Ref. : Stewart (1996)
Conséquences :

- Si la somme est exactement zéro : conditionnement infini.
- Lorsque les valeurs y_i ont le même signe : pas de problème.
- Lorsque les y_i sont de signes différents et d'ordre de grandeurs très différents : conditionnement élevé.

Moyenne - partie 1 : analyse du conditionnement

Exemple : 7680 valeurs issues d'un calcul de modélisation du climat.
Ordre de grandeur de $|y_i|$: 10^{15}.

```
-->exact=0.35798583924770355 2;
-->sum(x)
 ans  =
  - 2.9960938
-->accsum_dblcompsum(x)
 ans  =
    0.3579858
-->c=sum(abs(x))/abs(sum(x))
 c  =
    1.770D+16
```

Conclusion :

- Lorsque qu'on utilise un algorithme naïf, on obtient aucun chiffre significatif.
- Lorsque qu'on utilise un algorithme particulier (ici, l'algorithme de somme doublement compensée de Priest et Kahan), on obtient le résultat exact.

Ref. : Y. He, C.H.Q. Ding (2001), Higham (2002)

Moyenne - partie 2 : analyse probabiliste

Supposons que

$$Y \sim \mathcal{N}(\mu, \sigma^2).$$

et estimons l'espérance par la moyenne empirique.
Si $\mu = 0$:

- problème mal conditionné.
- y_i de signes différents, certaines d'entre elles étant de grande amplitude (mais elles sont rares).

Si $\mu \approx 1$:

- problème probablement bien conditionné.
- la plupart des y_i sont positifs.

Moyenne - partie 2 : analyse probabiliste

Conséquences :

- Si $\mu = 0$: erreur absolue (somme mal conditionnée).
- Si $\mu \approx 1$: erreur relative et le conditionnement est souvent raisonnable.

Analyse :

- Il peut être souhaitable d'utiliser un algorithme très précis.
- D'un autre côté, en général, l'intervalle de confiance sur M_n est souvent beaucoup plus large que l'erreur dûe au conditionnement (voir la suite).

Moyenne - partie 2 : analyse probabiliste

On note :

$$S_n^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - M_n)^2$$

l'écart-type empirique biaisé.
Si $n \gtrsim 100$, alors

$$P\left(M_n - z_{1-\alpha/2}\frac{S_n}{\sqrt{n-1}} \leq \mu \leq M_n + z_{1-\alpha/2}\frac{S_n}{\sqrt{n-1}}\right) \simeq 1-\alpha,$$

où $z_{1-\alpha/2}$ est le quantile de niveau $1-\alpha/2$ de la loi Normale standard.
Exemple : $\alpha = 0.05$

$$P\left(M_n - 1.96\frac{S_n}{\sqrt{n-1}} \leq \mu \leq M_n + 1.96\frac{S_n}{\sqrt{n-1}}\right) \simeq 0.95.$$

Interprétation pour la validation : utiliser une erreur absolue.

Tester un calcul probabiliste

Test :

```
grand("setsd",0)
n=1000
x=grand(n,1,"unf",0,1)
y=G(x)
computed=mean(y)
expected=0.5
abstol=1.96*st_deviation(y)/sqrt(n-1)  // <- Nouveau
assert_checkalmostequal(computed,expected,[],abstol)
```

Avantages :

- La tolérance absolue dépend de la valeur qu'on estime.
- Le test passe pour 95% des graînes.
- Le test est peu sensible au générateur.

Inconvénient :

- Il faut l'écart-type exact (ou bien une estimation).
- Il peut être nécessaire d'ajuster la graîne pour faire passer le test.

Aller plus loin

Pour aller plus loin, on pourrait

- tester la distribution des réalisations y_i
- répéter l'expérience et comparer la distribution des réalisations M_n avec la loi normale issue du TCL

Si on voulait tester le générateur de nombres pseudo-aléatoires :

- comparaison de la distribution des U_n avec la distribution uniforme théorique
- technique : tests statistiques (par exemple : test du χ^2 ou Kolmogorov-Smirnov).

(Autre exemple pour les probabilités :

- probabilité et probabilité complémentaire.)

Conclusion

- Pour tester un logiciel de calcul, connaître la différence entre un réel mathématique et un nombre à virgule flottante est utile.
- Une petite erreur relative en entrée X peut être amplifiée par un mauvais conditionnement, d'où une grande erreur relative en sortie Y.
- Utiliser une librairie d'assertions prenant en compte les erreurs relatives ou absolues permet de vérifier des calculs numériques.
- On peut valider des calculs probabilistes, en vérifiant les propriétés des estimateurs statistiques qu'on calcule.

Bibliographie

- "Accuracy and stability of numerical algorithms", N. Higham, 2002, Society for Industrial and Applied Mathematics
- "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications", Yun He and Chris H.Q. Ding. Journal of Supercomputing, Vol.18, Issue 3, 259-277, March 2001.
- "Handbook of Floating Point Computations", J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, 2010, Birkhäuser Basel.
- "Afternotes on numerical analysis", G.W. Stewart, 1996, Lecture 7 "Computing sums"

Bibliographie

- "Open Source and Traditional Technical Computing", Alan Edelman, Massachusetts Institute of Technology, Scilabtec10, June 16, 2010
- "Testing Math Functions in Microsoft Cloud Numerics", Stuart Brorson, Alan Edelman, Ben Moskowitz, MSDN Magazine, October 2012
- Eddins, "Automated Software Testing for MATLAB", Computing in Science & Engineering, 2009

Bibliographie

- A comparative study of the reliability of nine statistical software packages, Kellie B. Keeling, Robert J. Pavur, Computational Statistics & Data Analysis 51 (2007), 3811 – 3831
- Assessing the Reliability of Statistical Software : Part I, B. D. McCullough, The American Statistician, Vol. 52, No. 4 (Nov., 1998), pp. 358-366
- Assessing the Reliability of Statistical Software : Part II B. D. McCullough, The American Statistician, Vol. 53, No. 2 (May, 1999), pp. 149-159
- Fixing statistical errors in spreadsheet software : the case of Gnumeric and Excel, B.D. Mc Cullough, 2004
- The accuracy of statistical distributions in Microsoft Excel 2007, A. Talha Yalta, Computational Statistics and Data Analysis, 52 (2008) 4579-4586

Merci de votre attention !
Questions ?

ANNEXES

Flottants : détails

Flottants : détails

Bit implicite, flottants extrêmes et conditionnement d'une fonction réelle.

Nombres flottants

Unicité de la représentation ? Je divise M par 2, j'ajoute 1 à e.
Donc la définition précédente ne garantit pas l'unicité de la représentation.

Théorème 12

(Flottant normalisé) Un flottant est normalisé si M satisfait :

$$\beta^{p-1} \leq |M| < \beta^p.$$

Si x est un flottant normalisé non nul, alors sa représentation (M, e) est unique et

$$e = \lfloor \log_\beta(|x|) \rfloor, \qquad M = \frac{x}{\beta^{e-p+1}}.$$

Pour obtenir l'unicité, on ajoute une borne sur M.

Nombres flottants dénormalisés

Définition 13

(*Flottant dénormalisé*) Un flottant (M, e) est dénormalisé si $e = e_{min}$ et M satisfait :

$$|M| < \beta^{p-1}$$

En anglais : *subnormal*.

Nombres flottants

En base $\beta = 2$:

- Si x normalisé : $x = \pm(1.d_2 \cdots d_p)_2 \cdot 2^e$,
- Si x dénormalisé : $x = \pm(0.d_2 \cdots d_p)_2 \cdot 2^e$.

Dans le standard IEEE 754-2008 :

- Un encodage particulier de l'exposant permet de savoir si x est normalisé ou dénormalisé.
- Le bit d_1 est donc stocké de manière *implicite*.
- Cela explique pourquoi les doubles sont associés à la précision $p = 53$, alors que seulement 52 bits sont stockés.

Nombres flottants

Exemple 14

Considérons le système flottant jouet de base $\beta = 2$, de précision $p = 3$ et d'amplitude d'exposants $e_{min} = -2$ et $e_{max} = 3$.
Le nombre réel $x = 3$ peut être représenté par le nombre flottant $(M, e) = (6, 1)$:

$$x = 6 \cdot 2^{1-3+1} = 6 \cdot 2^{-1} = 3. \tag{4}$$

Vérifions les équations. La partie intégrale M satisfait

$$|M| = 6 \leq \beta^p - 1 = 2^3 - 1 = 7$$

et l'exposant e satisfait

$$-2 \leq e \leq 3$$

Nombres flottants

Exemple 15

Même système flottant "jouet" : $(\beta, p, e_{min}, e_{max}) = (2, 3, -2, 3)$.
Les nombres flottants normalisés sont tels que :

$$\beta^{p-1} = 4 \leq |M| < \beta^p = 8.$$

Exercice : montrer que $(M, e) = (6, 1)$ est le flottant normalisé pour $x = 3$, et que $(M, e) = (3, 2)$ ne l'est pas.

Nombres flottants

Exemple 16

Même système flottant "jouet" : $(\beta, p, e_{min}, e_{max}) = (2, 3, -2, 3)$.
On considère $x = 0.125$. On trouve :

$$\lfloor \log_\beta(|x|) \rfloor = -3$$

qui est plus petit que $e_{min} = -2$. Le nombre est dénormalisé. On met $e = -2$ et on calcule :

$$M = \frac{x}{\beta^{e-p+1}} = 2$$

On trouve que M est un entier. Donc le couple $(M, e) = (2, -2)$ est une représentation dénormalisée de $x = 0.125$ dans ce système.

Nombres réels en base β

Théorème 17

(Représentation d'un réel en base β) Supposons que x est un flottant. Alors, le nombre x peut être écrit sous la forme :

$$x = \pm \left(d_1 + \frac{d_2}{\beta} + \ldots + \frac{d_p}{\beta^{p-1}} \right) \cdot \beta^e,$$

ou encore :

$$x = \pm (d_1.d_2 \cdots d_p)_\beta \cdot \beta^e.$$

avec $d_1, ..., d_p \in \{0, 1, ..., b-1\}$.

Exercice : faire la preuve.

Idée : décomposer $|M|$ sous la forme :

$$|M| = d_1 \beta^{p-1} + d_2 \beta^{p-2} + \ldots + d_p.$$

Nombres flottants

Théorème 18

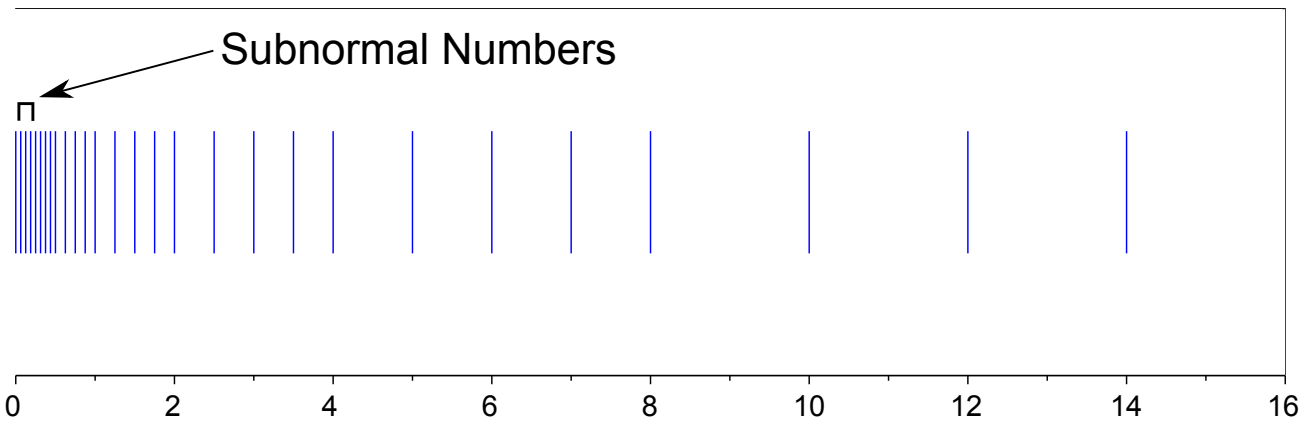
(Bit de tête de la représentation en base β) Supposons que x est un flottant.

- *Si x est normalisé, alors $d_1 \neq 0$.*
- *Si x est dénormalisé, alors $d_1 = 0$.*

Exercice : preuve.

Nombres flottants

Les flottants positifs (normalisés et dénormalisés) dans le système "jouet" : $(\beta, p, e_{min}, e_{max}) = (2, 3, -2, 3)$.



Nombres flottants extrêmes

Théorème 19

(Flottants extrêmes) Considérons le système flottant β, p, e_{min}, e_{max}.

- *Le plus petit flottant normalisé est*

$$\mu = \beta^{e_{min}}.$$

- *Le plus grand flottant normalisé est*

$$\Omega = (\beta - \beta^{1-p})\beta^{e_{max}}.$$

- *Le plus petit flottant dénormalisé est*

$$\alpha = \beta^{e_{min}-p+1}.$$

Conditionnement

Preuve :
Par définition, on a :

$$K_G(x) = \lim_{\Delta x \to 0} \left| \frac{\frac{\hat{y}-y}{y}}{\frac{\hat{x}-x}{x}} \right|$$

La formule de Taylor :

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + O((\Delta x)^2)$$

implique

$$\frac{\hat{y} - y}{y} = \frac{f(x + \Delta x) - f(x)}{f(x)} = \frac{f'(x)\Delta x + O((\Delta x)^2)}{f(x)}.$$

Conditionnement

Donc :

$$\left|\frac{E_{rel}(y,\hat{y})}{E_{rel}(x,\hat{x})}\right| = \left|\frac{\frac{f'(x)\Delta x+O((\Delta x)^2)}{f(x)}}{\frac{\Delta x}{x}}\right| \quad (5)$$

$$= \left|\frac{x}{\Delta x}\frac{f'(x)\Delta x+O((\Delta x)^2)}{f(x)}\right| \quad (6)$$

$$= \left|\frac{x}{f(x)}(f'(x)+O(\Delta x)\right| \quad (7)$$

Pour conclure, on prend $\Delta x \to 0$.

Conditionnement

Conditionnement de $\log(x)$ quand $x \to 1$:

x	$\log(x)$	$K(x)$
1.01	0.00995033085317	100.499170807
1.0001	9.99950003333e-05	10000.4999917
1.000001	9.99999499918e-07	1000000.50008

Erreur d'évaluation

Théorème 20

(Erreur relative des opérations algébriques.) Soit x et y des doubles. Les opérations arithmétiques +,-,,/ satisfont :*

$$fl(x\ op\ y) = (x\ op\ y)(1+\delta),$$

où

$$|\delta| \leq u.$$

L'erreur relative sur x op y est inférieure à u pour les opérations algébriques.

Erreur relative, absolue

Théorème 21

(Erreur relative (2ième définition)) Supposons que x et $\hat{x}$ sont deux réels différents de zéro. Alors il existe un réel ρ tel que :

$$E_{rel}(x, \hat{x}) = |\rho|$$

avec

$$\hat{x} = x(1 + \rho).$$

Démonstration.

On a $\hat{x} = x + \rho x$, donc :

$$\frac{\hat{x} - x}{x} = \rho.$$

La réciproque est triviale. □

Erreur relative, absolue

Théorème 22

(Invariance par rapport à un changement d'échelle.) On suppose que x, $\hat{x}$ sont différents de zéro. Soit $\alpha > 0$ un réel représentant un facteur de changement d'échelle. On considère

$$x' = \alpha x, \quad \hat{x}' = \alpha \hat{x}.$$

Alors

$$E_{rel}(x, \hat{x}) = E_{rel}(x', \hat{x}')$$

L'erreur relative est inchangée par changement d'échelle.

Testabilité

Testabilité

Améliorer la testabilité d'une fonction, la notion de privé/public pour gérer les tests.

"Tester, c'est impossible !"

- En général, une fonction qui n'est pas testable ne devrait pas être fournie à un utilisateur.
- Au contraire, la fonction doit être conçue pour être facile à tester.

Deux cas :

- Si l'entrée a une influence mesurable sur la sortie, alors la fonction est testable.
- Sinon la fonction n'est pas testable.

Exemple de fonction non testable : l'appel à la fonction est sous la forme :

$$G()$$

et met à jour un état interne caché à l'utilisateur.
Exemple : bouton "mise à jour" d'une interface graphique, sans aucun changement de l'état externe.

"Tester, c'est impossible !"

Si la fonction n'est pas testable, que faire ?

- Mieux choisir les entrées. Exemple : ajouter des entrées cachées du code pour le rendre testable (*certains* - mais pas tous - paramètres algorithmiques *en dur* : pas de discrétisation h, nombre d'itérations, maximal, etc...).
- Mieux choisir la fonction. Exemple : découper une fonction globale en plusieurs sous-fonctions pour pouvoir tester des parties de l'algorithme.
- Mieux choisir les sorties. Exemple : ajouter des sorties cachées au code pour le rendre testable.

Exemple :

- Pouvoir définir le pas de discrétisation h permet de tester la convergence d'une méthode de différences finies lorsque $h \to 0$.

Jusqu'où tester ?

Une *fonctionnalité* est un couple (X, G), où X est dans un certain espace des paramètres possibles (par exemple : $X_1 \in [0, 1]$, $X_1 = 0, 1, ..., 4$, etc...).
Objectif :

$$\text{nombre de tests } \geq C \times \text{ nombres de fonctionnalités,}$$

où $C \geq 1$ est la plus petite constante possible.
"Ma librairie a 10^6 *fonctionnalités différentes. Je dois faire* 10^6 *tests ?"*
En général, beaucoup moins : on ne teste que

- ce que l'utilisateur modifie en entrée X,
- les fonctions G que l'utilisateur peut manipuler,
- ce que l'utilisateur "voit" en sortie Y.

Jusqu'où tester ?

Principes :

- Economie : pourquoi tester une fonction invisible de l'utilisateur (privée) ?
- Confiance : pourquoi fournir à l'utilisateur une fonction (publique) non testée ?

Exemples :

- les états d'une interface graphique (p.ex. :allumé/éteint),
- les valeurs dans un fichier de sortie.

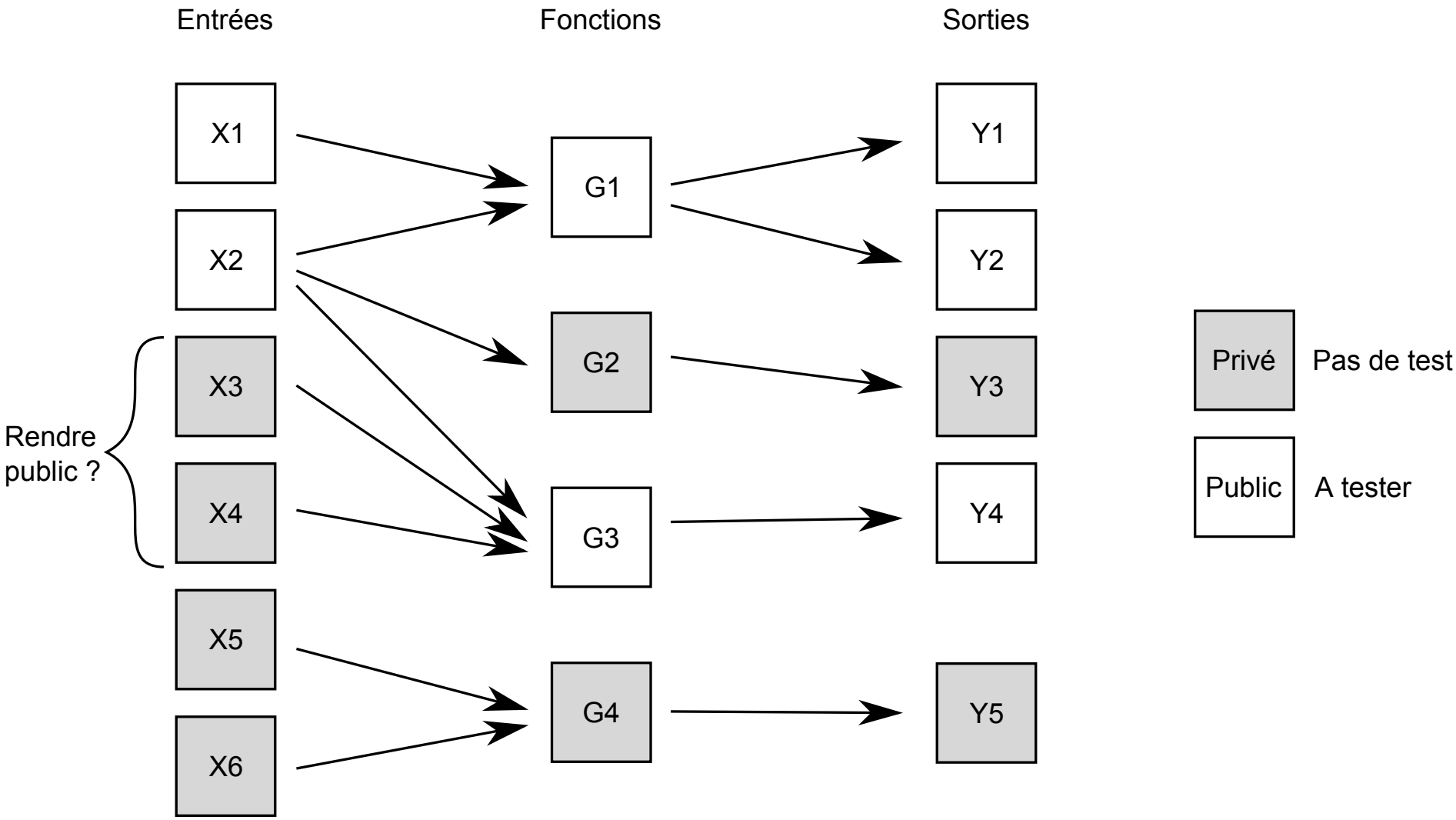
Jusqu'où tester ?

Conséquences :

- Limiter ce que "voit" l'utilisateur en entrée ou en sortie permet de limiter les tests : notion de privé/publique dans les langages de programmation.
- A l'extrême, l'unique fonction facile à tester n'a aucune entrée et aucune sortie $G()$. Problème : la fonction n'est plus utilisable !
- En pratique, compromis entre testabilité et utilisabilité :

  Compte tenu de l'effort à fournir pour tester une fonctionnalité, est-ce possible de la rendre publique ?

Jusqu'où tester ?



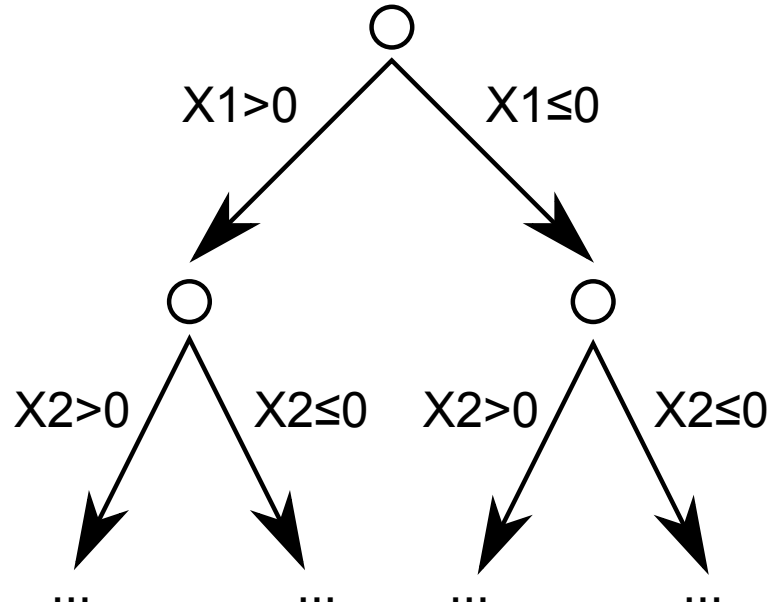
Combinatoire

Combinatoire

Limiter le nombre de combinaisons d'options à tester, choisir ses expériences selon un plan.

"Impossible de tester avec toutes ces combinaisons !"

"Ma fonction a 10 *options binaires différentes. Je dois faire* 2^{10} *tests ?"*



A minima, 10 tests suffisent car :

- on peut *factoriser* le code source pour regrouper les branches de l'arbre combinatoire,
- la plupart du temps, le développeur est suffisamment "économe" (ou flemmard ?) pour écrire un code factorisé.

Limiter la combinatoire

Exemple 23

On ne dit pas :

```
if (x1>0)&(x2>0)
      // Cas 1
elseif (x1>0)&(x2<=0)
      // Cas 2
elseif (x1<=0)&(x2>0)
      // Cas 3
elseif (x1<=0)&(x2<=0)
      // Cas 4
end
```

On dit (quand c'est possible) :

```
if (x1>0)
      // Cas 1
else
      // Cas 2
end
if (x2>0)
      // Cas 3
else
      // Cas 4
end
```

Car :

- le premier bloc `if` est exécuté quelque soit `x2`,
- le second bloc `if` est exécuté quelque soit `x1`.

Limiter la combinatoire

Exemple 24

On suppose qu'on a 2 options avec 4 niveaux.
Quelles expériences (x_1, x_2) doit-on réaliser ?

Full factorial :
$E = 16$

(1,1)	(3,1)
(1,2)	(3,2)
(1,3)	(3,3)
(1,4)	(3,4)
(2,1)	(4,1)
(2,2)	(4,2)
(2,3)	(4,3)
(2,4)	(4,4)

One at a time :
$E = 8$

(1,1)
(2,1)
(3,1)
(4,1)
(1,1)
(1,2)
(1,3)
(1,4)

Diagonal :
$E = 4$

(1,1)
(2,2)
(3,3)
(4,4)

Lorsque les options de G ne sont plus binaires

Supposons que les n variables d'entrée discrèetes ont m niveaux.
Combien d'expériences E ?

- A minima, $E = m$ tests pour un plan "Diagonal".
  - Minimum possible d'expériences qui couvrent toutes les options.
- En général, $E = nm$ tests pour un plan "One At A Time".
  - Chaque option est testée séparément, ce qui facilite l'écriture du test.
- Au mieux, $E = n^m$ tests pour un plan "Full Factorial".
  - Toutes les combinaisons sont couvertes.

Moyenne - partie 2 : analyse probabiliste

(No offense...)
On note :

$$S_n^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - M_n)^2$$

l'écart-type empirique biaisé.

Théorème 25

Supposons que $\{y_i\}_{i=1,\dots,n}$ *sont des réalisations indépendantes de loi* $\mathcal{N}(\mu, \sigma^2)$. *Alors :*

$$\frac{M_n - \mu}{S_n/\sqrt{n-1}} \sim \mathcal{T}_{n-1},$$

où $\mathcal{T}_{n-1}$ *est la loi de Student à* $n-1$ *degrés de liberté.*

Dans le cas général, la loi de Y est inconnue, et il n'est pas possible de calculer un intervalle de confiance sur M_n.

Moyenne - partie 2 : analyse probabiliste

(No offense...)

Théorème 26

(Théorème central limite) Supposons que $\{y_i\}_{i=1,\dots,n}$ *sont des réalisations indépendantes de même loi, de moyenne* μ *et de variance* σ^2*. Alors :*

$$\frac{1}{\sqrt{n}} \sum_{i=1}^{n} \frac{y_i - \mu}{\sigma} \to \mathcal{N}(0,1).$$

De plus, lorsque n est grand, la loi de Student converge vers la loi normale.

Conditionnement

Le conditionnement

- est une propriété du problème mathématique
- n'est pas une propriété de la méthode numérique pour résoudre le problème

Le conditionnement mesure

- l'amplification d'une erreur relative sur x
- et son impact relatif sur y

Lorsque les calculs s'enchaînent, l'erreur peut

- augmenter
- diminuer

en fonction de :

- du conditionnement du problème mathématique
- de la méthode numérique utilisée

Erreur forward, backward

Définition 27

(*Backward stable*) Une méthode pour calculer $y = G(x)$ est backward stable si, pour tout x, elle produit une valeur $\hat{y}$ avec une petite erreur backward, c'est à dire telle que

$$\hat{y} = G(x + \Delta x)$$

pour un Δx "petit".

La manière de définir "petit" dépend du contexte.