Fonction ξ de Riemann et *vdp* de van der Pol en python (Denise Vella-Chemla, avril 2024)

1) Dessiner le graphe de la fonction ξ de Riemann

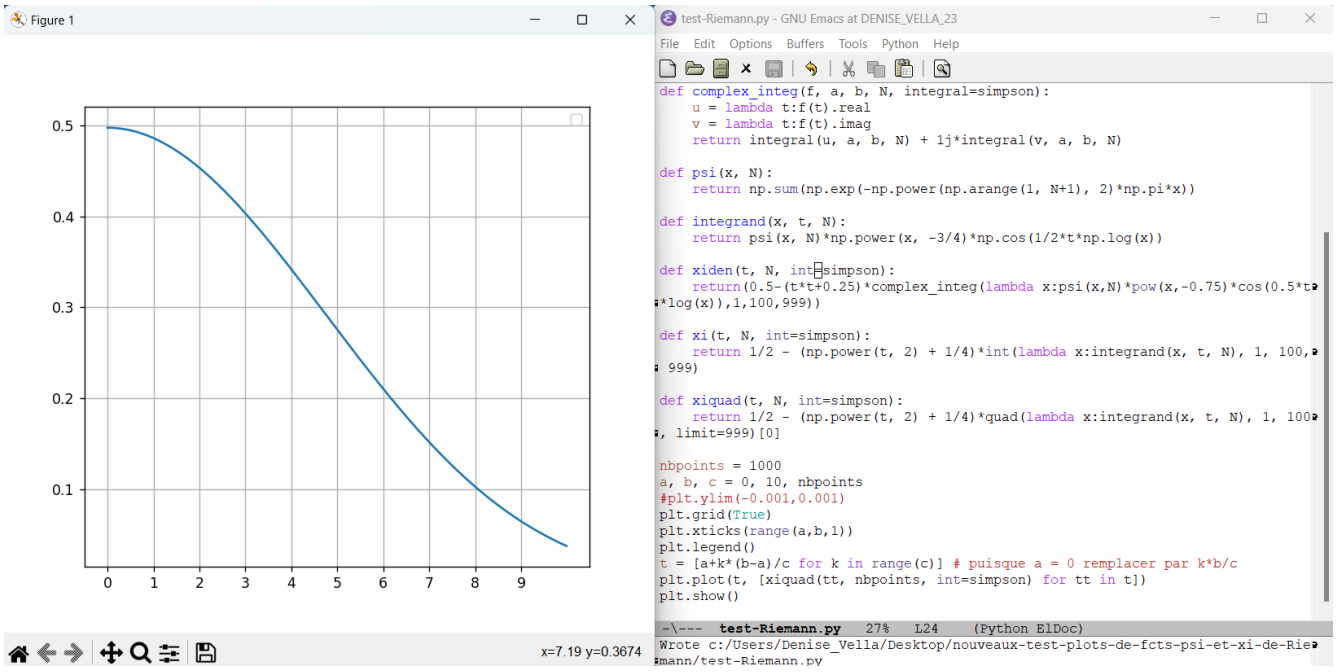
On essaie de programmer la fonction ξ de l'article énonçant l'hypothèse de Riemann. Il est écrit en note, dans la traduction en français de l'article en question, que la phrase "*il est très probable que toutes les racines* [de cette fonction ξ] *sont réelles.*" est la première formulation de l'hypothèse de Riemann.

Voici la définition de la fonction ξ :

$$\xi(t) = \frac{1}{2} - \left(t^2 + \frac{1}{4}\right) \int_1^{\infty} \psi(x) x^{-\frac{3}{4}} \cos\left(\frac{1}{2} t \log x\right) dx$$

avec $\psi(x) = \sum_1^{\infty} e^{-n^2 \pi x}$.

On dessine le graphe de cette fonction pour vérifier du moins pour les petites valeurs, qu'elle s'annule sur les mêmes points que ζ. Voici ce qu'on obtient (de 0 à 10, puis de 10 à 20, etc...).



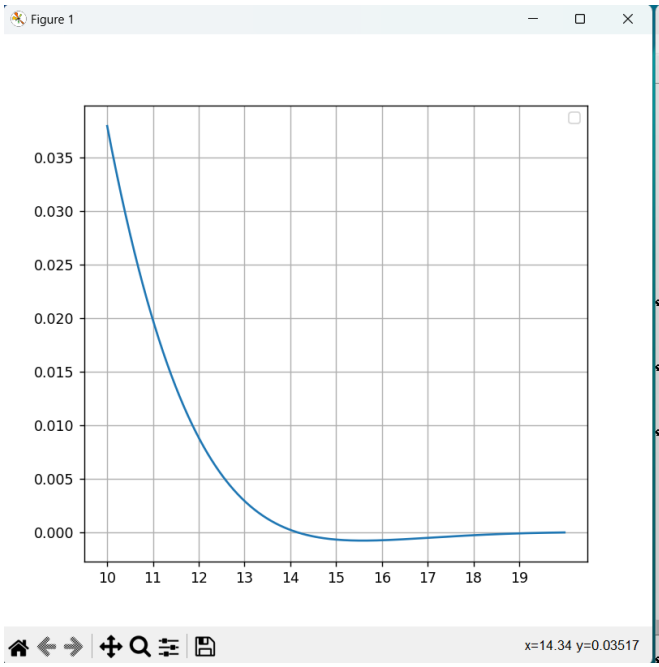
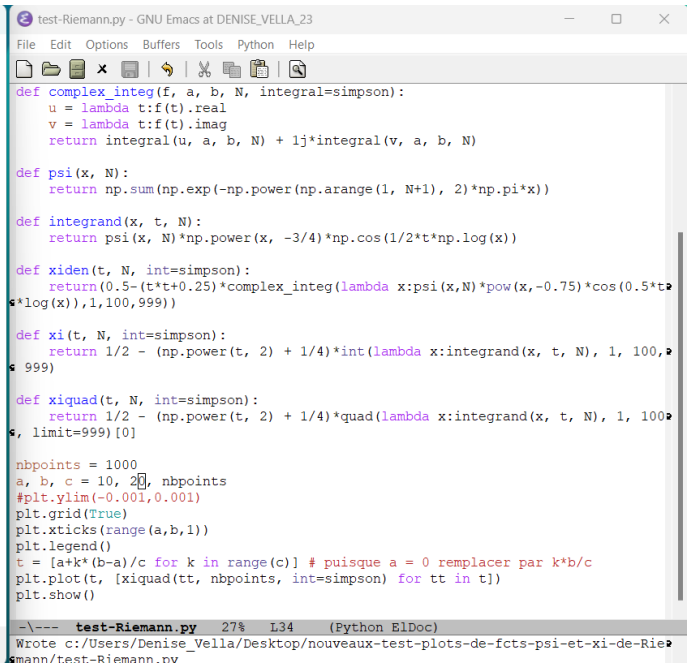


Figure 1
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
10
11
12
13
14
15
16
17
18
19
x=14.34 y=0.03517


test-Riemann.py - GNU Emacs at DENISE_VELLA_23
File Edit Options Buffers Tools Python Help
def complex_integ(f, a, b, N, integral=simpson):
u = lambda t:f(t).real
v = lambda t:f(t).imag
return integral(u, a, b, N) + 1j*integral(v, a, b, N)
def psi(x, N):
return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))
def integrand(x, t, N):
return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))
def xiden(t, N, int=simpson):
return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*log(x)),1,100,999))
def xi(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100, 999)
def xiquad(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100, limit=999)[0]
nbpoints = 1000
a, b, c = 10, 20, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
-\--- test-Riemann.py 27% L34 (Python ElDoc)
Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Riemann/test-Riemann.py

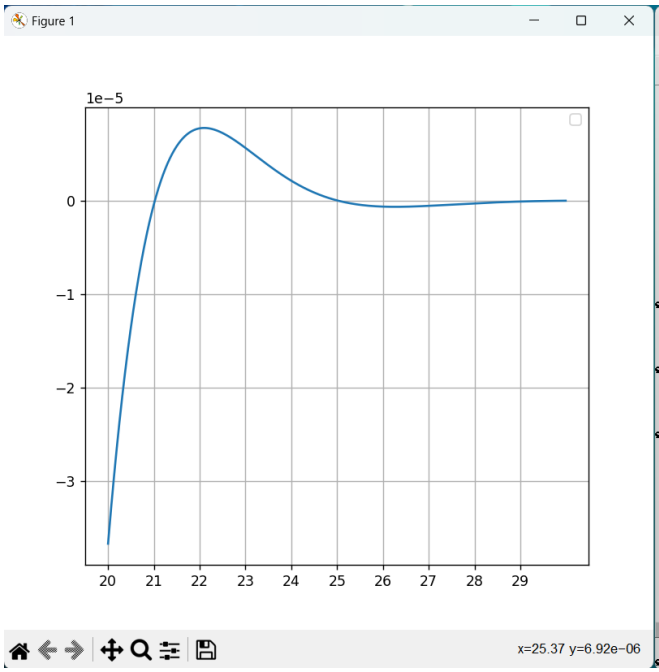
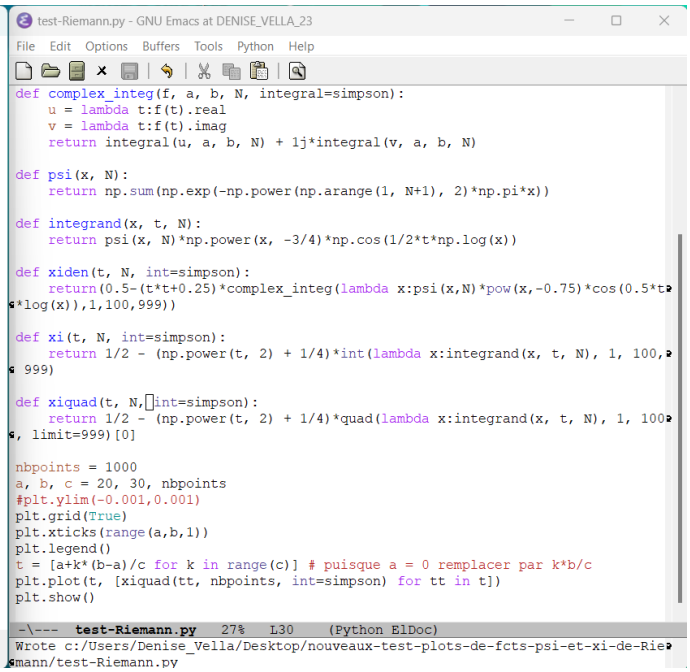
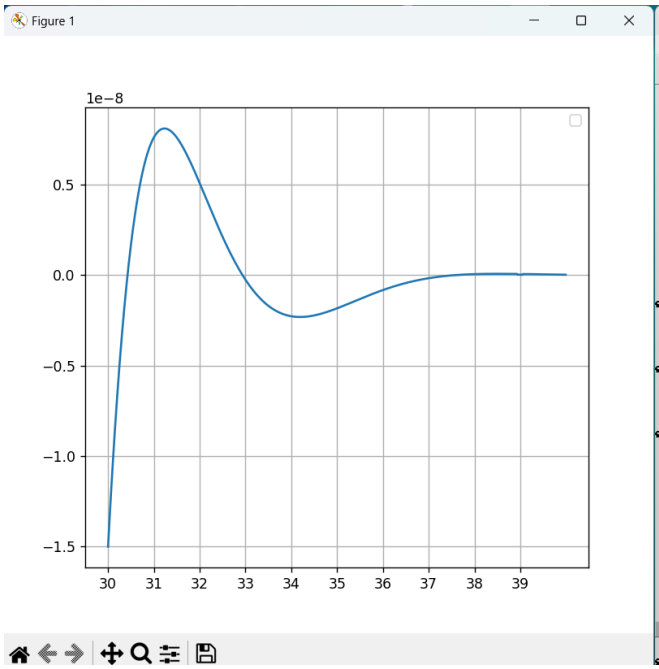


Figure 1
1e−5
0
−1
−2
−3
20
21
22
23
24
25
26
27
28
29
x=25.37 y=6.92e−06


test-Riemann.py - GNU Emacs at DENISE_VELLA_23
File Edit Options Buffers Tools Python Help
def complex_integ(f, a, b, N, integral=simpson):
u = lambda t:f(t).real
v = lambda t:f(t).imag
return integral(u, a, b, N) + 1j*integral(v, a, b, N)
def psi(x, N):
return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))
def integrand(x, t, N):
return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))
def xiden(t, N, int=simpson):
return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*log(x)),1,100,999))
def xi(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100, 999)
def xiquad(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100, limit=999)[0]
nbpoints = 1000
a, b, c = 20, 30, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
-\--- test-Riemann.py 27% L30 (Python ElDoc)
Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Riemann/test-Riemann.py



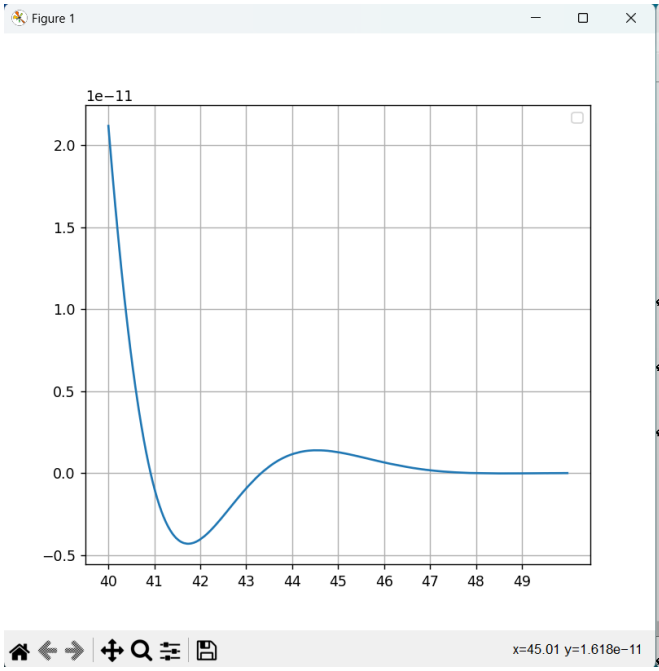
test-Riemann.py - GNU Emacs at DENISE_VELLA_23

File Edit Options Buffers Tools Python Help

```
def complex_integ(f, a, b, N, integral=simpson):
    u = lambda t:f(t).real
    v = lambda t:f(t).imag
    return integral(u, a, b, N) + 1j*integral(v, a, b, N)

def psi(x, N):
    return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))

def integrand(x, t, N):
    return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))

def xiden(t, N, int=simpson):
    return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*
*log(x)),1,100,999))

def xi(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100,
 999)

def xiquad(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100
, limit=999)[0]

nbpoints = 1000
a, b, c = 30, 40, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
```

-\--- **test-Riemann.py** 27% L34 (Python ElDoc)

Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Rie
mann/test-Riemann.py



test-Riemann.py - GNU Emacs at DENISE_VELLA_23

File Edit Options Buffers Tools Python Help

```
def complex_integ(f, a, b, N, integral=simpson):
    u = lambda t:f(t).real
    v = lambda t:f(t).imag
    return integral(u, a, b, N) + 1j*integral(v, a, b, N)

def psi(x, N):
    return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))

def integrand(x, t, N):
    return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))

def xiden(t, N, int=simpson):
    return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*
*log(x)),1,100,999))

def xi(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100,
 999)

def xiquad(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100
, limit=999)[0]

nbpoints = 1000
a, b, c = 40, 50, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
```

-\--- **test-Riemann.py** 27% L34 (Python ElDoc)

Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Rie
mann/test-Riemann.py

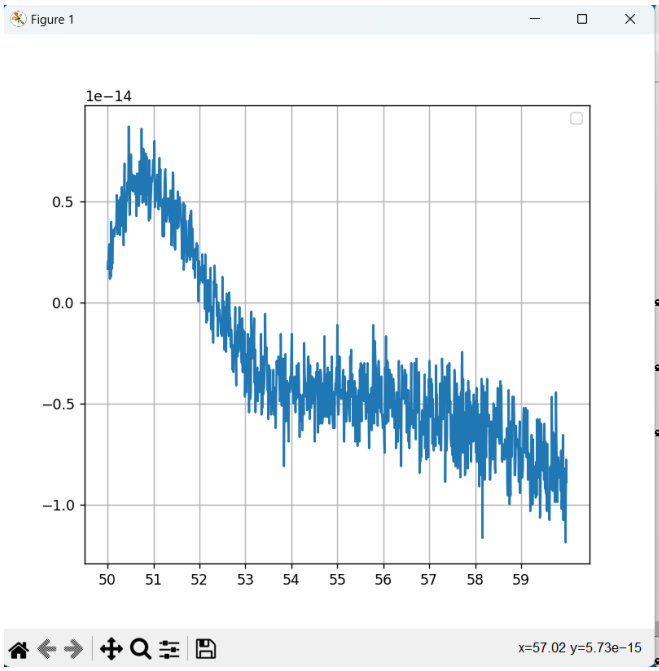
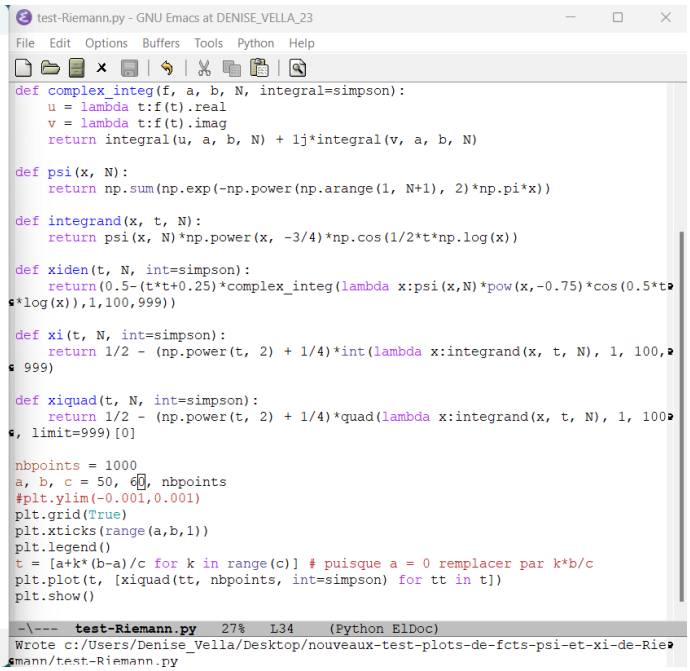


Figure 1
1e−14
0.5
0.0
−0.5
−1.0
50
51
52
53
54
55
56
57
58
59
x=57.02 y=5.73e−15


test-Riemann.py - GNU Emacs at DENISE_VELLA_23
File Edit Options Buffers Tools Python Help
def complex_integ(f, a, b, N, integral=simpson):
u = lambda t:f(t).real
v = lambda t:f(t).imag
return integral(u, a, b, N) + 1j*integral(v, a, b, N)
def psi(x, N):
return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))
def integrand(x, t, N):
return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))
def xiden(t, N, int=simpson):
return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t
*log(x)),1,100,999))
def xi(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100,
999)
def xiquad(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100
, limit=999)[0]
nbpoints = 1000
a, b, c = 50, 60, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
-\--- test-Riemann.py 27% L34 (Python ElDoc)
Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Rie
mann/test-Riemann.py

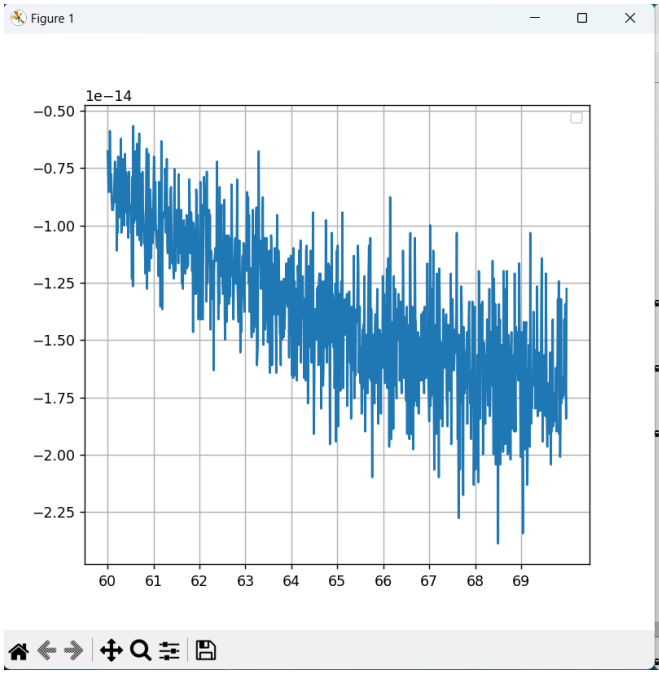
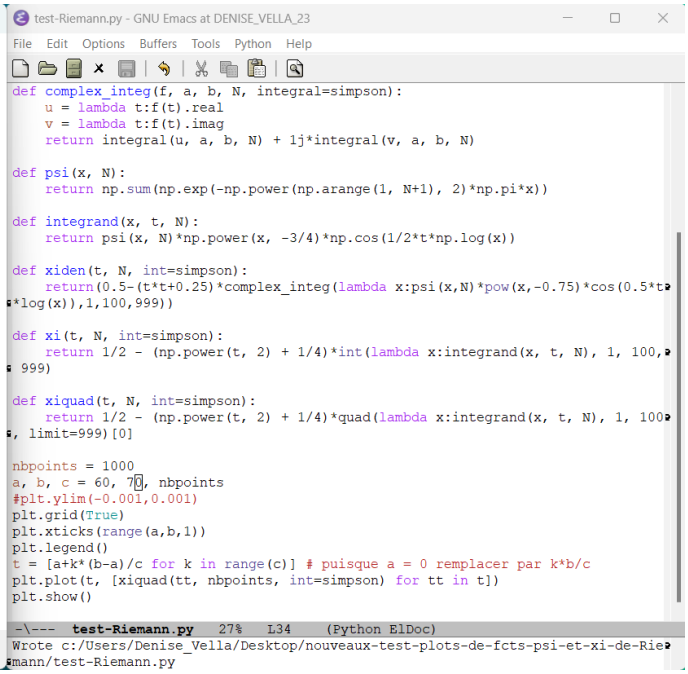
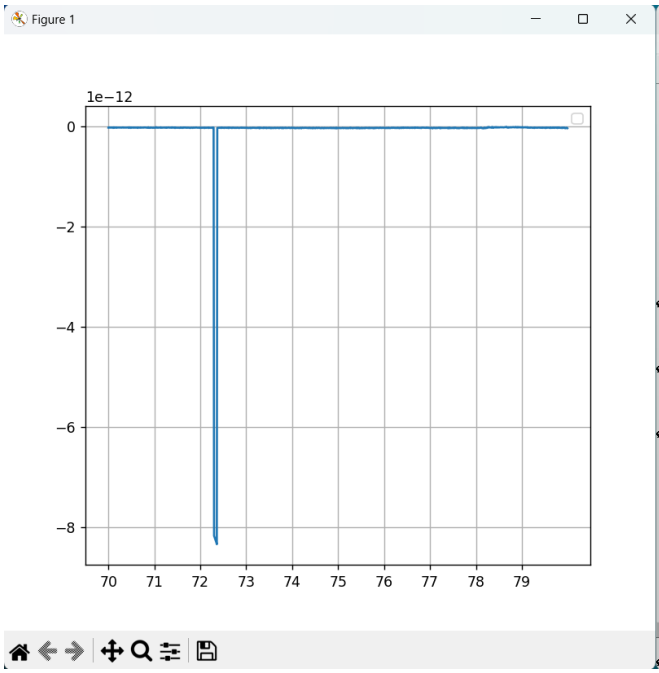


Figure 1
1e−14
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
−2.00
−2.25
60
61
62
63
64
65
66
67
68
69


test-Riemann.py - GNU Emacs at DENISE_VELLA_23
File Edit Options Buffers Tools Python Help
def complex_integ(f, a, b, N, integral=simpson):
u = lambda t:f(t).real
v = lambda t:f(t).imag
return integral(u, a, b, N) + 1j*integral(v, a, b, N)
def psi(x, N):
return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))
def integrand(x, t, N):
return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))
def xiden(t, N, int=simpson):
return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t
*log(x)),1,100,999))
def xi(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100,
999)
def xiquad(t, N, int=simpson):
return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100
, limit=999)[0]
nbpoints = 1000
a, b, c = 60, 70, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
-\--- test-Riemann.py 27% L34 (Python ElDoc)
Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Rie
mann/test-Riemann.py



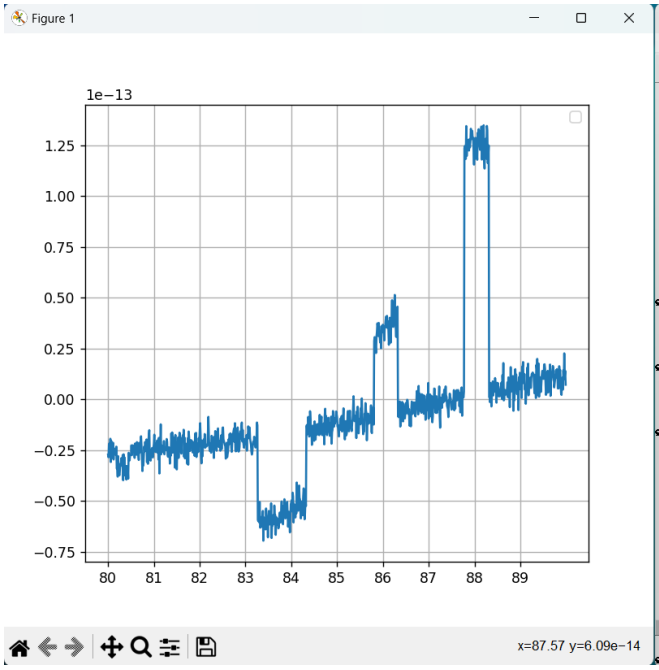
test-Riemann.py - GNU Emacs at DENISE_VELLA_23

File Edit Options Buffers Tools Python Help

```
def complex_integ(f, a, b, N, integral=simpson):
    u = lambda t:f(t).real
    v = lambda t:f(t).imag
    return integral(u, a, b, N) + 1j*integral(v, a, b, N)

def psi(x, N):
    return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))

def integrand(x, t, N):
    return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))

def xiden(t, N, int=simpson):
    return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*log(x)),1,100,999))

def xi(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100, 999)

def xiquad(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100, limit=999)[0]

nbpoints = 1000
a, b, c = 70, 80, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
```

-\--- **test-Riemann.py** 27% L34 (Python ElDoc)

Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Riemann/test-Riemann.py



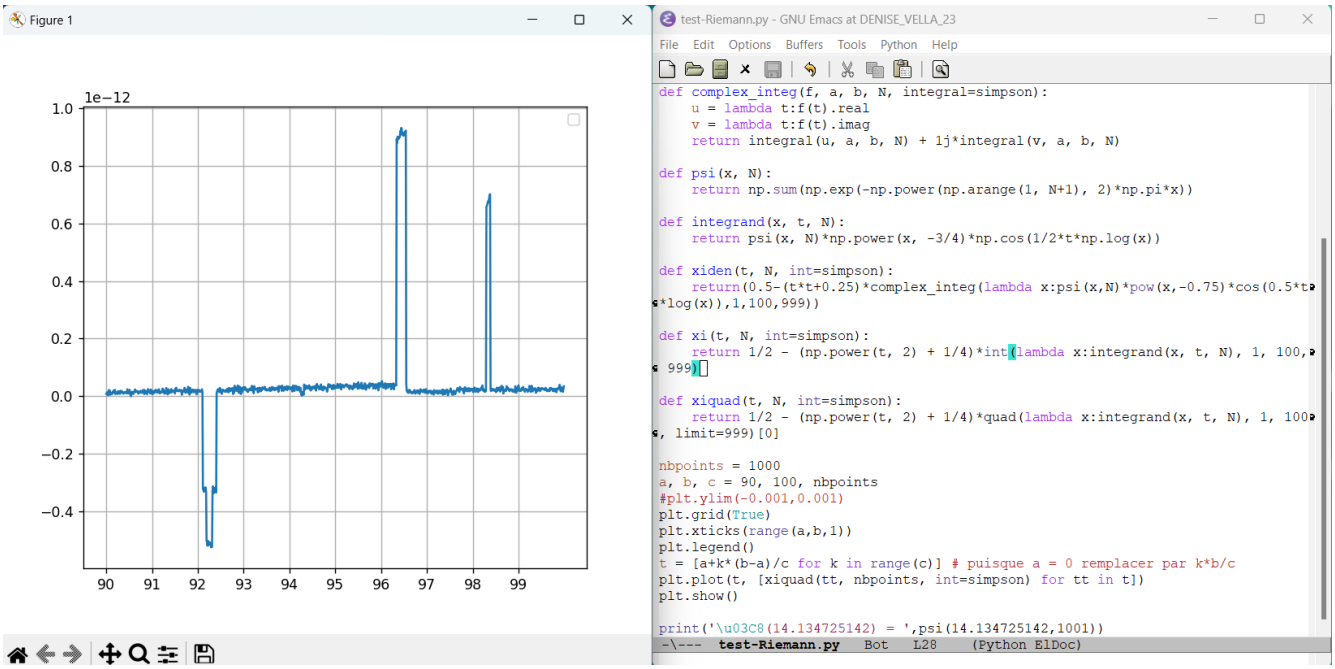
test-Riemann.py - GNU Emacs at DENISE_VELLA_23

File Edit Options Buffers Tools Python Help

```
def complex_integ(f, a, b, N, integral=simpson):
    u = lambda t:f(t).real
    v = lambda t:f(t).imag
    return integral(u, a, b, N) + 1j*integral(v, a, b, N)

def psi(x, N):
    return np.sum(np.exp(-np.power(np.arange(1, N+1), 2)*np.pi*x))

def integrand(x, t, N):
    return psi(x, N)*np.power(x, -3/4)*np.cos(1/2*t*np.log(x))

def xiden(t, N, int=simpson):
    return(0.5-(t*t+0.25)*complex_integ(lambda x:psi(x,N)*pow(x,-0.75)*cos(0.5*t*log(x)),1,100,999))

def xi(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*int(lambda x:integrand(x, t, N), 1, 100, 999)

def xiquad(t, N, int=simpson):
    return 1/2 - (np.power(t, 2) + 1/4)*quad(lambda x:integrand(x, t, N), 1, 100, limit=999)[0]

nbpoints = 1000
a, b, c = 80, 90, nbpoints
#plt.ylim(-0.001,0.001)
plt.grid(True)
plt.xticks(range(a,b,1))
plt.legend()
t = [a+k*(b-a)/c for k in range(c)] # puisque a = 0 remplacer par k*b/c
plt.plot(t, [xiquad(tt, nbpoints, int=simpson) for tt in t])
plt.show()
```

-\--- **test-Riemann.py** 27% L34 (Python ElDoc)

Wrote c:/Users/Denise_Vella/Desktop/nouveaux-test-plots-de-fcts-psi-et-xi-de-Riemann/test-Riemann.py



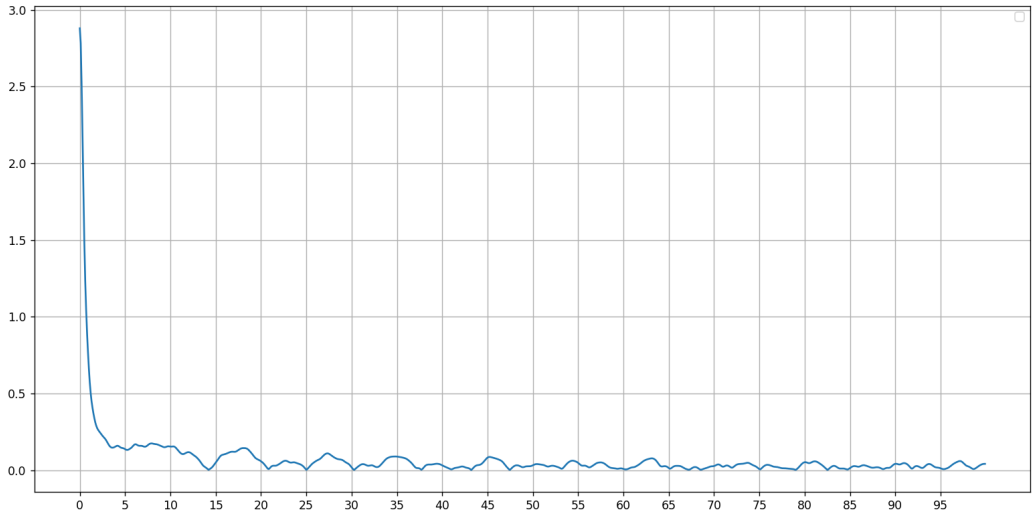
Les zéros ont bien l'air correct jusqu'à 50 mais à partir de là, le graphe devient totalement inexploitable. On abandonne ces tentatives de comprendre l'article de Riemann et la fonction ζ.

2) Dessiner le graphe de la fonction *vdp* de van der Pol

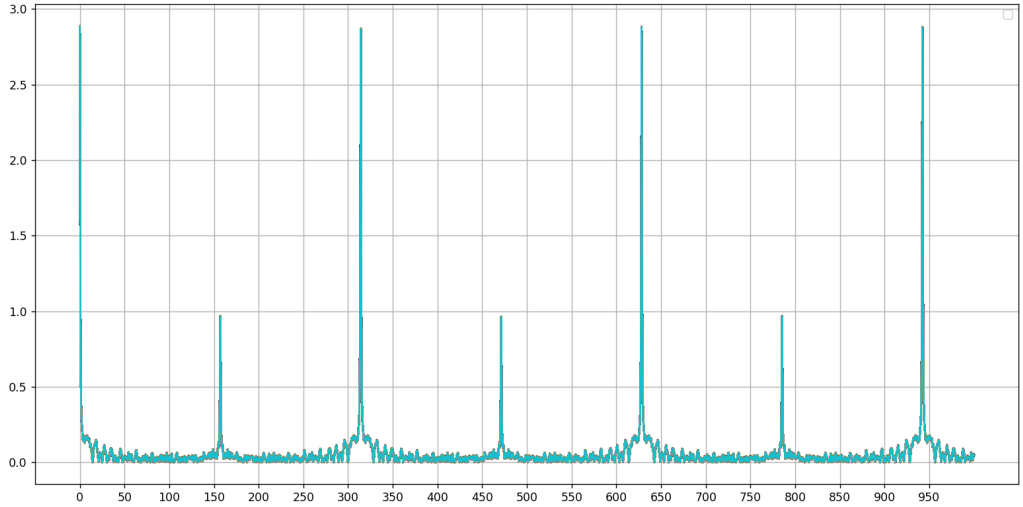
Si on se concentre sur la formule de la fonction *vdp* de van der Pol qui permet de retrouver les zéros de la fonction ζ de Riemann, et que l'on programme comme

$$vdp(t) = \sqrt{\left(\int_{x=-10}^{10} \frac{(\exp(x)\ \%\ 1) * \cos(x*t)}{\exp\left(\frac{x}{2}\right)}\right)^2 + \left(\int_{x=-10}^{10} \frac{(\exp(x)\ \%\ 1) * \sin(x*t)}{\exp\left(\frac{x}{2}\right)}\right)^2}$$

les zéros apparaissent bien approximativement.



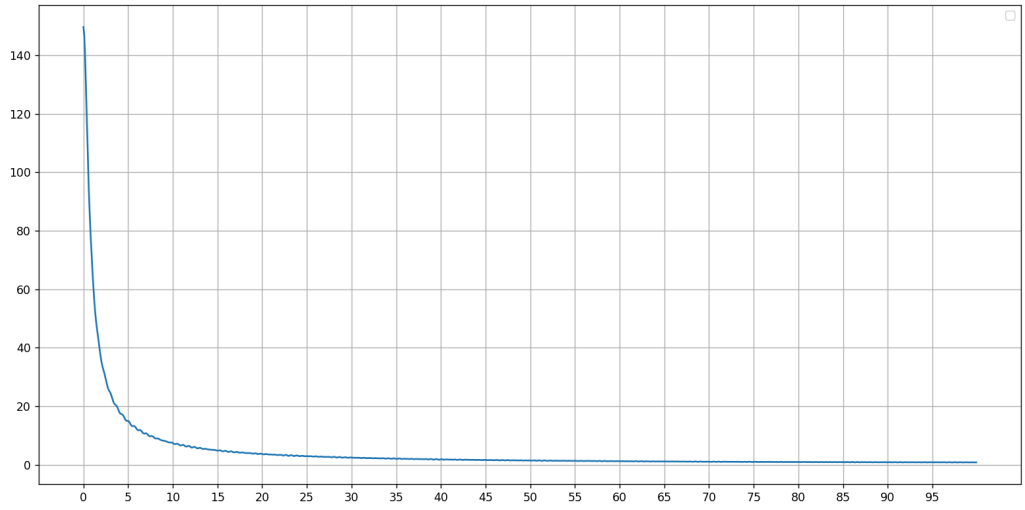
On voit les périodicités de cette formule en allant jusqu'à 1000.



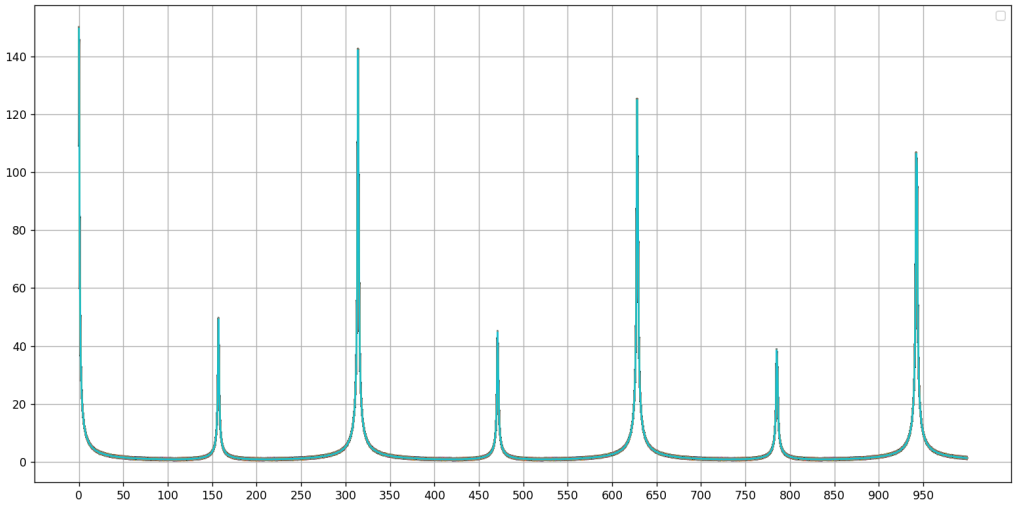
Il faut être cependant vigilant : bien qu'on ait une illusion de périodicité, avec des pics bien placés aux multiples de 157 ($\simeq 100 \times \frac{\pi}{2}$.), en fait, il y a une "accélération des zéros" ; il y a 57 (à gauche du pic central à 157) et 90 (à droite du pic central) zéros dans la première tranche jusqu'à 314 ; dans la seconde tranche, il y en a 102 et 110 et dans la troisième tranche, il y en a 118 et 122. Pour simuler ce comportement, il nous faudrait trouver une fonction assez périodique mais dont le rythme d'annulation s'accélère lentement.

Si l'on teste (pour éliminer le modulo 1, qui est une fonction scie problématique) le remplacement du x mod 1 (noté x % 1) par son approximation qui est $-\sum_{1}^{10000} \frac{\sin(2\pi kx)}{k\pi}$, alors les zéros disparaissent.

On voit mieux leur disparition en n'allant que jusqu'à 100.



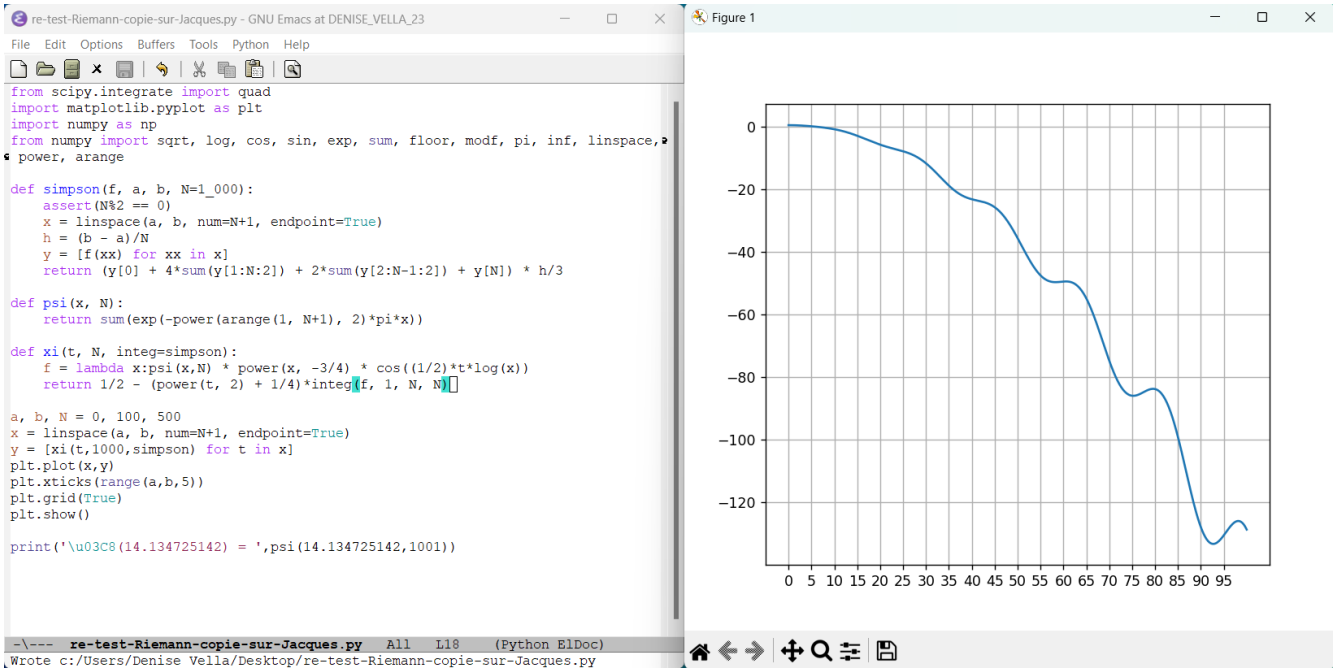
Dernière exécution avec des abscisses plus lisibles.



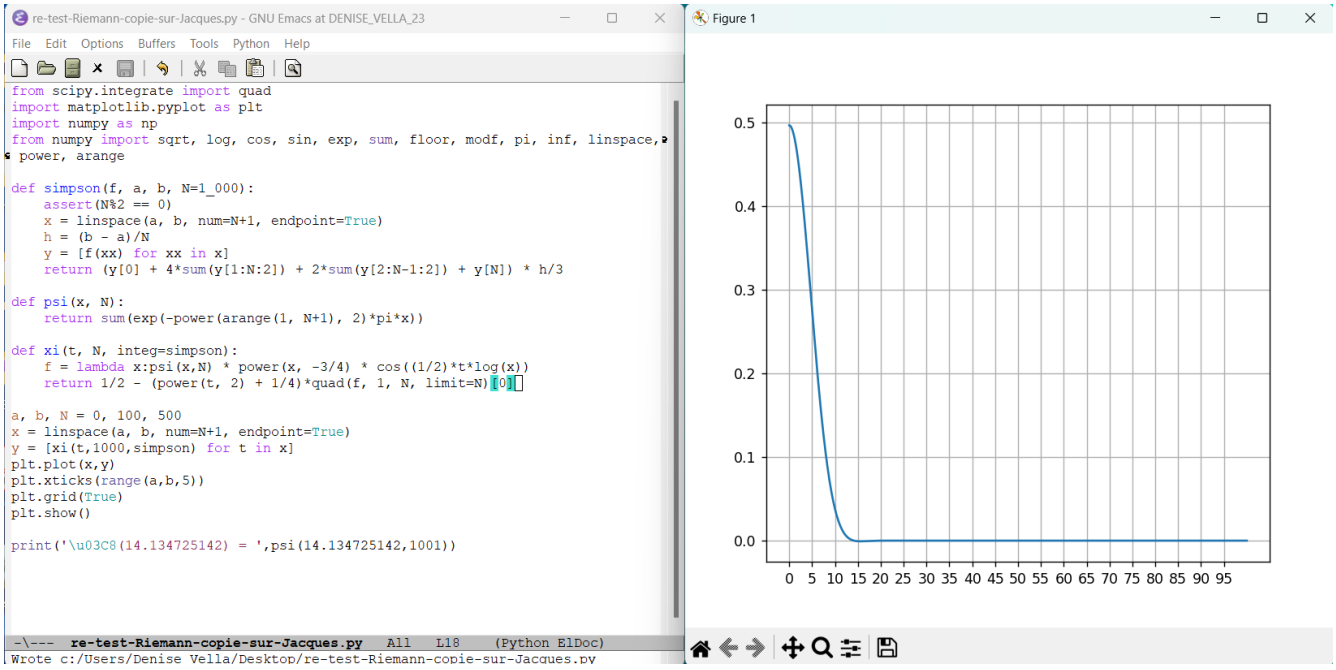
3) De la nécessité d'utiliser une fonction de calcul d'intégrale fournie par python

Ci-dessous le résultat de deux programmes très proches : on utilise dans le premier cas l'intégrale calculée par méthode Simpson 1/3 et dans l'autre cas la même intégrale mais calculée par python. Les résultats d'exécution sont très très différents :

Calcul de l'intégrale par méthode Simpson 1/3 (pas très loin)



Calcul de l'intégrale par la fonction du package numpy de python numpy.quad



On rappelle ici que $\zeta(0) = -\frac{1}{2}$ et que $\ln \zeta(0) = -\ln 2 + \pi.i$ [1]

On reste avec le doute, par rapport à cette fonction ξ de l'article de Riemann car elle est notée comme une fonction travaillant sur les réels, puisqu'elle est définie en fournissant l'expression de

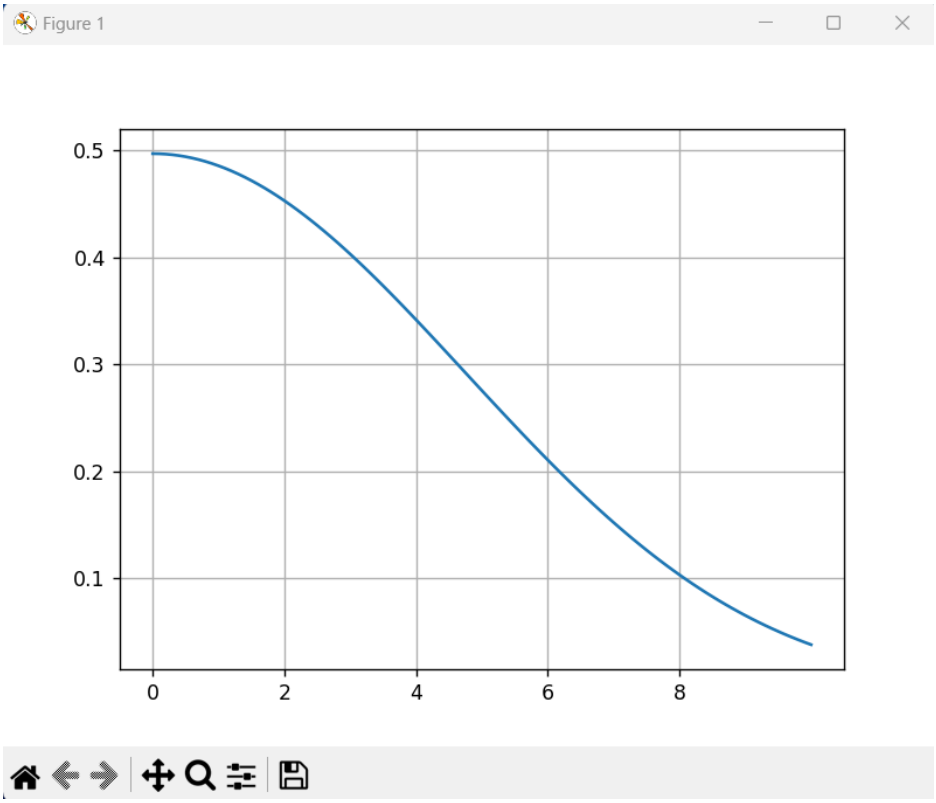
[1] Cela ne correspond pas aux valeurs fournies par le traducteur dans la note 8 à la suite de l'article de Riemann ici lien .

$\xi(t)$ avec un peu plus haut écrit $s = \frac{1}{2} + ti$ donc t réel mais par ailleurs, Riemann note :

> "*la fonction* $\xi(t)$ *peut seulement s'évanouir lorsque la partie imaginaire de* t *se trouve comprise entre* $\frac{1}{2}i$ *et* $-\frac{1}{2}i$. *Le nombre de racines de* $\xi(t) = 0$ *dont les parties réelles sont comprises entre 0 et* T *est environ égal à*
>
> $$\frac{T}{2\pi} \log \frac{T}{2\pi} - \frac{T}{2\pi}.$$"

ce qui exprime le fait que la fonction ξ est une fonction définie pour des nombres complexes.

On note que l'utilisation de *quad* du module `numpy` de python n'a cependant pas amélioré grand chose quand on regarde les résultats détaillés : pour des petits intervalles de 10 en 10 on ne parvient pas à voir les zéros correctement à nouveau à partir de 50.



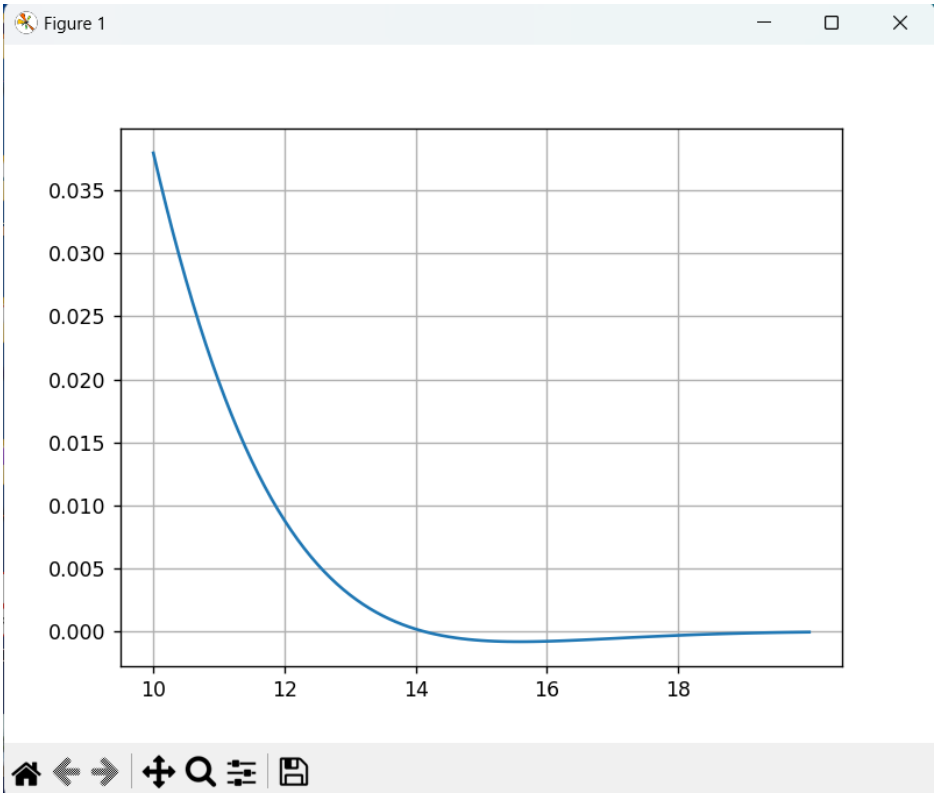


Figure 1
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
10
12
14
16
18

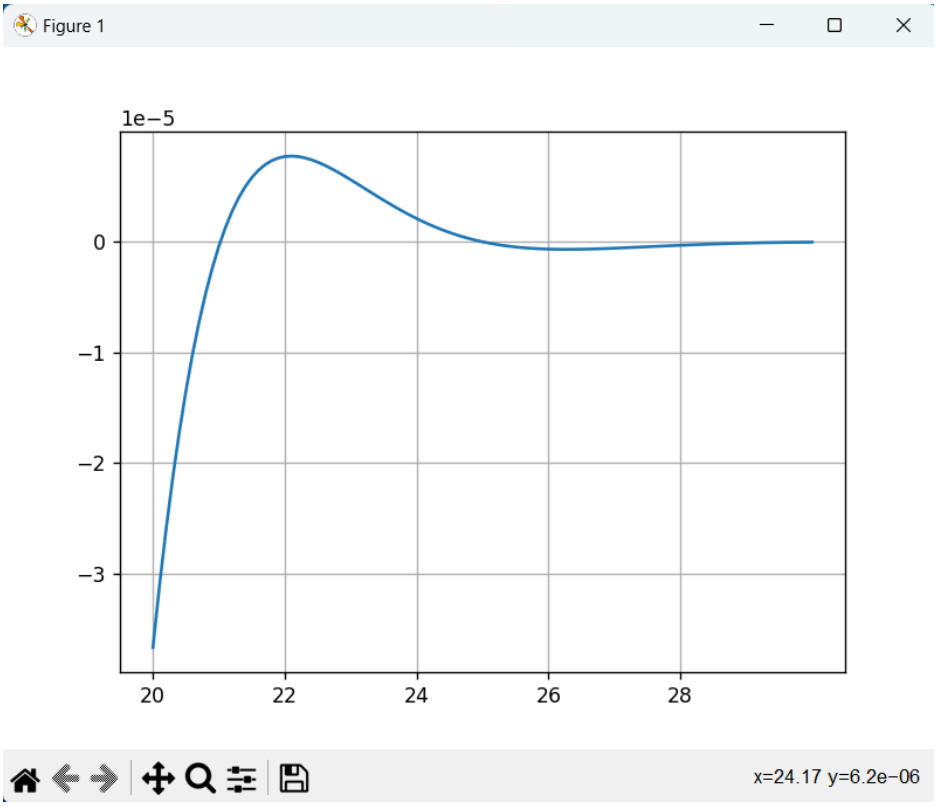


Figure 1
1e−5
0
−1
−2
−3
20
22
24
26
28
x=24.17 y=6.2e−06

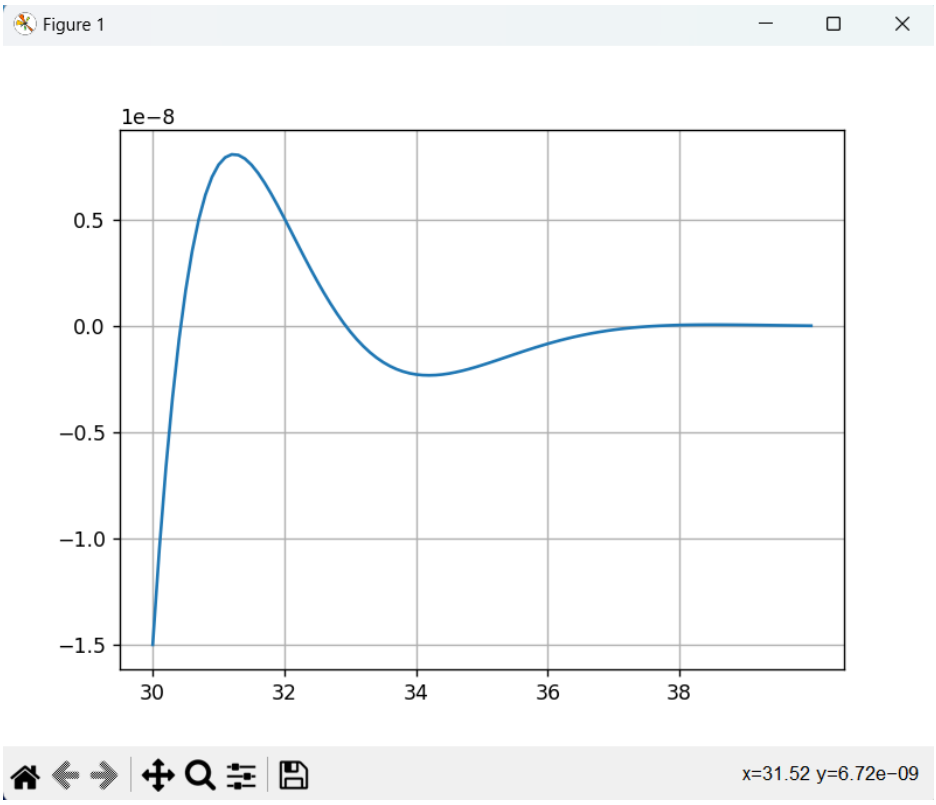


Figure 1
1e−8
0.5
0.0
−0.5
−1.0
−1.5
30
32
34
36
38
x=31.52 y=6.72e−09

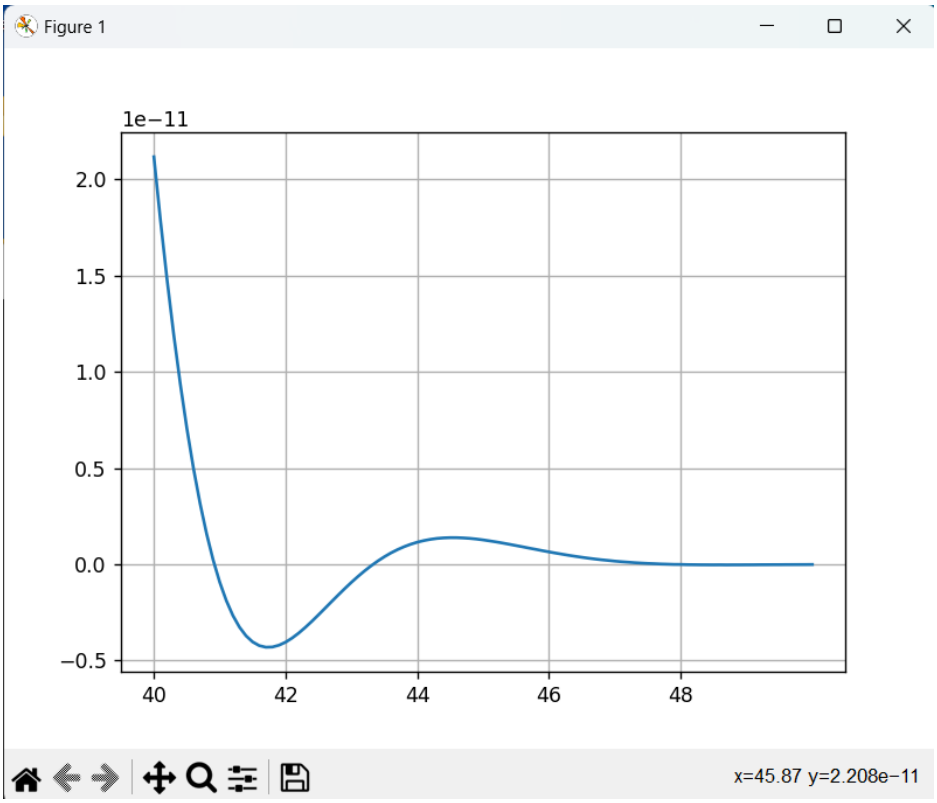


Figure 1
1e−11
2.0
1.5
1.0
0.5
0.0
−0.5
40
42
44
46
48
x=45.87 y=2.208e−11

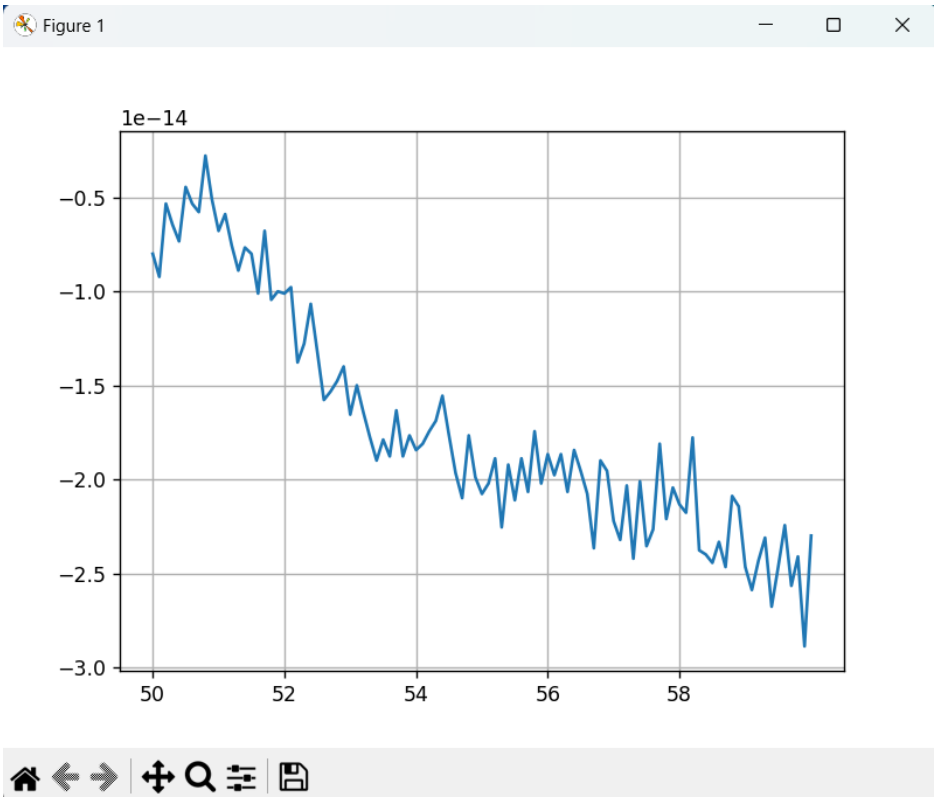


Figure 1
1e−14
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
50
52
54
56
58

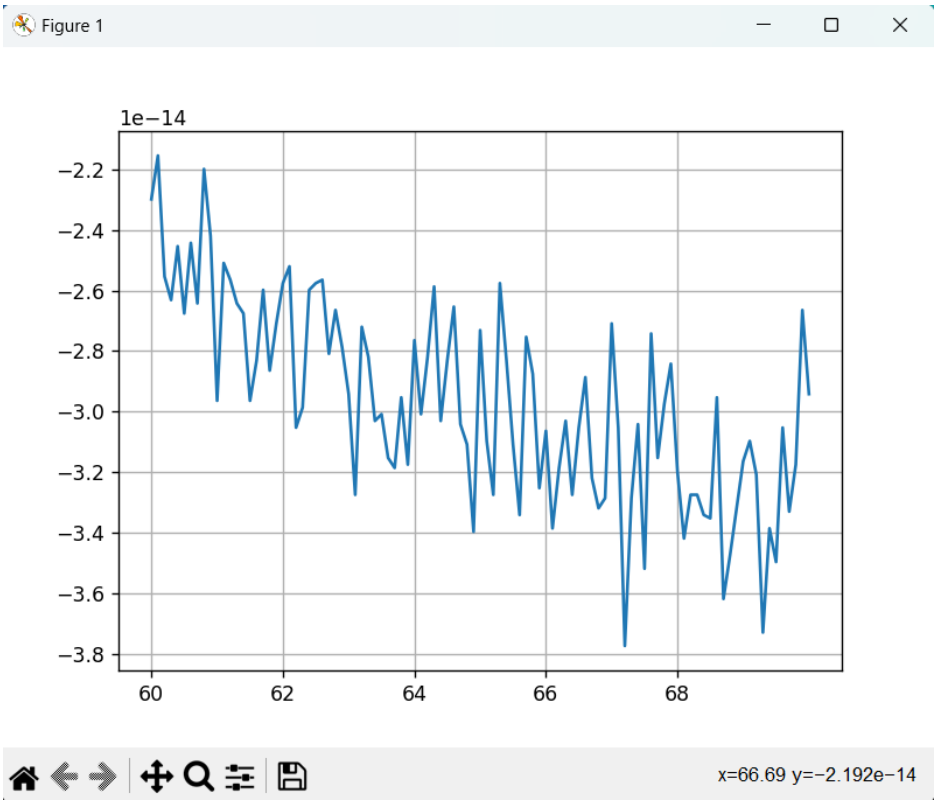


Figure 1
1e−14
−2.2
−2.4
−2.6
−2.8
−3.0
−3.2
−3.4
−3.6
−3.8
60
62
64
66
68
x=66.69 y=−2.192e−14

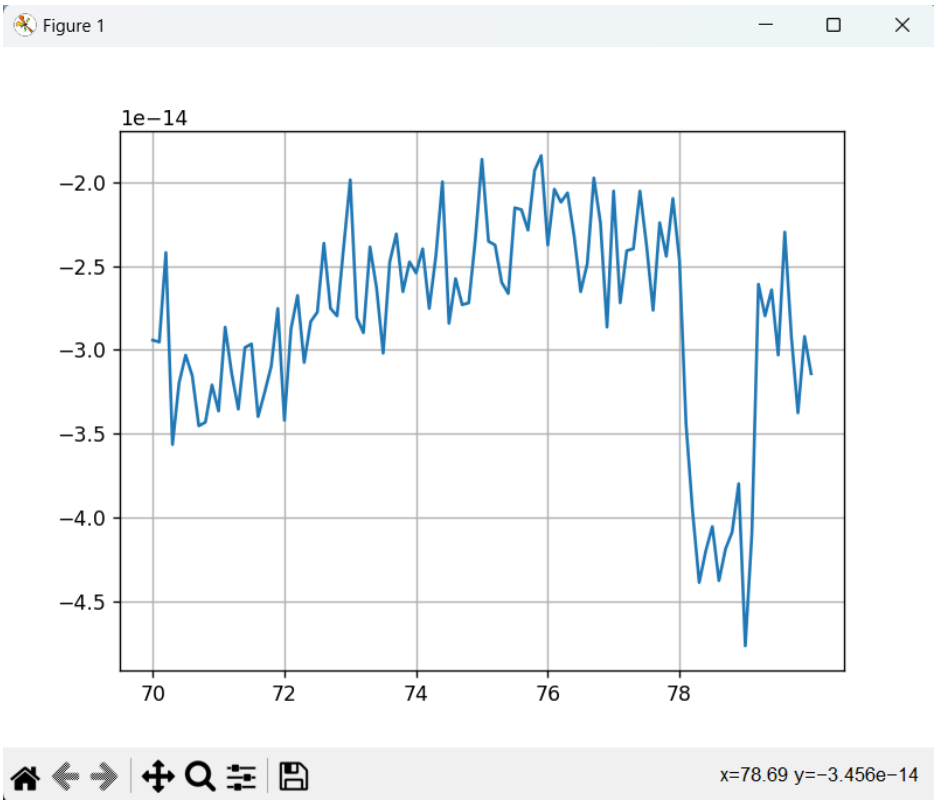


Figure 1
1e−14
−2.0
−2.5
−3.0
−3.5
−4.0
−4.5
70
72
74
76
78
x=78.69 y=−3.456e−14

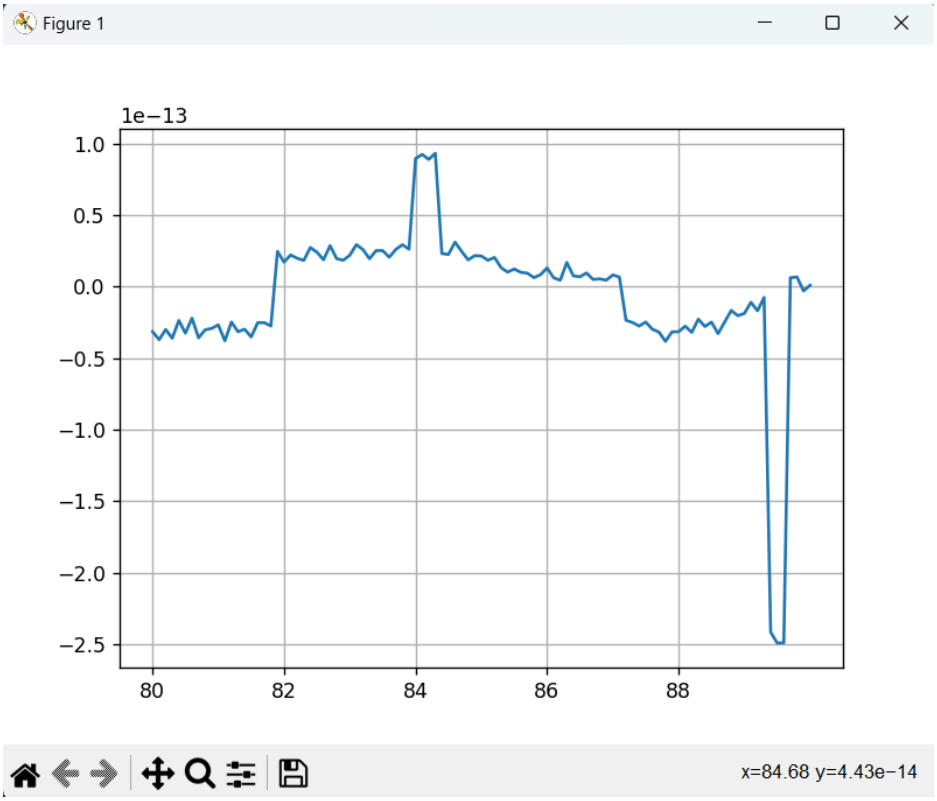
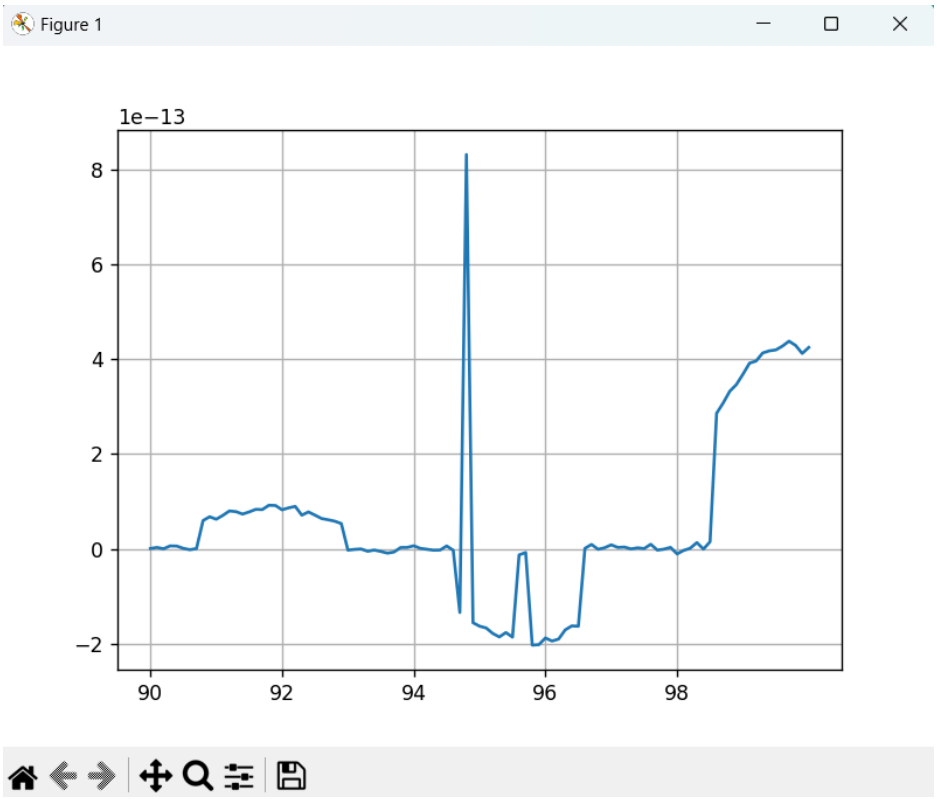
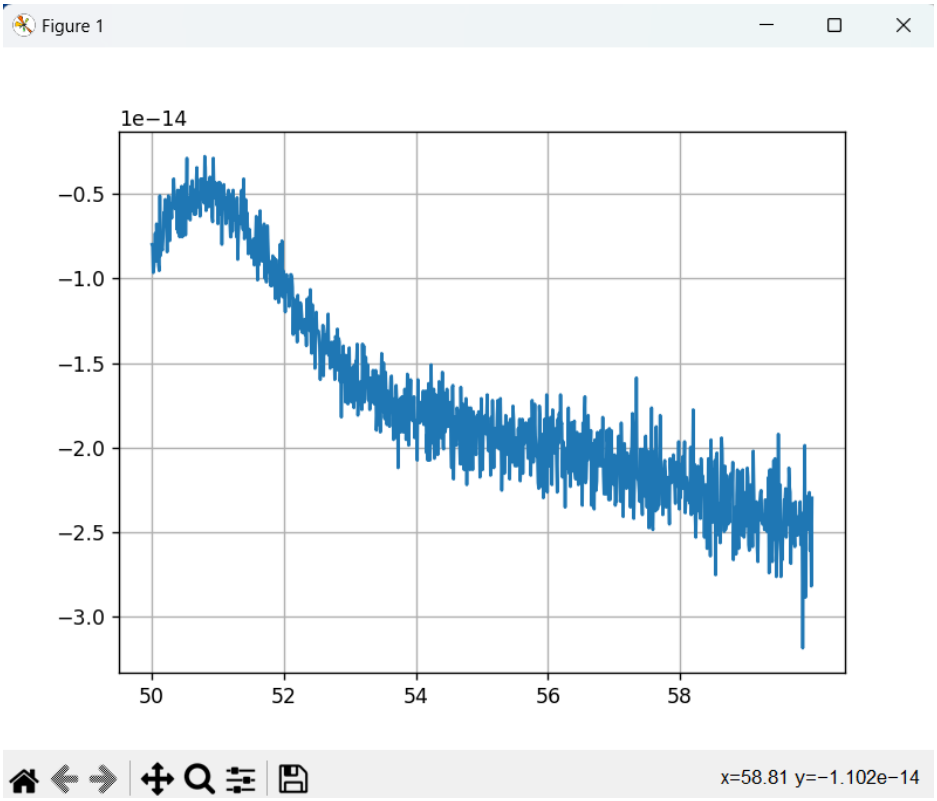


Figure 1
1e−13
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
80
82
84
86
88
x=84.68 y=4.43e−14



Et les résultats sont mauvais à partir de 50 même en augmentant le nombre de points d'un facteur 10 (au-delà, les temps d'exécution sont prohibitifs). Ci-dessous, de 50 à 60 avec 1000 points ou de 90 à 100 avec 1000 points.



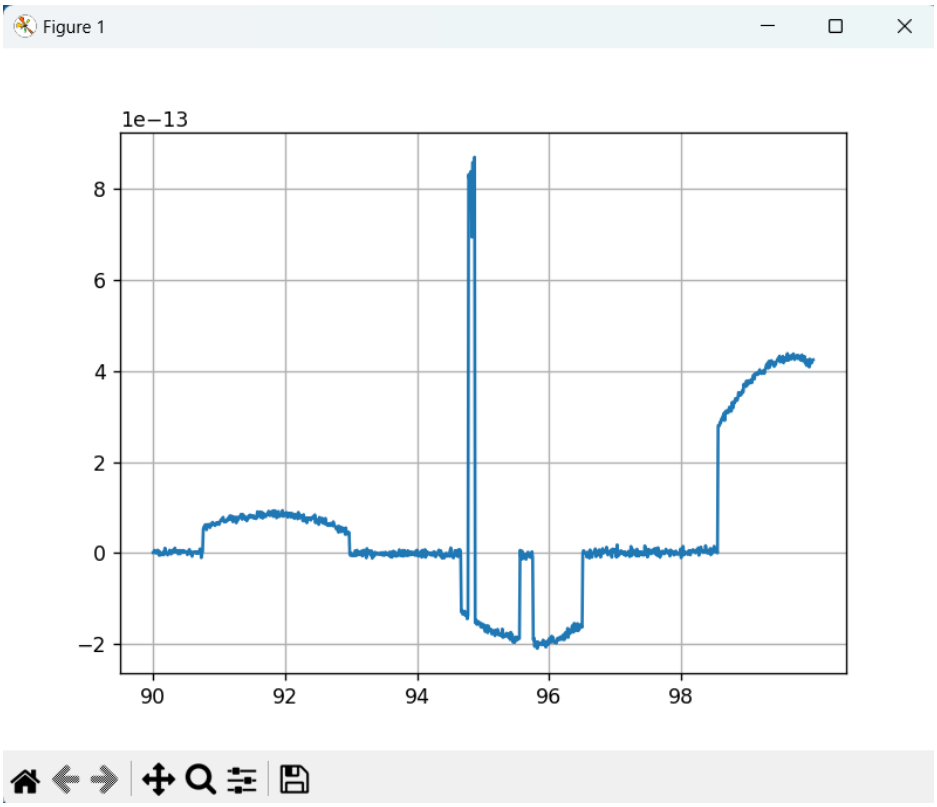


Figure 1
1e−13
8
6
4
2
0
−2
90
92
94
96
98