TP 18

Programmation en Python

Hugo Salou

MP2I

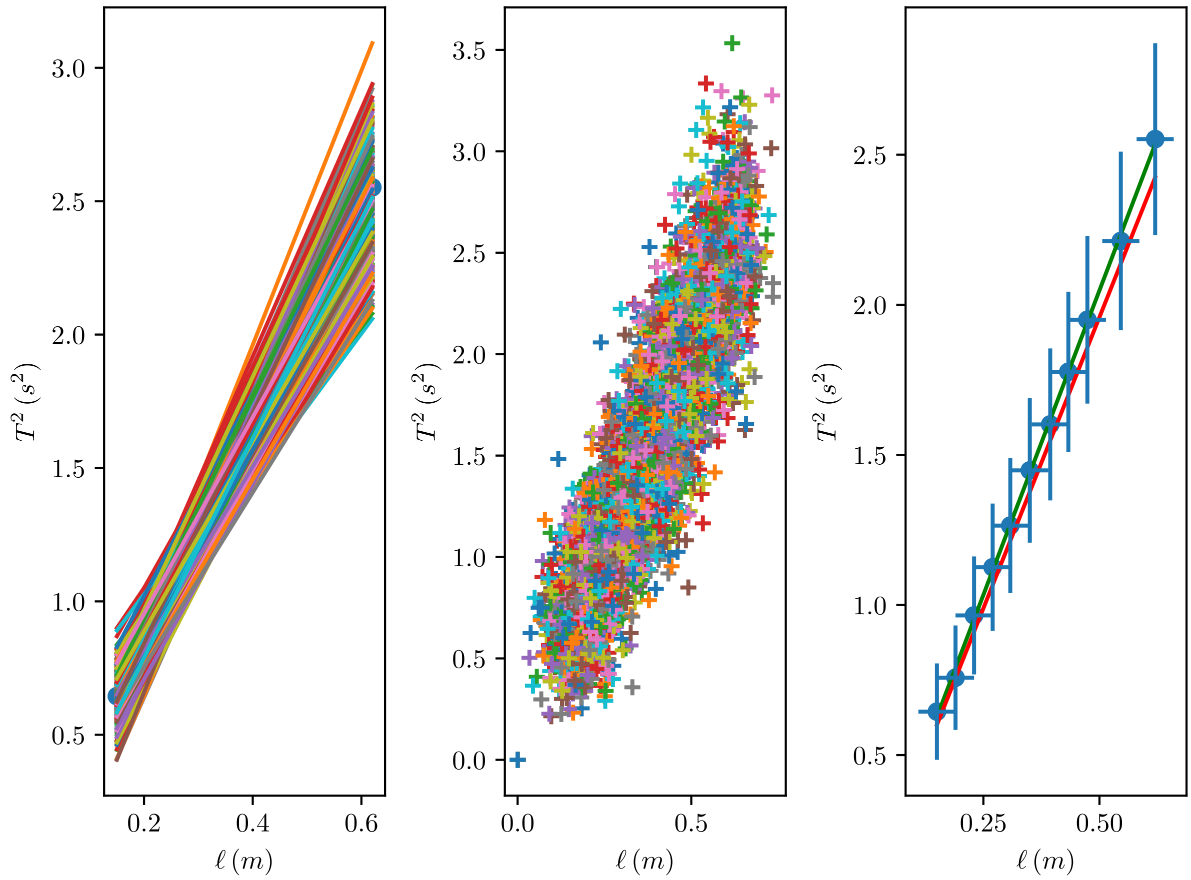
I Régression linéaire sur des données expérimentales

On a $T = 2\pi\sqrt{\frac{\ell}{g}}$.

```
import numpy as np
import matplotlib.pyplot as plt

T20=np.array([31.95, 29.75, 27.93, 26.66, 25.31, 24.07, 22.49,
                                21.22,19.65, 17.41, 16.06])

L0=np.array([0.620, 0.546, 0.474, 0.433, 0.394, 0.350, 0.308, 0.270,
                         0.23, 0.190, 0.15])

N = 400

plt.title("recherche de droite pour modeliser T=f(l)")
ax1 = plt.subplot(1, 3, 1) # droites
ax2 = plt.subplot(1, 3, 2) # points aléatoires
ax3 = plt.subplot(1, 3, 3) # courbe avec les points

card=len(T20)
uT=[0.10]*card
uL0=[0.04]*card

T2=[ (x / 20.) ** 2 for x in T20] # calcul de T^2
uT2=[ 2. * (t / 20.) * u_t for t, u_t in zip(T20, uT)] # calcul de u(T^2)

# Les droites sont de la forme y = ax + b

a_vals = [] # coefficients directeurs
b_vals = [] # ordonnées à l'origine

ax1.scatter(L0, T2)
ax2.scatter(L0, T2)
ax3.scatter(L0, T2)

for k in range(N):
        # le point (T = 0, ℓ = 0) est toujours présent
        courbe_t2 = [0]
        courbe_l = [0]

        for i in range(card):
                t2 = T2[i]
                u_t2 = uT2[i]
                l = L0[i]
                u_l = uL0[i]

                # on choisit un point aléatoire aux coordonées (t1^2, ℓ1)
                t2_1 = np.random.normal(t2, u_t2)
                l_1 = np.random.normal(l, u_l)

                courbe_t2.append(t2_1)
                courbe_l.append(l_1)

        # régression linéaire
        a, b = [ float(x) for x in np.polyfit(courbe_l, courbe_t2, 1) ]
        ax2.scatter(courbe_l, courbe_t2, marker="+") # on affiche les points sur graph2
        ax1.plot(L0, a * L0 + b) # et la courbe sur graph1

        a_vals.append(a)
```

```python
        b_vals.append(b)

# régression linéaire sans prendre en compte les incertitudes
p = np.polyfit(L0,T2,1)
a=float(p[0])
b=float(p[1])

ax3.plot(L0, a*L0+b, color="green")

# calcul des valeurs moyennes et des incertitudes
ma = np.mean(a_vals)
mb = np.mean(b_vals)

ua = np.std(a_vals, ddof=1)
ub = np.std(b_vals, ddof=1)

ax3.plot(L0, ma*L0+b, color="red")

print(f"a = {ma:0.4} ± {ua:0.4}")
print(f"b = {mb:0.4} ± {ub:0.4}")

# calcul de la valeur de g et son incertitude

g = 1 / (a / (np.pi**2 * 4))
ug = (2*np.pi/a)**2 * ua
print(f"g = {g:0.4} ± {ug:0.4}")

ax3.errorbar(L0, T2, uT2, uL0, fmt='none')

ax1.set_xlabel("l (m)")
ax1.set_ylabel("T^2 (s^2)")
ax2.set_xlabel("l (m)")
ax2.set_ylabel("T^2 (s^2)")
ax3.set_xlabel("l (m)")
ax3.set_ylabel("T^2 (s^2)")

plt.show()
```

On obtient $\boxed{g = 9.728 \pm 1.079}$.

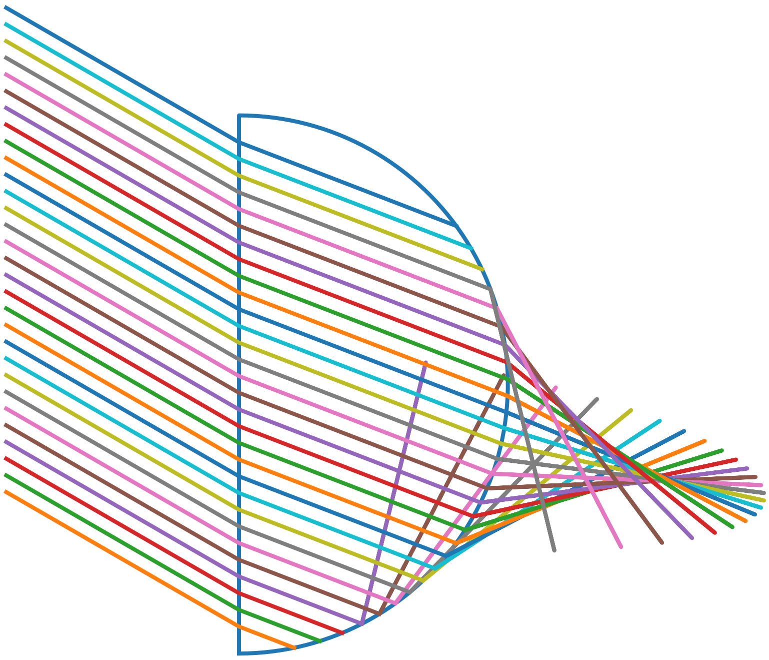


II Conditions de Gauss

```
import numpy as np
import matplotlib.pyplot as plt

plt.gca().set_aspect('equal')

# ici, alpha est l'angle d'incidence des rayons
alpha = np.pi / 6
n1, n2 = 1, 1.458 # indices de réfraction (air et verre)
r = 1 # rayon de la lentille

beta = -np.arcsin((n1/n2)*alpha)

# on affiche la lentille
ctheta = np.linspace(-np.pi/2, np.pi/2, 100)
cx = r * np.cos(ctheta)
cy = r * np.sin(ctheta)

plt.plot(np.append(cx, 0), np.append(cy, -r))

# coordonée y des rayons
rayons = np.linspace(-r+0.1, r-0.1, 30)

for y in rayons:
        # rayon avant d'entrer dans la lentille
        r1x = r * np.linspace(-1, 0, 10) * np.cos(alpha)
        r1y = -r * np.linspace(-1, 0, 10) * np.sin(alpha) + y

        # rayon dans la lentille
```

```
        px, py = 0, y
        fact = 1/2.

        # on cherche à estimer la position du point de sortie
        # de la lentille avec une méthode type dichotomie
        # la solution est plus simple dans le cas où α = 0
        # ( x = √(r²  - y²) ) mais, je n'ai pas trouvé d'expression
        # exacte pour x avec α quelconque

        for i in range(20):
                dt = fact**i

                # si le point est en dehors de la lentille,
                # on se rapproche de la lentille, sinon, on
                # s'éloigne
                if px**2 + py**2 < r**2:
                        px += r * dt * np.cos(beta)
                        py += r * dt * np.sin(beta)
                else:
                        px -= r * dt * np.cos(beta)
                        py -= r * dt * np.sin(beta)

        # normale du de la lentille
        normal = np.array([px,py]) / r

        r2x = np.array([px])
        r2y = np.array([py])

        # ici, vec représente un vecteur unitaire
        # dans la même direction que le rayon
        vec = np.array([px,py-y])
        vec = vec / np.linalg.norm(vec)

        # φ est l'angle incident et ψ est l'angle réfracté
        cos_phi = min(np.dot(normal, vec), 1)
        sin_phi = np.sqrt(1 - cos_phi**2)
        sin_psi = (n2/n1) * sin_phi

        if abs(sin_psi) > 1: # réfléction totale
                rx = np.concatenate((r1x, r2x))
                ry = np.concatenate((r1y, r2y))

                plt.plot(rx,ry)
                continue

        cos_psi = np.sqrt(1 - sin_psi**2)

        s = np.sign(-y) # correction pour le signe de sin(ψ)

        r3x = r * np.linspace(1, 0, 10) * cos_psi + px
        r3y = s * r * np.linspace(1, 0, 10) * sin_psi + py

        # on affiche les rayons
        rx = np.concatenate((r1x, r2x, r3x))
        ry = np.concatenate((r1y, r2y, r3y))

        plt.plot(rx,ry)

plt.axis('off')
plt.show()
```

Avec $\alpha = \frac{\pi}{6}$,



Il faut que les rayons soient proche du centre de la lentille pour respecter les conditions de Gauss. Sinon, il n'y a pas un foyer image où les rayons parallèles convergent.

III Système linéaire du premier ordre

L'équation différentielle est $\frac{\mathrm{d}u_C}{\mathrm{d}t} + \frac{1}{\tau}u_C(t) = \frac{1}{\tau}e(t)$ avec $\tau = RC$.
En posant $t' = \frac{t}{\tau}$, on obtient

$$\frac{1}{\tau}\frac{\mathrm{d}u_C}{\mathrm{d}t'} = \frac{e(t') - u_C(t')}{\tau}$$

On résout donc l'équation différentielle $\frac{\mathrm{d}u}{\mathrm{d}t'} = e(t') - u(t')$.
On doit avoir $\Delta t' \ll 1$ pour avoir $\Delta t \ll \tau$.
Dans le programme, la variable `t` représente t' au lieu de t.

```
import numpy as np
import matplotlib.pyplot as plt

T = 30 # periode du signal

def e(t): # signal d'entrée
    return np.floor((t % T)/T + 0.5)

dt = 0.01 # valeur de Δt
ts = np.arange(0., 100., dt) # valeurs de t

plt.plot(ts, e(ts)) # on affiche e
```

```
u = 0
us = []

for t in ts:
    # calcul de du avec l'équation différentielle
    du = (e(t) - u) * dt
    # on modifie u et on stocke le résultat
    u += du
    us.append(u)

plt.plot(ts, us)
plt.show()
```