Extension trigo : documentation partielles

Projet GL groupe 13

Janvier 2020

1 Introduction

Dans ce compte rendu nous faisons état de l'évolution du contenu de notre extension Trigo. A ce titre, nous présentons différentes façons de coder les approximations des fonctions trigonométriques classiques, en nous appuyant sur un nombre aussi diverse que possible de techniques nous permettant d'approcher de la manière la plus efficace chacune des fonctions.

2 Fonction sinus

2.1 Polynômes de Chebyshev

Dans un espace de Sobolev (un espace de Sobolev est un espace vectoriel de fonctions muni de la norme obtenue par la combinaison de la norme Lp de la fonction elle-même et de ses dérivées jusqu'à un certain ordre), l'ensemble des polynomes de Chebyshev forme une base orthonormale, permettant l'expression d'une fonction continue sur [-1,1] via l'expression :

$f(x) = \sum_{n=0}^{\infty} a_n T_n(x)$

On peut souligner qu'en cas de discontinuité, la série va converger vers la moyenne de la limite droite et gauche.

2.2 Proposition d'approximation de la fonction sinus

Nous proposons de projeter la fonction sinus dans la base de Chebyshev à la différence que nous la divisons par un polynôme de degré impair valant 0 en 0 et pi et 1 en pi /2 dans le but de réduire l'erreur dynamique.
Nous utilisons le polynôme : $\frac{8x}{3\pi} + \frac{8x^3}{3\pi^3}$

En outre,la spécifité des polynômes de Chebyshev réside dans l'égalité :

$$\int_{-1}^{1} T_n(x)T_m(x)\,\frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & n \neq m \\ \pi & n = m = 0 \\ \frac{\pi}{2} & n = m \neq 0 \end{cases}$$

qui peut être démontrée en posant x = cosθ et en utilisant $T_n(cos\theta)$ = cos n.
En exprimant notre fonction dans la base de Chebyshev on obtient donc :

$$\int_{-1}^{+1} \frac{T_m(x)f(x)}{\sqrt{1-x^2}}\,dx = \sum_{n=0}^{\infty} a_n \int_{-1}^{+1} \frac{T_m(x)T_n(x)}{\sqrt{1-x^2}}\,dx,$$

Or cette somme étant nulle pour tout n différent de m, nous obtenons nos coefficients a_n
Ensuite afin d'étudier la fonction sur l'intervalle [-1,1] nous nous intéressons à sin(xπ).

Il nous est simplement nécéssaire de calculer les coefficients pairs car cette dernière fonction est paire. Nous décidons de faire les calculs de ces coefficients en Python qui permet de mener plus aisément une étude statistique. En utilisant la fonction integrate.quad qui utilise la librairie Fortran library QUADPACK. Nous obtenons notre premier jeu de coefficients, qui nous permettent de reconstruire l'approximation de sinus en la corrigeant du polynôme de construction et en annulant le changement de paramètre.

A noter, que nous devons stocker en dur les derniers coefficients dans la fonction approximationsin() car Python est incapable de simplifier le résultat obtenu en transformation2(). Nous devons donc effectuer ces calculs manuellement.

```
#!/usr/bin/env python3

from scipy import integrate
from math import sin,pi,sqrt
from sympy import symbols

def transformation1():
    return [(integrate.quad(lambda x : (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))\
    /(sqrt(1-x**2)),-1,0)[0]+integrate.quad(lambda x : (sin(pi*x)/(8/(3*pi)*pi*x\
    -8/(3*pi**3)*(pi*x)**3))/(sqrt(1-x**2)),0,1)[0])/pi,(integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(2*x**2-1)/(sqrt(1-x**2)),-1,0)\
    [0]+integrate.quad(lambda x : (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**\
    3))*(2*x**2-1)/(sqrt(1-x**2)),0,1)[0])*2/pi,(integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(8*x**4-8*x**2+1)/(sqrt(1-x**2)\
    ),-1,0)[0]+integrate.quad(lambda x : (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*\
    (pi*x)**3))*(8*x**4-8*x**2+1)/(sqrt(1-x**2)),0,1)[0])*2/pi,(integrate.quad\
    (lambda x : (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(32*x**6-48*\
    x**4+18*x**2-1)/(sqrt(1-x**2)),-1,0)[0]+integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(32*x**6-48*x**4+18*x**2-1)/\
    (sqrt(1-x**2)),-0,1)[0])*2/pi,(integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(128*x**8-256*x**6+160*x**4-32\
    *x**2+1)/(sqrt(1-x**2)),-1,0)[0]+integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(128*x**8-256*x**6+160*x**4-32\
    *x**2+1)/(sqrt(1-x**2)),0,1)[0])*2/pi,(integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(512*x**10-1280*x**8+1120*x**6\
    -400*x**4+50*x**2-1)/(sqrt(1-x**2)),-1,0)[0]+integrate.quad(lambda x :\
     (sin(pi*x)/(8/(3*pi)*pi*x-8/(3*pi**3)*(pi*x)**3))*(512*x**10-1280*x**8+1120*x**6\
    -400*x**4+50*x**2-1)/(sqrt(1-x**2)),0,1)[0])*2/pi]

def transformation2():
    x = symbols("x")
    poly_3 = 0
    T = transformation1()
    Cheby_2 = [1, 2*(x/pi)**2-1,8*(x/pi)**4-8*(x/pi)**2+1,32*(x/pi)**6-48*(x/pi)**4+18*(x/pi)**2-1\
    ,128*(x/pi)**8-256*(x/pi)**6+160*(x/pi)**4-32*(x/pi)**2+1,512*(x/pi)**10-1280\
    *(x/pi)**8+1120*(x/pi)**6-400*(x/pi)**4+50*(x/pi)**2-1]
    for i in range(len(T)):
        poly_3 += T[i]*Cheby_2[i]
    poly_4 = (poly_3)*((8/(3*pi))*x-(8/(3*pi**3))*x**3)
    return poly_4

def approximation_sin():
    coeff = [0.99999999708279,0,-0.1666666450029,0,0.00833330578942,0,-0.000198399079322,0,\
```

```
    2.75242860644e-6,0,2.46295041543e-8,0,1.32910789005e-10]
    x = symbols("x")
    poly = 0
    for i in range(len(coeff)):
        poly += coeff[i] * x**(i+1)
    return poly

def mon_sin(x):
    return 0.9999999970827936*x - 0.1666666450030143*x**3 + 0.008333305789420957\
    *x**5 -0.0001983990793222505 2*x**7 + 2.7524286064437253e-06*x**9\
     -2.462950415436452 5e-08*x**11 + 1.3291078900480108e-10*x**13
```

2.3 Etude de l'approximation proposée

Dans un premier temps nous nous proposons d'étudier l'écart relatif entre entre la fonction sinus implementée en Python et notre fonction sinus.

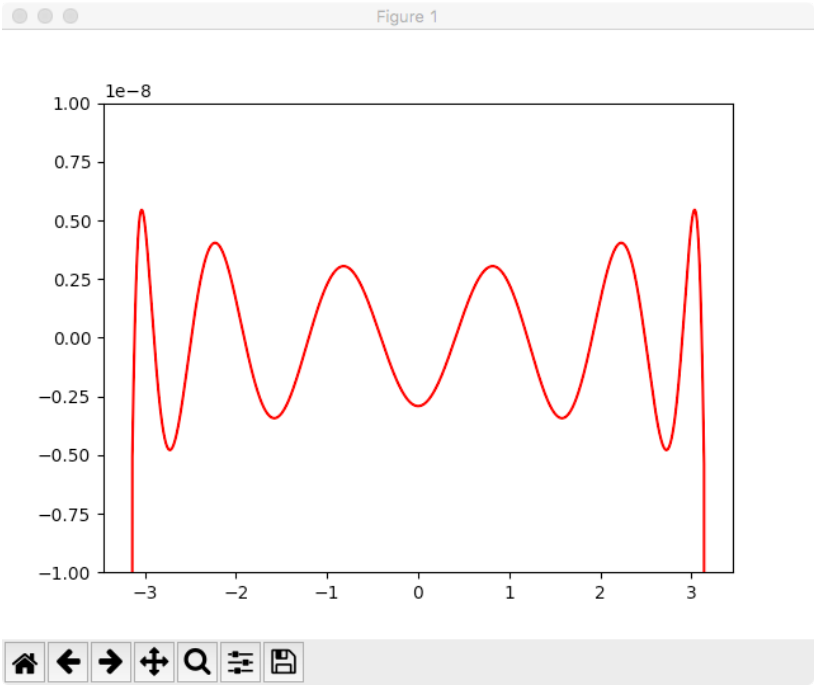

Figure 1: Ecart relatif sinus

On peut constater que l'écart relatif est très faible de l'ordre de 10-9 et est également distribué au contraire par exemple des séries de Taylor qui auraient une forte convergence au centre de l'intervalle et assez peu à ses extrémités.

On propose une petite étude statistique sur [-pi/2,pi/2] afin d'exclure le cas pi, -pi.

L'étude se fait en excluant les valeurs NaN (dont la présence témoigne de la bone précision de notre fonction)

nanmax	3.429128536680537e-09
nanmin	2.0935054242642857e-12
nanmean	1.9636880530194773e-09
nanstd	9.60473880179682e-10
nanvar	9.225100745074141e-19

Figure 2: Résumé statistique

3 Fonction cosinus

3.1 Fonction de Bessel

Les fonctions de Bessel du premier ordre sont solutions de l'equation différentielle de Bessel. Avec Γ la fonction gamma d'Euler.

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

En constatant que :

$$e^{\left(\frac{x}{2}\right)\left(t-\frac{1}{t}\right)} = \sum_{n=-\infty}^{\infty} J_n(x)t^n$$

et que :

cos(x sin ϕ) + isin(x sin ϕ) = $e^{ixsin\phi}$

puis en posant ϕ =π/2 On peut démontrer que :

cos(x) = $J_0(x) + 2\sum_{n=-\infty}^{\infty}(-1)^n J_n(x)$

3.2 Proposition d'approximation de la fonction cosinus

En utilisant le résultat précédent on propose une approximation de la fonction cosinus.

```
from math import factorial,gamma
from sympy import symbols

def J(n):
    x = symbols("x")
    tab = [(-1)**j*(x/2)**(2*j+n)/(factorial(j)*gamma(j+1+n)) for j in range(20)]
    return sum(tab)

def mon_poly_gamma():
    x = symbols("x")
    base = [2*(-1)**n*J(2*n) for n in range(1,20)]
    return J(0) + sum(base)

def mon_cos_gamma(x):
    return 5.36923126214664e-117*x**76 - 9.18138545827075e-113*x**74 + \
    1.19476512039168e-108*x**72 - 1.37236145747945e-104*x**70 + \
    1.43887270901671e-100*x**68 - 1.38769242112892e-96*x**66 + \
```

```
1.23127942085417e-92*x**64 - 1.00279166468403e-88*x**62 + \
7.47199933967046e-85*x**60 - 5.07455387910634e-81*x**58 + \
3.12818379283804e-77*x**56 - 1.74245784661659e-73*x**54 + \
8.72747376097333e-70*x**52 - 3.91004418776769e-66*x**50 + \
1.55803098471521e-62*x**48 - 5.48821278213621e-59*x**46 + \
1.69802377065023e-55*x**44 - 4.58338898467404e-52*x**42 + \
1.07196093802408e-48*x**40 - 1.91196320504028e-45*x**38 + \
2.68822026628664e-42*x**36 - 3.38715753552116e-39*x**34 + \
3.80039075485474e-36*x**32 - 3.76998762881591e-33*x**30 + \
3.27988923706984e-30*x**28 - 2.4795962632248e-27*x**26 + \
1.61173757109612e-24*x**24 - 8.89679139245057e-22*x**22 + \
4.11031762331216e-19*x**20 - 1.56192069685862e-16*x**18 + \
4.77947733238739e-14*x**16 - 1.14707455977297e-11*x**14 + \
2.08767569878681e-9*x**12 - 2.75573192239859e-7*x**10 + \
2.48015873015873e-5*x**8 - 0.0013888888888889*x**6 + \
0.0416666666666667*x**4 - 0.5*x**2 + 1.0
```

3.3 Etude de l'approximation proposée

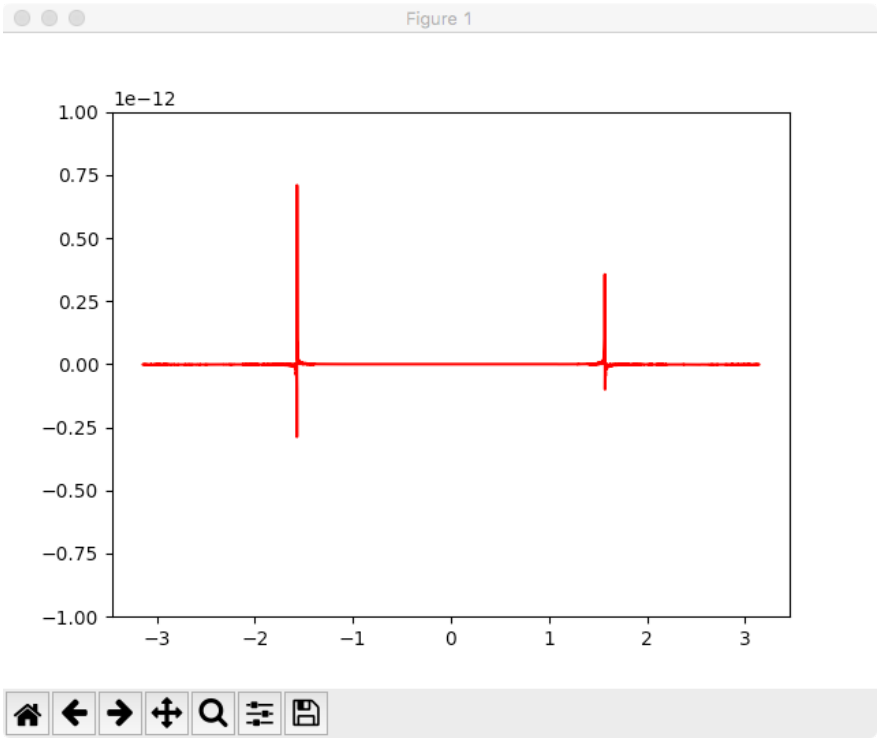

Figure 3: Ecart relatif cosinus

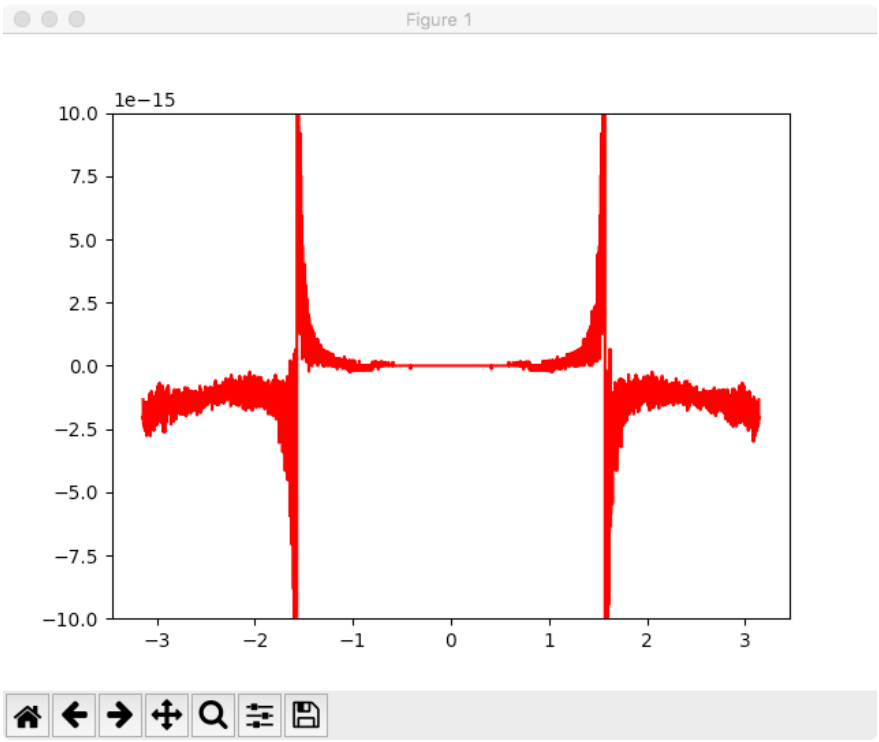

Figure 4: Ecart relatif cosinus

nanmax	1.5642066678596621e-16
nanmin	0.0
nanmean	4.525590718527718e-18
nanstd	2.4365221861678385e-17
nanvar	5.9366403636881025e-34

Figure 5: Résumé statistique dans -pi/4 pi/4

4 Fonction tan

En utilisant nos fonctions cos et sin on peut construire la fonction tangente.

4.1 Etude de l'approximation proposée

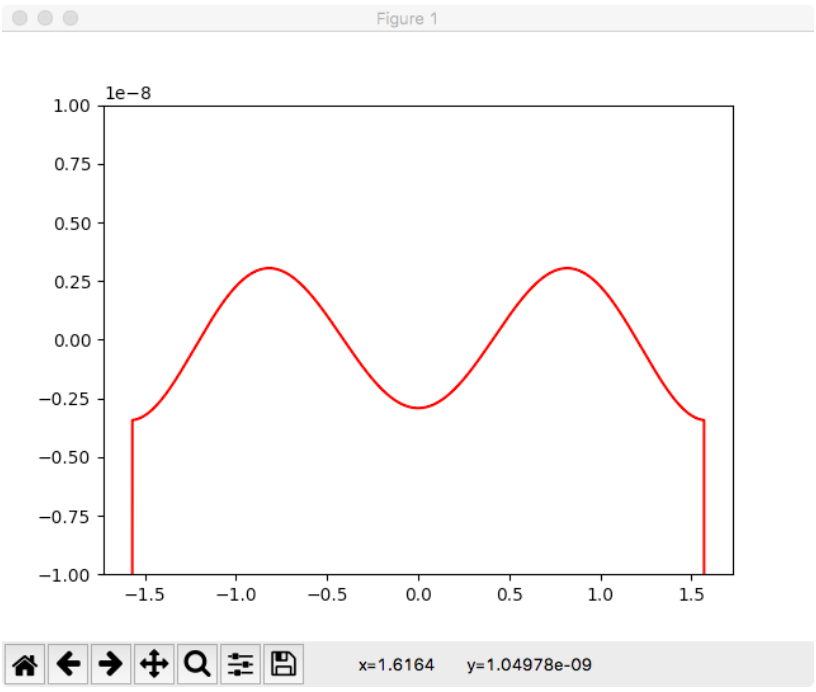

Figure 6: Ecart relatif tan

nanmax	3.0211814161518195e-09
nanmin	2.128209052886472e-13
nanmean	1.8464321329965337e-09
nanstd	9.069088335169482e-10
nanvar	8.224836323110717e-19

Figure 7: Résumé statistique sur -$\pi/4\pi/4$

5 Fonction arctan

5.1 Algorithme de Cordic

Partant d'un vecteur 2x1, on calcule un nouveau vecteur en le multipliant avec la matrice de rotation initialisé avec un angle de 45 degrés :

$v_{i+1} = R_i v_i$ $v_{i+1} = R_i v_i$

avec :

$$R_i = \begin{pmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{pmatrix}$$

$$v_{i+1} = R_i v_i = \cos\gamma_i \begin{pmatrix} 1 & -\sigma_i \tan\gamma_i \\ \sigma_i \tan\gamma_i & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Les rotations successives permettent de converger vers l'angle désiré.

On peut limiter le choix des angles de telle façon que $1/\tan()$ soit une puissance de 2. Dans notre cas, on stocke ces données.

$$v_{i+1} = R_i v_i = \cos(\arctan(2^{-i})) \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = K_i \begin{pmatrix} x_i - \sigma_i 2^{-i} y_i \\ x_i \sigma_i 2^{-i} + y_i \end{pmatrix}$$

avec $K_i = \cos(\arctan(2^{-i}))$

$\beta_{i+1} = \beta_i - \sigma_i\gamma_i. \quad \gamma_i = \arctan 2^{-i} \beta_{i+1} = \beta_i - \sigma_i\gamma_i. \quad \gamma_i = \arctan 2^{-i}$

En étudiant le signe de la deuxième composante, on décide sil'on effectue une rotation horaire ou anti-horaire. Obtenir arctan(a) en utilisant l'algorithme de cordic nécéssite d'initialiser notre vecteur v0 à (1,a) et d'utiliser le triplet :

$$\begin{cases} x_{k+1} = x_k - \sigma_k y_k 2^{-k} \\ y_{k+1} = y_k + \sigma_k x_k 2^{-k} \\ z_{k+1} = z_k - \sigma_k \varepsilon_k \end{cases}$$

Avec ϵ_k nos arctan enregistrées. Nous renvoyons z.

5.2 Proposition d'approximation de la fonction arctan

En utilisant le résultat précédent on propose une approximation de la fonction arctan.

```
def cordit_arctan(a):
    nos_arctan  =[0.7853981633974483, 0.4636476090008061, 0.24497866312686414, \
    0.12435499454676144, 0.06241880999595735, 0.031239833430268277, 0.015623728620476831, \
    0.007812341060101111, 0.0039062301319669718, 0.0019531225164788188, \
    0.0009765621895593195, 0.0004882812111948983, 0.00024414062014936177, \
    0.00012207031189367021, 6.103515617420877e-05, 3.0517578115526096e-05, \
    1.5258789061315762e-05, 7.62939453110197e-06, 3.814697265606496e-06, \
    1.907348632810187e-06, 9.536743164059608e-07, 4.7683715820308884e-07, \
    2.3841857910155797e-07, 1.1920928955078068e-07, 5.960464477539055e-08, \
    2.9802322387695303e-08, 1.4901161193847655e-08, 7.450580596923828e-09, \
    3.725290298461914e-09, 1.862645149230957e-09, 9.313225746154785e-10, \
    4.656612873077393e-10, 2.3283064365386963e-10, 1.1641532182693481e-10, \
    5.820766091346741e-11, 2.9103830456733704e-11, 1.4551915228366852e-11, \
    7.275957614183426e-12, 3.637978807091713e-12, 1.8189894035458565e-12]
    x, y, z = 1, a, 0
    t = 1;
    for i in range(len(nos_arctan)):
        x1 = 0
        if (y < 0):
            x1 = x - y*t
            y = y + x*t
            z = z - nos_arctan[i]
        else:
            x1 = x + y*t
            y = y - x*t
            z = z + nos_arctan[i]
        x = x1
        t /= 2
    return z
```

5.3 Etude de l'approximation proposée

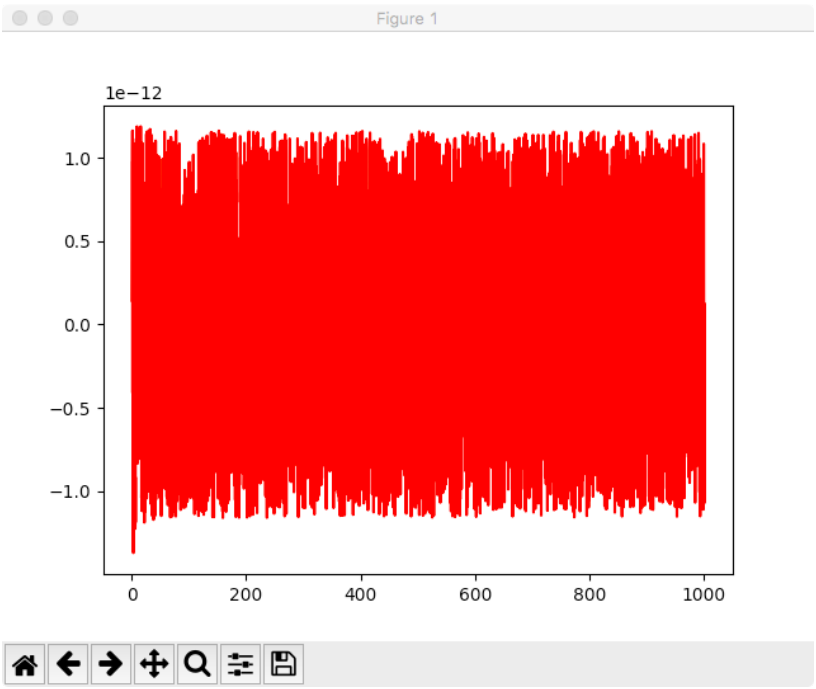

Figure 8: Ecart relatif arctan

nanmax	1.3700069980077967e-12
nanmin	2.8293021336720964e-16
nanmean	5.75558605593629e-13
nanstd	3.318803038385423e-13
nanvar	1.1014453607596315e-25

Figure 9: Résumé statistique

6 Fonction arcsin

6.1 Utilisation de arctan

Nous implementons arcsin en utilisant l'égalité $\arcsin(x) = \arctan(x/\sqrt{1-x^2})$, qui nécéssite donc un algorithme de racine carré que nous nous proposons d'implémenter.

6.2 Approximation de racine carrée

Afin d'approcher la racine carrée, nous utilisons simplement la méthode de Newton, pour trouver les zéros d'une fonction.

$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

ce qui donne dans notre cas de figure :

$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$

Afin d'obtenir un algorithme rapide nous limitons le nombre d'itération à 1000, ce qui est amplement suffisant pour calculer des racines carrées très grandes.

Nous proposons l'implémentation suivante.

```python
def ma_racine_caree(a):
    x = 1
    maximum_iteration = min(int(a)//100 if a > 1000 else 40,1000)
    for i in range(maximum_iteration):
        x = 1/2*(x+a/x)
    return x
```

6.3 Etude de l'approximation proposée : sqrt

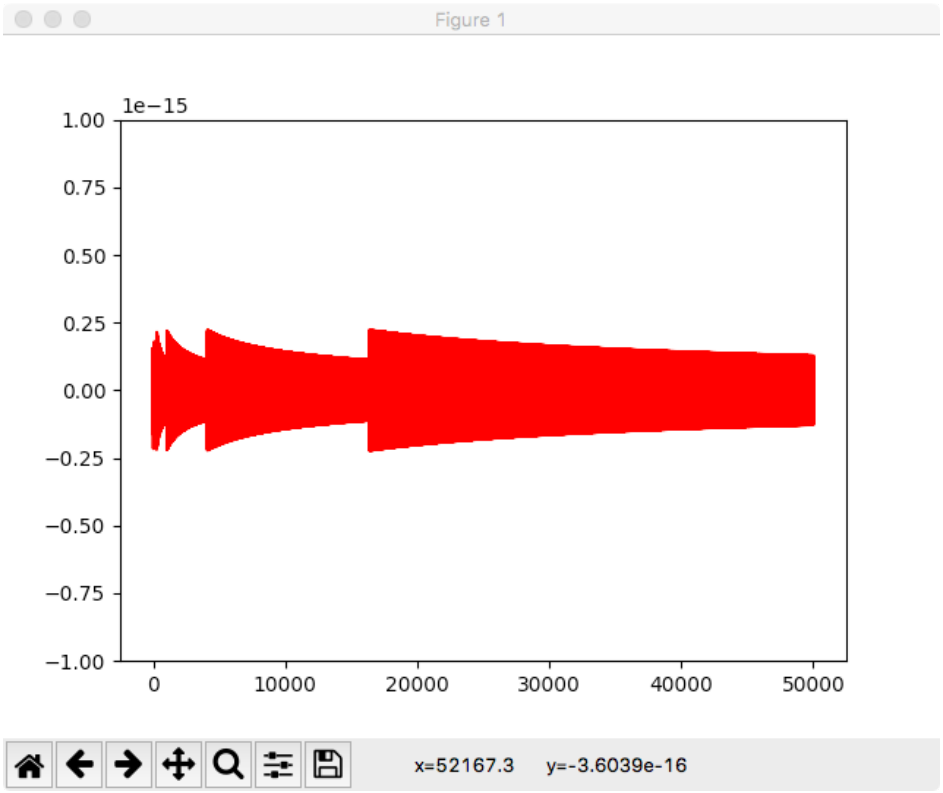

Figure 10: Ecart relatif racine carrée

nanmax	2.2135630626480553e-16
nanmin	0.0
nanmean	3.571197755392954e-17
nanstd	6.494591851406062e-17
nanvar	4.217972331635003e-33

Figure 11: Résumé statistique

6.4 Etude de l'approximation proposée : arcsin

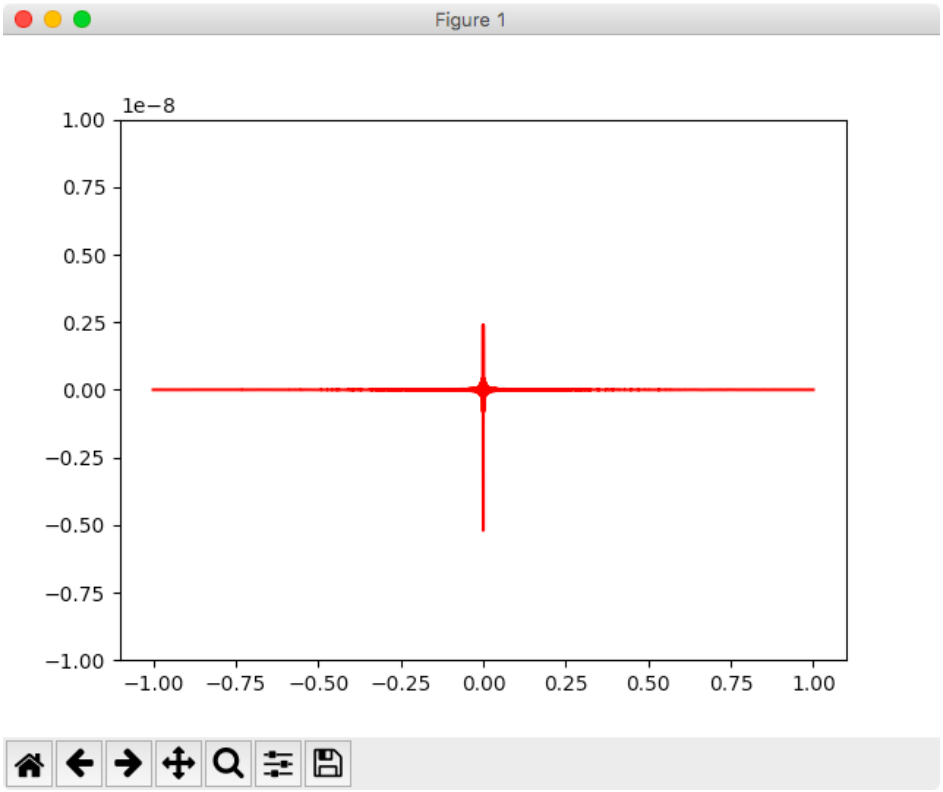

Figure 12: Ecart relatif arcsin

nanmax	5.203623314460027e-09
nanmin	5.799526106855611e-16
nanmean	9.785894575227016e-12
nanstd	1.213994110192895e-10
nanvar	1.4737816995830394e-20

Figure 13: Résumé statistique

7 Fonction arccos

7.1 Utilisation de arctan

Nous implementons arccos en utilisant l'égalité $\arccos(x) = \arctan(\sqrt{1-x^2}/x)$.

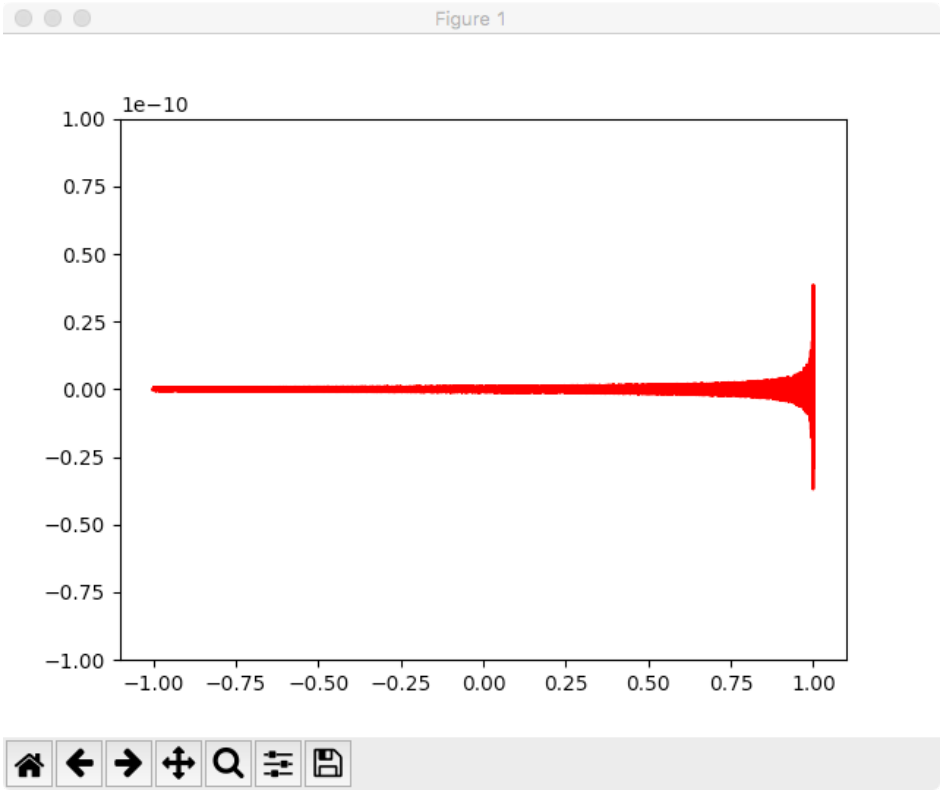

Figure 14: Ecart relatif arccos

7.2 Etude de l'approximation proposée

7.3 Fonction ULP

Pour l'instant la fonction ULP ne fonctionne que pour des flottant codés sur 32 bits c'est à dire avec 1 bit dédié à l'exposant, 8 pour l'exposant et 23 pour la mantisse suivant la norme IEE 754.

7.4 Norme IEE 754

Figure 15: représentation floatant

Une fois cette représentation comprise, on comprend bien qu'il est tout à fait possible de représenter tout les floatants réprésentables par la machine sur une droite des réels.

La fonction ulp apparait comme une évidence car lorsque qu'un réel est codé, il se trouve en fait dans un intervalle autour du floatant obtenu. On utilisise alors la définition suivante de la fonction ULP: C'est la distance entre le prochain (ie: supérieur) floatant codable par la machine et le floatant codé par la machine.

7.5 ULP

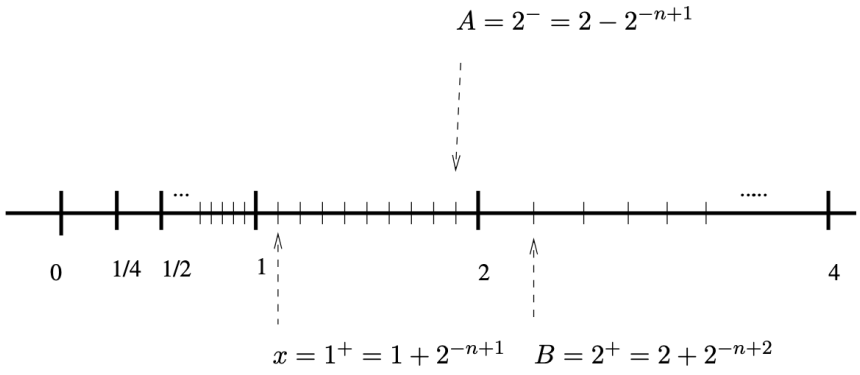

Figure 16: ULP

Pour utiliser la fonction ulp, il suffit de lancer le script ulp.deca, un appel de méthode sera nécessaire lorsque l'on aura implémenté les classes.