Tour d'horizon des capacités numériques

Déroulement de la présentation

Un tour du programme de physique et de sciences industrielles

Usage général de Numpy/Scipy/Matplotlib

Traitement des signaux

Équations différentielles

Pivot de Gauss

Physique : tracé

Utiliser les fonctions de base de la bibliothèque matplotlib pour

- représenter un nuage de points
- tracer la courbe représentative d'une fonction
- tracer une courbe plane paramétrée
- représenter les résultats d'un ensemble de tirages d'une variable aléatoire avec la fonction `hist` (sa spécification étant fournie)

Physique : aléatoire

Utiliser les fonctions de base des bibliothèques `random` et/ou `numpy` (leurs spécifications étant fournies) pour réaliser des tirages d'une variable aléatoire

Utiliser la fonction `random.normal` de la bibliothèque numpy (sa spécification étant fournie) pour simuler un processus aléatoire

Déterminer la moyenne et l'écart-type d'un ensemble de tirages d'une variable aléatoire

Simuler un processus aléatoire permettant de caractériser la variabilité de la valeur d'une grandeur composée

Simuler un processus aléatoire de variation des valeurs expérimentales de l'une des grandeurs – simulation Monte-Carlo – pour évaluer l'incertitude sur les paramètres du modèle

Physique : modélisation

Utiliser la fonction polyfit de la bibliothèque numpy (sa spécification étant fournie) pour exploiter des données

Physique : dérivation, intégration

Mettre en œuvre la méthode des rectangles pour calculer une valeur approchée d'une intégrale sur un segment

Utiliser un schéma numérique pour déterminer une valeur approchée du nombre dérivé d'une fonction en un point

Physique : analyse spectrale

Calculer la transformée de Fourier discrète d'un signal numérique

Calculer la transformée de Fourier discrète d'un signal à valeurs réelles en utilisant la fonction `rfft` de la bibliothèque `numpy.fft` (sa spécification étant donnée)

Simuler l'action d'un filtre sur un signal périodique dont le spectre est fourni

Réaliser un filtrage numérique d'un signal issu d'une acquisition, et mettre en évidence la limitation introduite par l'échantillonnage

Mettre en évidence l'influence des caractéristiques du filtre sur l'opération de filtrage

Physique : recherche de racines

Déterminer un intervalle adapté à la recherche numérique d'une racine par une méthode dichotomique

Mettre en œuvre une méthode dichotomique afin de résoudre une équation avec une précision donnée

Utiliser la fonction bisect de la bibliothèque `scipy.optimize` (sa spécification étant fournie)

Déterminer le(s) point(s) de fonctionnement (température et taux de conversion) d'un réacteur ouvert siège d'une transformation modélisée par une réaction isotherme unique et en discuter la stabilité

Physique : équations différentielles

Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1

Mettre en œuvre la méthode d'Euler pour simuler la réponse d'un système linéaire du premier ordre à une excitation de forme quelconque

Transformer une équation différentielle d'ordre n en un système différentiel de n équations d'ordre 1

Résoudre numériquement une équation différentielle du deuxième ordre non-linéaire et faire apparaître l'effet des termes non-linéaires

Utiliser la fonction `odeint` de la bibliothèque `scipy.integrate` (sa spécification étant fournie)

Physique : équations différentielles (exemples)

Obtenir des trajectoires d'un point matériel soumis à un champ de force centrale conservatif

Mettre en évidence le non isochronisme des oscillations

Étudier les variations de température et de pression dans l'atmosphère

Simuler l'évolution temporelle d'un signal généré par un oscillateur

Résoudre l'équation de la diffusion thermique à une dimension par une méthode des différences finies dérivée de la méthode d'Euler explicite de résolution des équations différentielles ordinaires

Mettre en œuvre une méthode des différences finies explicite pour résoudre l'équation de diffusion à une dimension en régime variable

Physique : autres

Simuler la propagation d'un paquet d'ondes dans un milieu dispersif et visualiser le phénomène d'étalement

(Pas de compétences à maîtriser)

Sciences industrielles

La frontière est un peu plus floue

Le niveau de maîtrise attendue aussi

Suit une partie des items au programme

SII : intelligence artificielle

Régression et classification

Apprentissages supervisé et non supervisé

Phases d'apprentissage et d'inférence

Modèle linéaire monovariable ou multivariable

Réseaux de neurones (couches d'entrée, cachées et de sortie, neurones, biais, poids et fonction d'activation)

SII : intelligence artificielle

Matrice de confusion (tableau de contingence)

Sensibilité et spécificité d'un test

Choix des algorithmes

- réseaux de neurones
- k plus proches voisins
- régression linéaire multiple

SII : filtrage numérique

Diagrammes de Bode

Signaux à temps discret (échantillonnage et quantification)

Modélisation par équations aux différences d'un correcteur numérique

Allure des réponses temporelles (impulsion, échelon, rampe, sinus)

SII : résolution numérique

Résolution de problèmes du type $f(x) = 0$

(méthodes de dichotomie et de Newton)

Résolution d'un système linéaire du type $\mathrm{AX} = \mathrm{B}$

Résolution d'équations différentielles

(schéma d'Euler explicite)

Intégration et dérivation numérique

(schémas arrière et avant)

SII : traitement des données

Traitement de fichiers de données

Moyenne et écart-type

Moyenne glissante

Filtres numériques passe-bas du premier et du second ordre

Déroulement de la présentation

Quelques modules utiles

Les capacités utilisent différents modules :

- `numpy` (`np`) : vecteurs, matrices, calcul numérique
- `numpy.fft` : transformée de Fourier
- `scipy.integrate` (`itg`) : intégrales, équations différentielles
- `scipy.optimize` (`opt`) : résolutions d'équations, de systèmes d'équations, minimisations, régressions, etc.
- `scipy.misc` (`msc`) : dérivations
- `matplotlib.pyplot` (`plt`) : graphes

Importer une fonction

```python
import numpy
numpy.foo(...)
```

Importer une fonction

```python
import numpy
numpy.foo(...)
```

```python
import numpy as np
np.foo(...)
```

Importer une fonction

```python
import numpy
numpy.foo(...)
```

```python
import numpy as np
np.foo(...)
```

```python
from numpy import foo
foo(...)
```

Calculs vectoriels

Le module numpy fournit un type « tableau » (ndarray)

Utilisé pour les vecteurs/matrices

```
In []: X = np.array([1, 2, 3])

In []: M = np.array([[0, 1, 0], [2, 1, 1]])

In []: M
Out[]:
array([[0, 1, 0],
       [2, 1, 1]])
```

Indexation

Indexation comme les listes / listes de listes :

```
In []: X[1]
Out[]: 2

In []: M[1]
Out[]: array([2, 1, 1])

In []: M[1][0]
Out[]: 2
```

Mais on peut écrire aussi :

```
In []: M[1,0]
Out[]: 2
```

Calculs vectoriels

Intérêt : on peut « vectoriser » les calculs :

```
In []: X = np.array([1, 2, 3])

In []: Y = np.array([4, 5, 6])

In []: 2*X
Out[]: array([2, 4, 6])

In []: X+Y
Out[]: array([5, 7, 9])
```

Calculs vectoriels

Attention, * est un produit terme à terme :

```
In []: X = np.array([1, 2, 3])

In []: Y = np.array([4, 5, 6])

In []: X*Y
Out[]: array([ 4, 10, 18])

In []: sum(X*Y)   # Produit scalaire
Out[]: 32
```

Calculs vectoriels

Le produit matriciel s'écrit, au choix :

- `np.dot(A, B)`
- `A.dot(B)`
- `A @ B`

```
In []: M @ X
Out[]: array([2, 7])
```

numpy « devine » si un vecteur 1D est ligne ou colonne

Calculs vectoriels

La vectorisation fonctionne aussi avec les fonctions :

```
In [ ]: np.sin(X)
Out[ ]: array([0.84147098, 0.90929743, 0.14112001])

In [ ]: def foo(x):
  ...:     return 2*x+3

In [ ]: foo(X)
Out[ ]: array([5, 7, 9])

In [ ]: foo(M)
Out[ ]:
array([[3, 5, 3],
       [7, 5, 5]])
```

Créer des tableaux

```
In []: np.zeros(5)
Out[]: array([0., 0., 0., 0., 0.])

In []: np.ones(5)
Out[]: array([1., 1., 1., 1., 1.])

In []: np.ones((2,3))
Out[]:
array([[1., 1., 1.],
       [1., 1., 1.]])

In []: 42 * np.ones(5)
Out[]: array([42., 42., 42., 42., 42.])
```

Créer des tableaux

```
In []: np.eye(3)
Out[]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In []: np.ones((3,3)) - np.eye(3)
Out[]:
array([[0., 1., 1.],
       [1., 0., 1.],
       [1., 1., 0.]])
```

Créer des tableaux

5 valeurs réparties entre 0 et 10 :

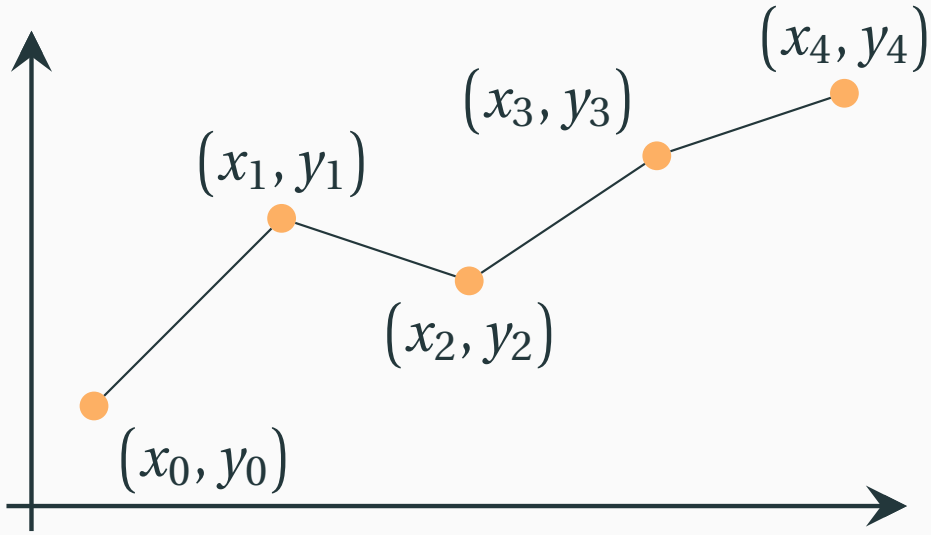
```
In []: np.linspace(0, 10, 5)
Out[]: array([ 0., 2.5, 5., 7.5, 10. ])
```

Valeurs de 0 à 10 (exclu) par pas de 2.5 :

```
In []: np.arange(0, 10, 2.5)
Out[]: array([ 0., 2.5, 5., 7.5 ])
```

Tracé d'une fonction

Une courbe est un ensemble de points (x_i, y_i)



Tracer une courbe se fait en trois étapes :

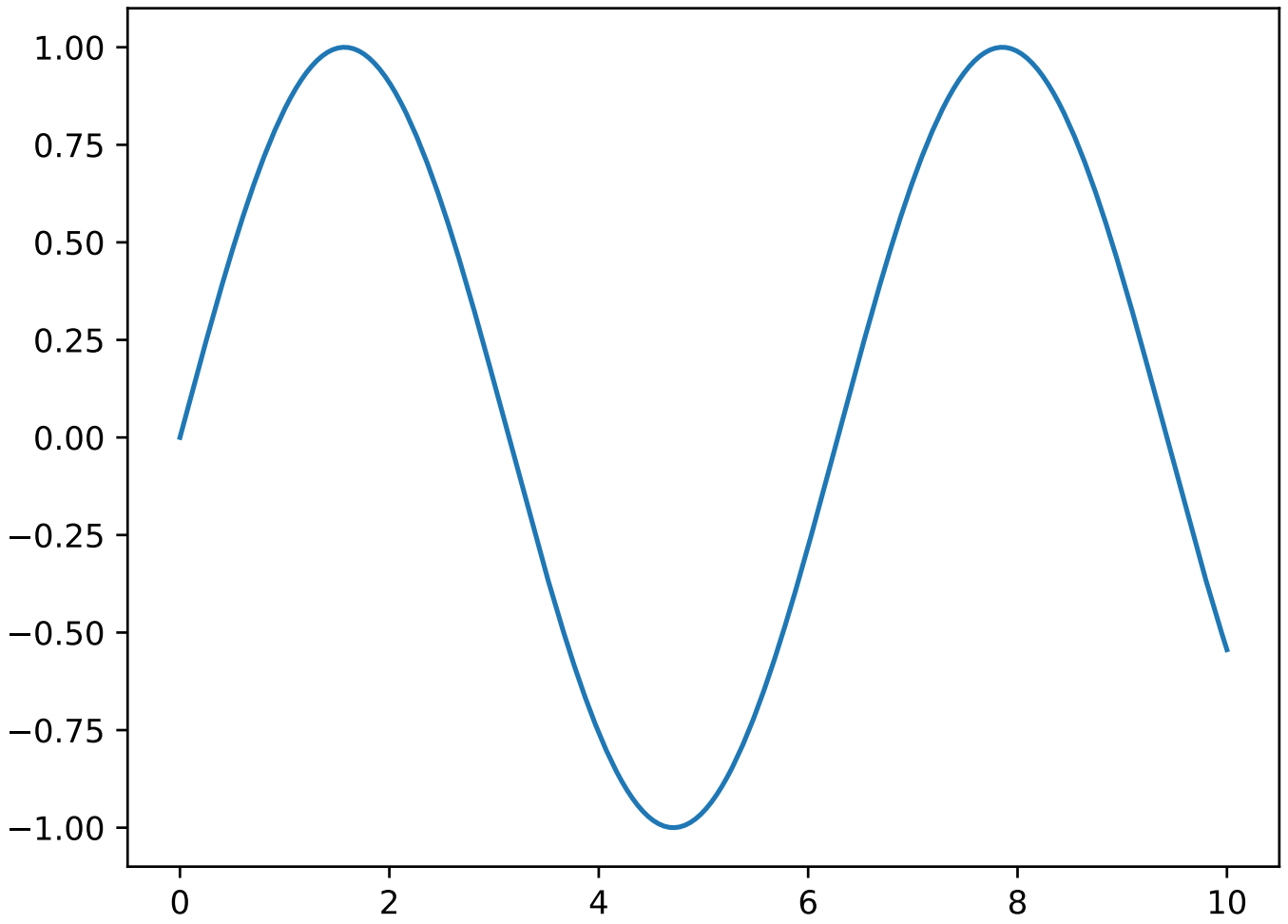
- on construit l'ensemble `X` des x_i
- on construit l'ensemble `Y` des y_i
- on appelle la fonction plot avec `X` et `Y` comme arguments

Tracé d'une fonction

Pour tracer $\sin(x)$ pour $x \in [0, 10]$:

```python
import numpy as np                    # <- imports des
import matplotlib.pyplot as plt       #    modules utiles

X = np.linspace(0, 10, 200)           # <- construction
Y = np.sin(X)                         #    des x_i et y_i

plt.plot(X, Y)                        # <- tracé
```

Tracé d'une fonction

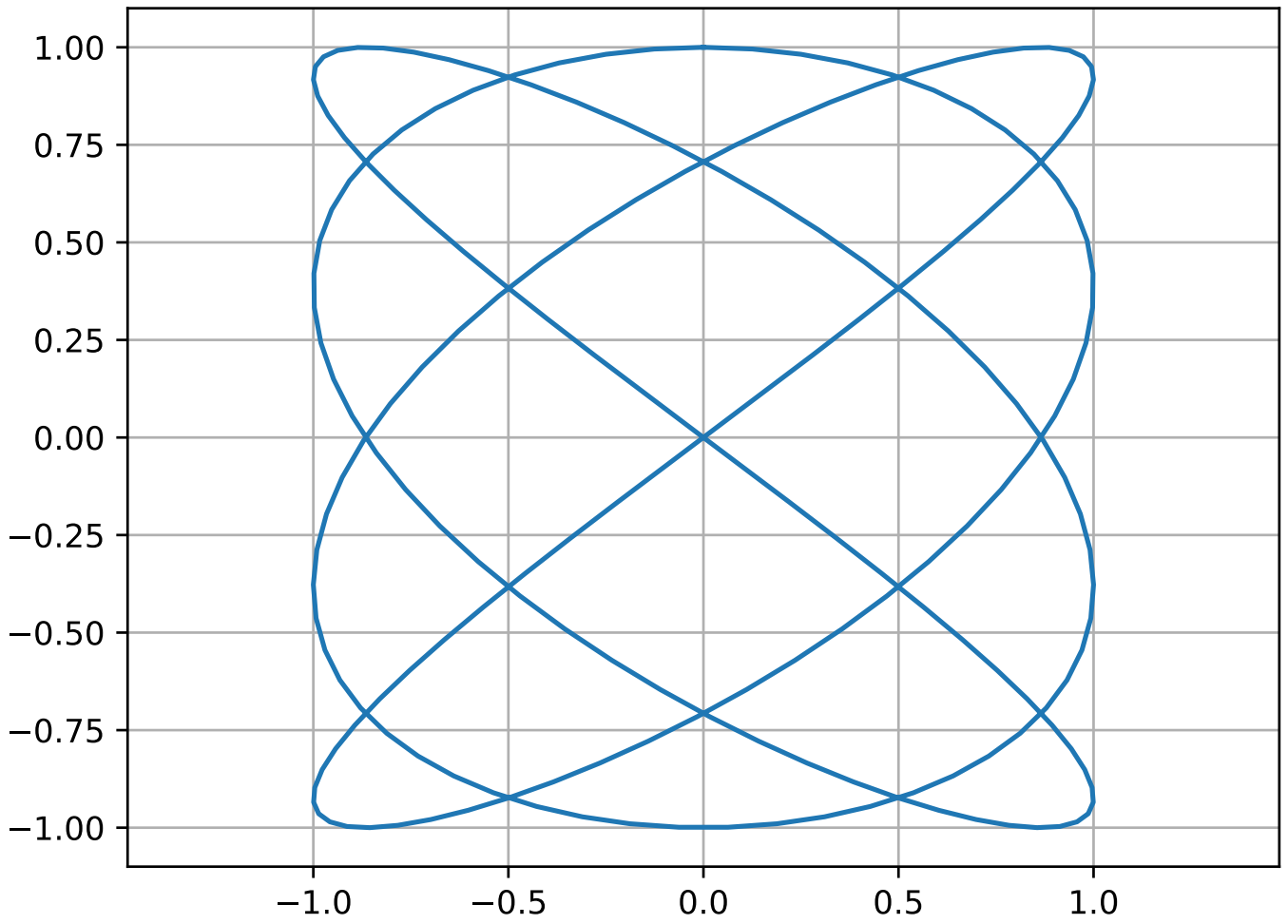


Tracé d'une fonction paramétrique

Tracer une fonction paramétrique est similaire :

```
import numpy as np                    # <- imports des
import matplotlib.pyplot as plt       #    modules utiles

T = np.linspace(0, 2*np.pi, 200) # <- construction
X = np.sin(4*T)                  #    des x_i et y_i
Y = np.cos(3*T)                  #

plt.plot(X, Y)                   # <- tracé

plt.grid()
plt.axis("equal")
```

Tracé d'une fonction paramétrique



Décoration du tracé

De nombreuses fonctions pour affiner le graphe :

- `plt.grid()` : afficher une grille
- `plt.title(ch)` : titre
- `plt.xlabel(ch)` : annotation abscisses
- `plt.ylabel(ch)` : annotation ordonnées
- `plt.axis("equal")` : repère orthonormé
- ...

Omis dans la suite

Génération de mesures

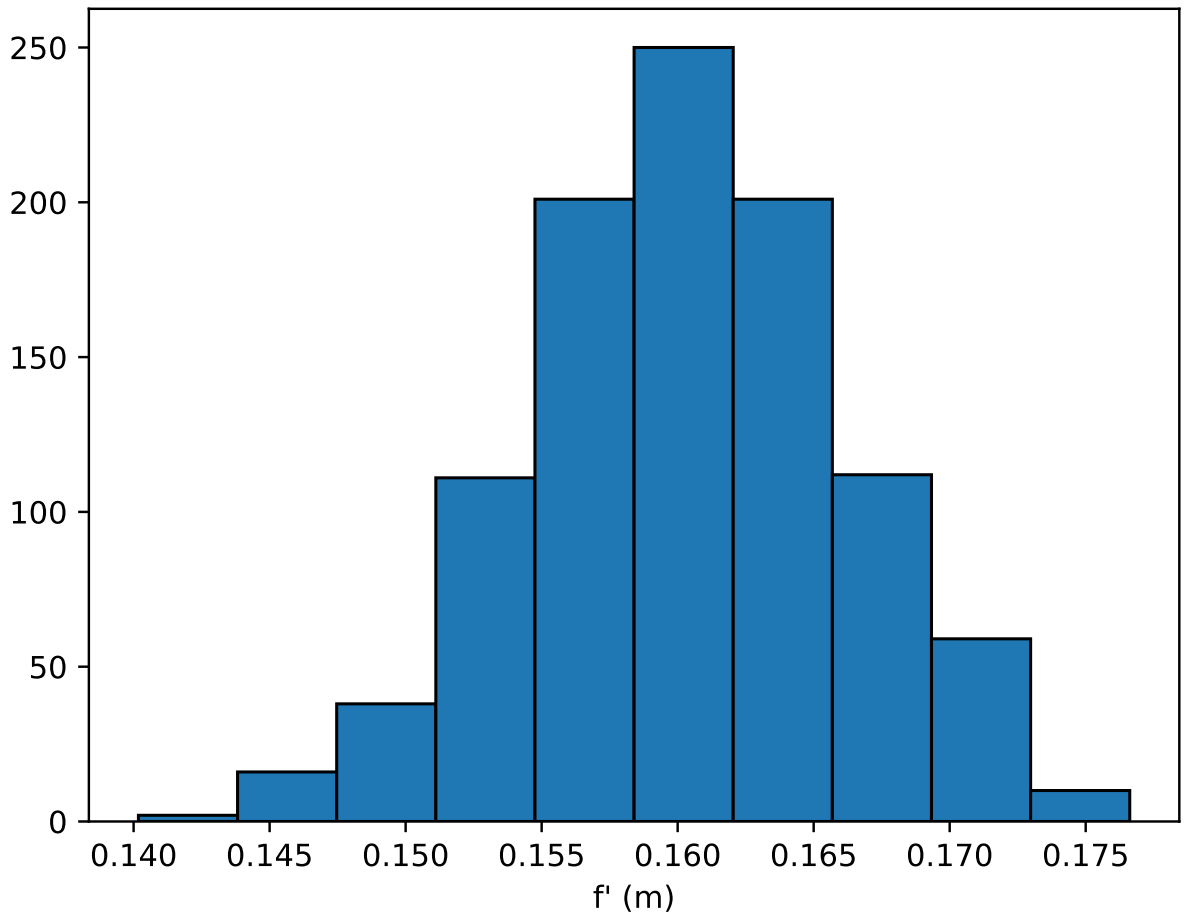
$$\frac{1}{f'} = \frac{1}{p'} - \frac{1}{p}$$

Simulation de 1000 mesures de $p = -45\,\text{cm}$ et $p' = 25\,\text{cm}$ à 5%

```python
N = 1000

P = [np.random.normal(-0.45, 0.0225) for _ in range(N)]
Pp = [np.random.normal(0.25, 0.0125) for _ in range(N)]

F = []
for i in range(len(P)):
    F.append(1.0 / (1.0/Pp[i] - 1.0/P[i]))
```

Histogramme

```
plt.hist(F, edgecolor='black')
```



Moyenne, écart-type

$$\mu = \frac{1}{N}\sum_i f_i \qquad \sigma \simeq \sqrt{\frac{1}{N}\sum_i f_i^2 - \mu^2}$$

```python
s = sum(f for f in F)
s2 = sum(f**2 for f in F)

m = s/len(F)
sig = (s2/len(F) - m**2.0)**0.5
```

Donne ici $\mu = 0{,}16\,\mathrm{m}$ et $\sigma = 5{,}8 \times 10^{-3}\,\mathrm{m}$

Moyenne, écart-type

Alternativement :

```
In []: np.mean(F)
Out[]: 0.16043431117292362

In []: np.var(F)**0.5
Out[]: 0.005841781613182127
```

Conforme à la propagation de l'erreur :

$$\Delta f = \sqrt{\left(\frac{\partial f}{\partial p}\right)^2 (\Delta p)^2 + \left(\frac{\partial f}{\partial p'}\right)^2 (\Delta p')^2} \simeq 5{,}9 \times 10^{-3}\ \mathrm{m}$$

Déroulement de la présentation

Signaux temporels

Les signaux $y(t)$ après acquisition sont de la forme

$$[\, y(t_0),\, y(t_1),\, y(t_2),\, \ldots,\, y(t_{\mathrm{N}-1})\,]$$

usuellement avec un pas de temps $\mathrm{d}t = t_{i+1} - t_i$ constant

Rangés typiquement dans un `np.ndarray`

Lecture de fichier

Une donnée (flottante) par ligne :

```
f = open("chemin/vers/fichier.dat")

signal = []

for ligne in f:
    signal.append(float(ligne))
```

Lecture de fichier

Deux données (flottantes) par ligne :

```
f = open("chemin/vers/fichier.dat")

T, signal = [], []

for ligne in f:
    ti, yi = ligne.split(";")
    T.append(float(ti))
    signal.append(float(yi))
```

Exemple d'enregistrement

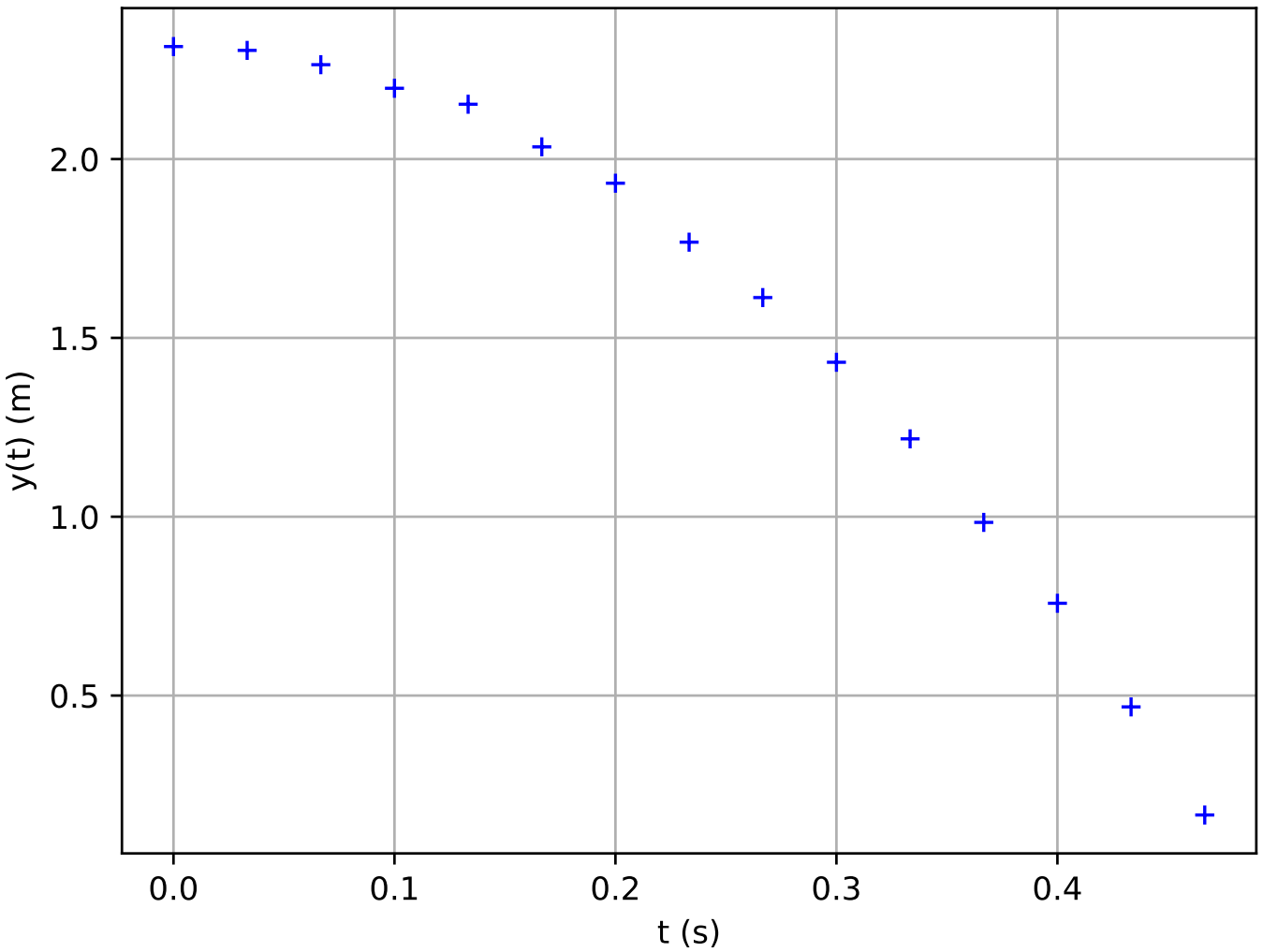
Chute d'une masse m sans vitesse initiale

Acquisition vidéo à 30 images par seconde

```python
dt = 1.0/30.0  # 30 fps
T = [i*dt for i in range()]
```

`Y` lu dans un fichier (pointage)

Exemple d'enregistrement



Ajustement

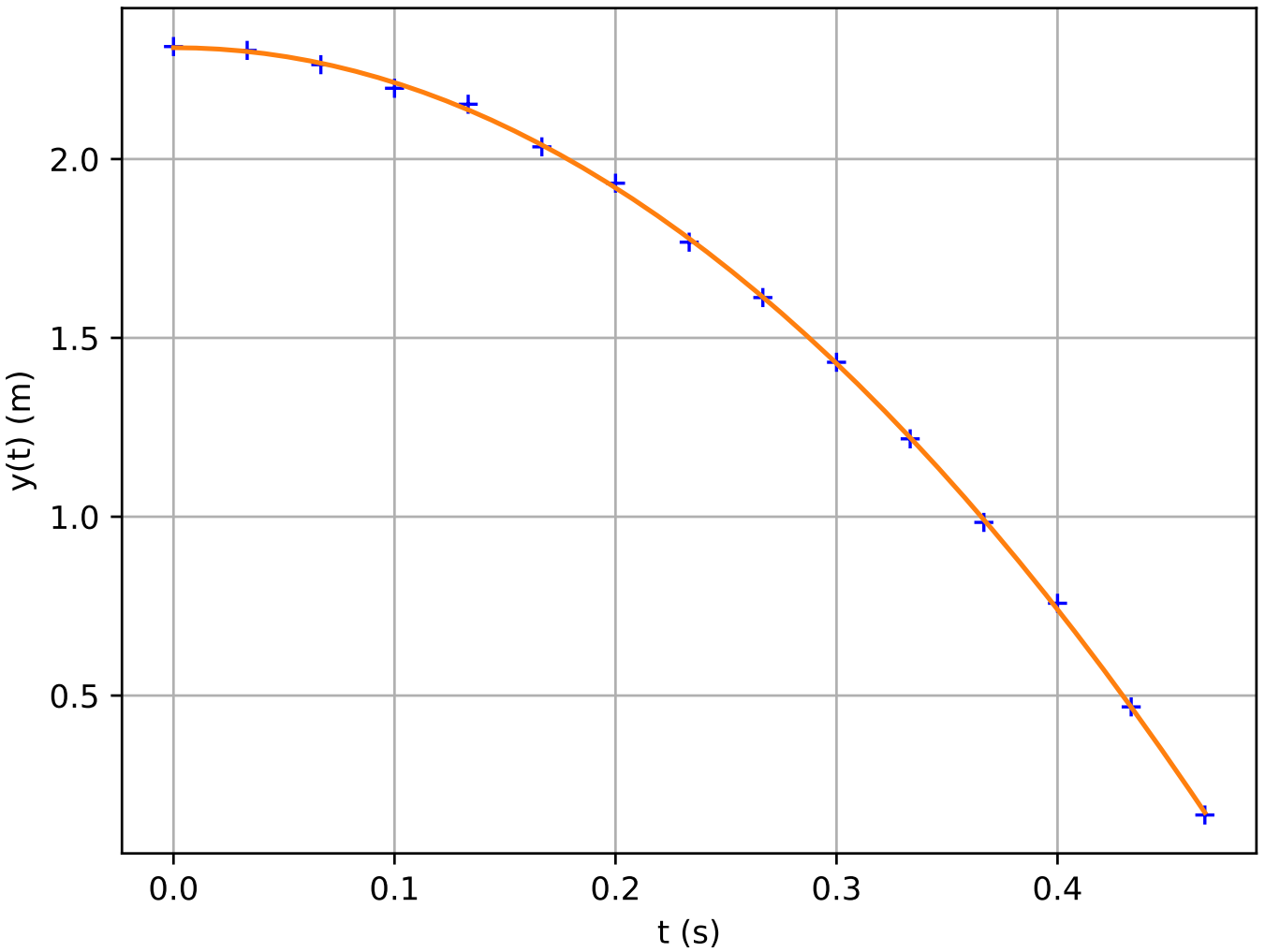
Ajustement avec $\alpha t^2 + \beta t + \gamma$:

```
In []: a, b, c = np.polyfit(T, Y, deg=2)


In []: a, b, c
Out[]: (-9.83919005e+00,  9.37817567e-03,
                                        2.31133888e+00)
```

Tracé du la parabole

```
Tf = np.linspace(T[0], T[-1], 500)
Yf = [a*t**2.0 + b*t + c for t in Tf]
plt.plot(Tf, Yf, "tab:orange")
```

Ajustement



Remarques sur l'ajustement

Des fonctions comme `polyfit` supposent :

- les incertitudes en x négligeables devant celles en y
- les incertitudes en y (grossièrement égales) (sinon, on peut indiquer les Δy_i de *chaque* mesure)

L'incertitude sur les paramètres obtenus est complexe à définir

Exemple de signal

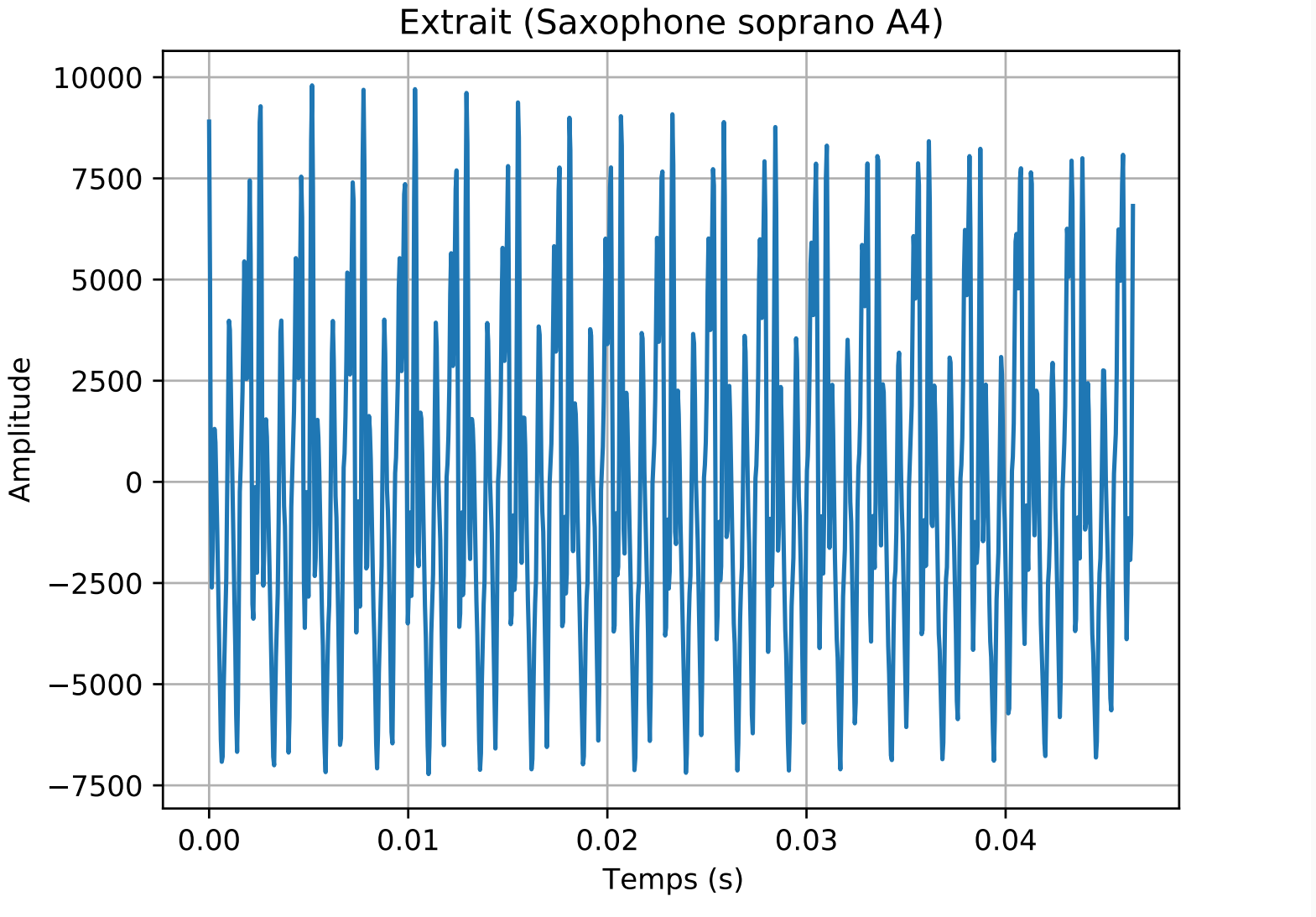
Extrait d'un saxophone soprano, A4

Acquisition à $\mathrm{F_{ech.}} = 22{,}05\,\mathrm{kHz}$ donc $\mathrm{d}t \simeq 0{,}0454\,\mathrm{ms}$ $(\mathrm{T_{ech.}})$

Pendant $\mathrm{T_{acq.}} = \mathrm{N} \times \mathrm{T_{ech.}} \simeq 46{,}39\,\mathrm{ms}$ $(\mathrm{N} = 1024$ points$)$

```
plt.plot(T, signal)
```

Exemple de signal



Analyse spectrale

Effectuer une analyse de Fourier de l'extrait est simple :

```python
from numpy.fft import rfft


spectre = rfft(signal)
```

Résolution et critère de Shanon

Résolution fréquentielle : $\frac{1}{\mathrm{T_{acq}}}$

`spectre[i]` correspond à la fréquence $f_i = i \times \frac{1}{\mathrm{T_{acq}}}$

N/2 valeurs, de 0 Hz à $\frac{1}{(2\mathrm{T_{ech}})} = \frac{\mathrm{N}}{2} \times \frac{1}{\mathrm{T_{acq.}}} = \frac{\mathrm{F_{ech}}}{2}$

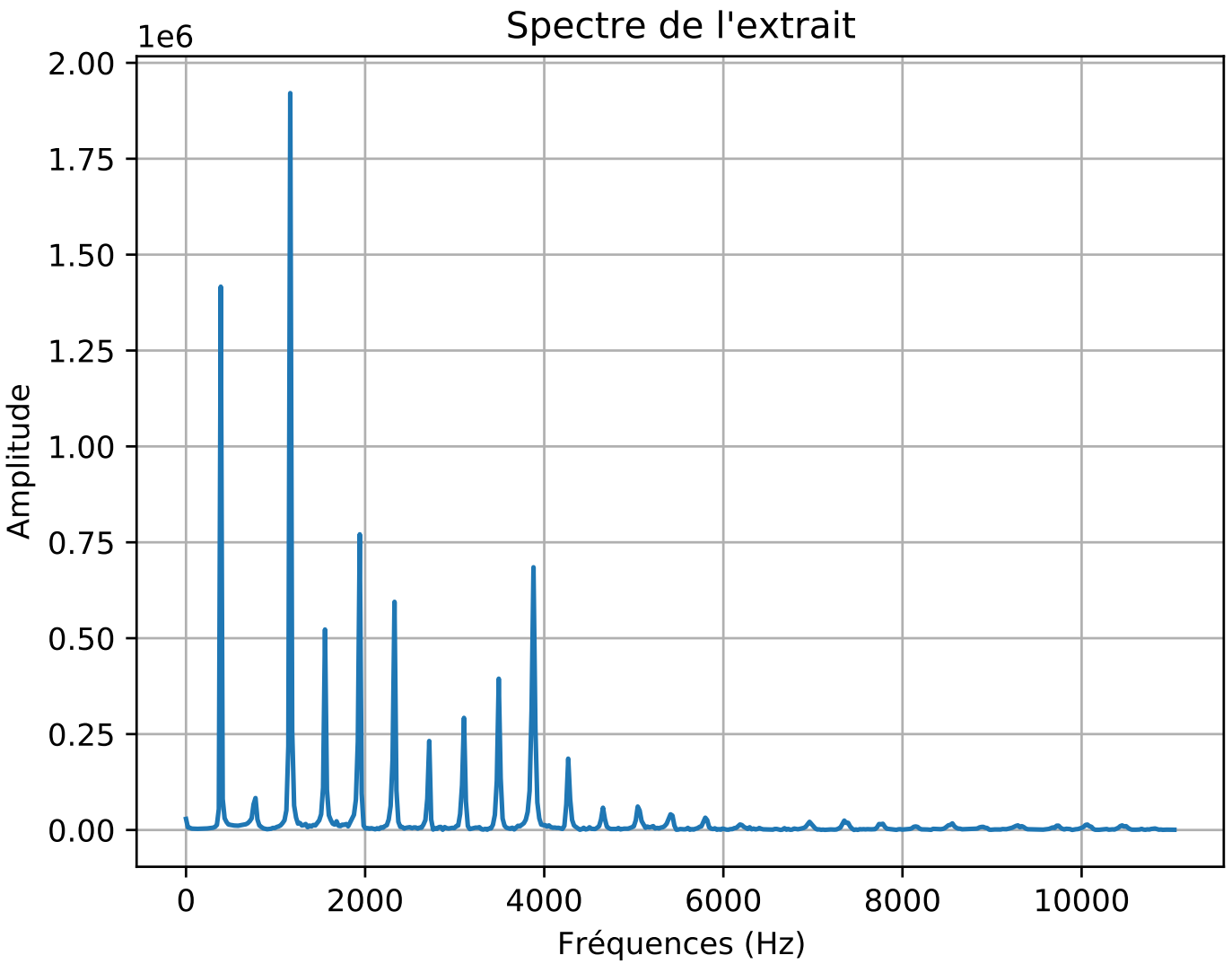
```
freqs = np.array([i/T_acq for i in range(N)])
```

Tracé du spectre

Pour tracer l'amplitude du spectre :

```python
from numpy.fft import rfft

spectre = rfft(signal)

freqs = np.array([i/T_acq for i in range(N)])

plt.plot(freqs, np.abs(spectre))
```

Analyse spectrale

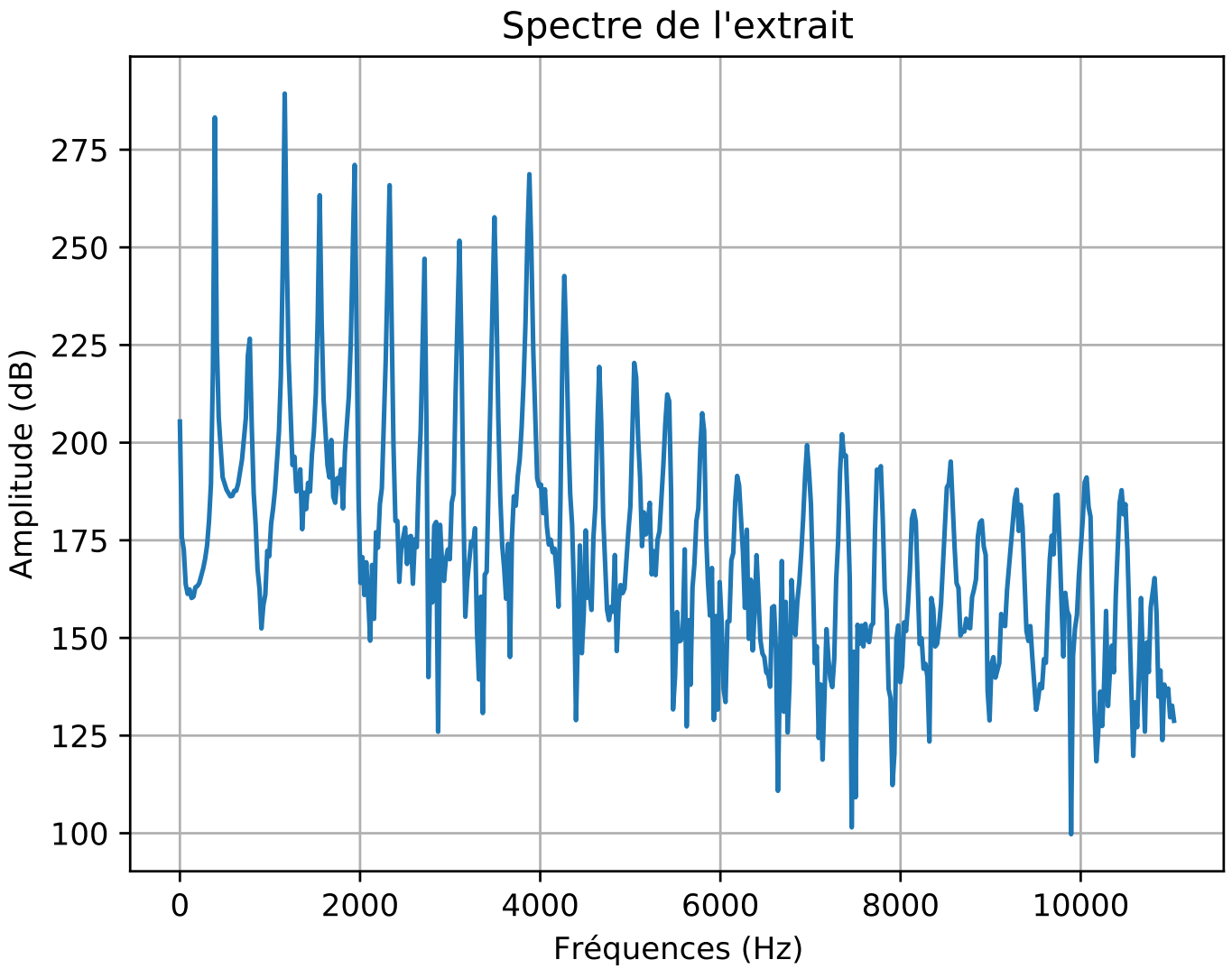


Analyse spectrale

Pour une échelle en décibels, juste un changement :

```
plt.plot(freqs, 20.0*np.log10(np.abs(spectre)))
```

Analyse spectrale

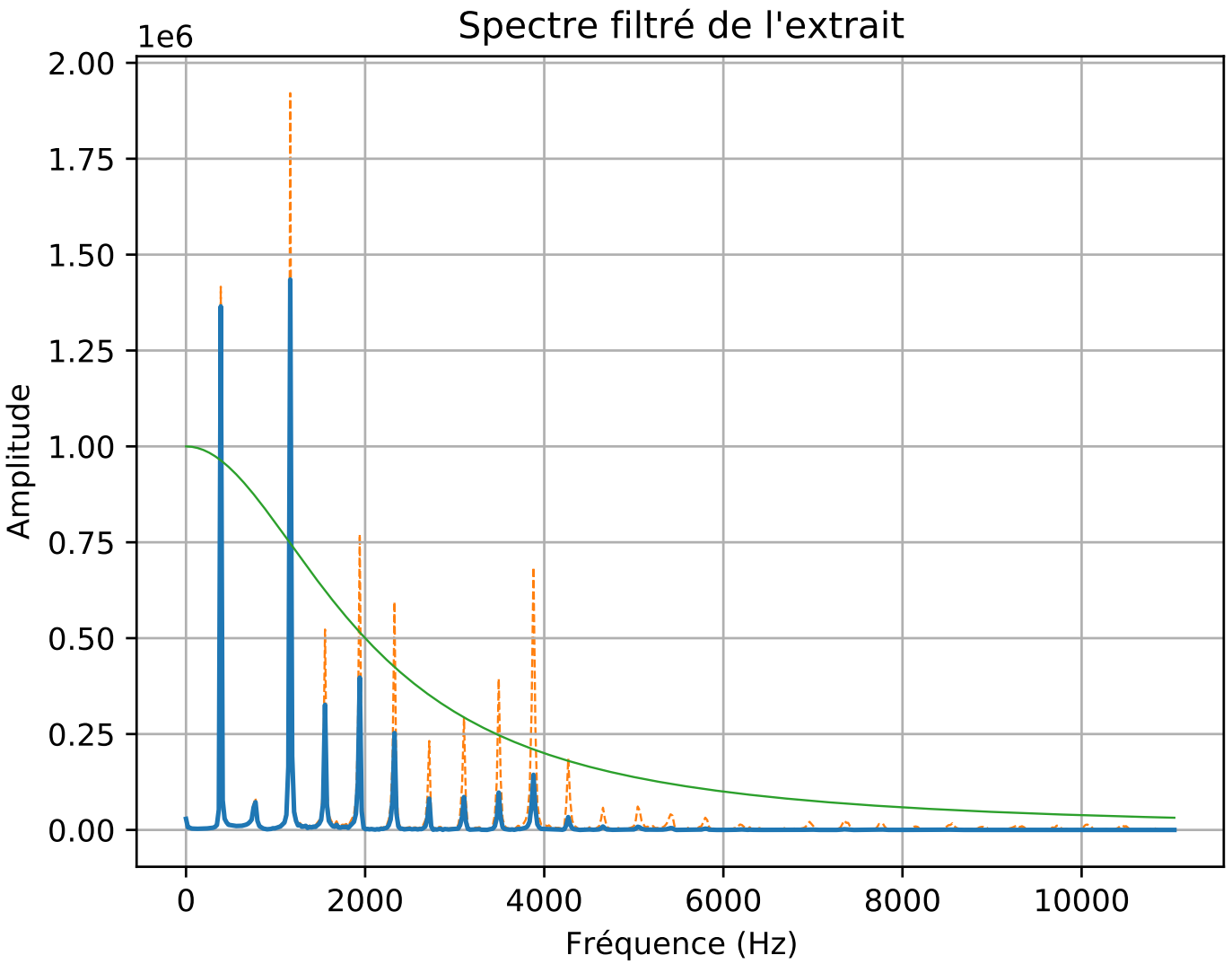


Filtrage passe-bas du second ordre

$$\mathrm{H}(f) = \frac{1}{\left(1 + \mathrm{j}\frac{f}{f_0}\right)^2}$$

```python
def H(f):
    fc = 2000.0 # Hz
    return 1 / (1+1j*f/fc)**2

spectre_filtre = spectre * H(freqs)

plt.plot(freqs, np.abs(spectre_filtre))
```

Spectre filtré

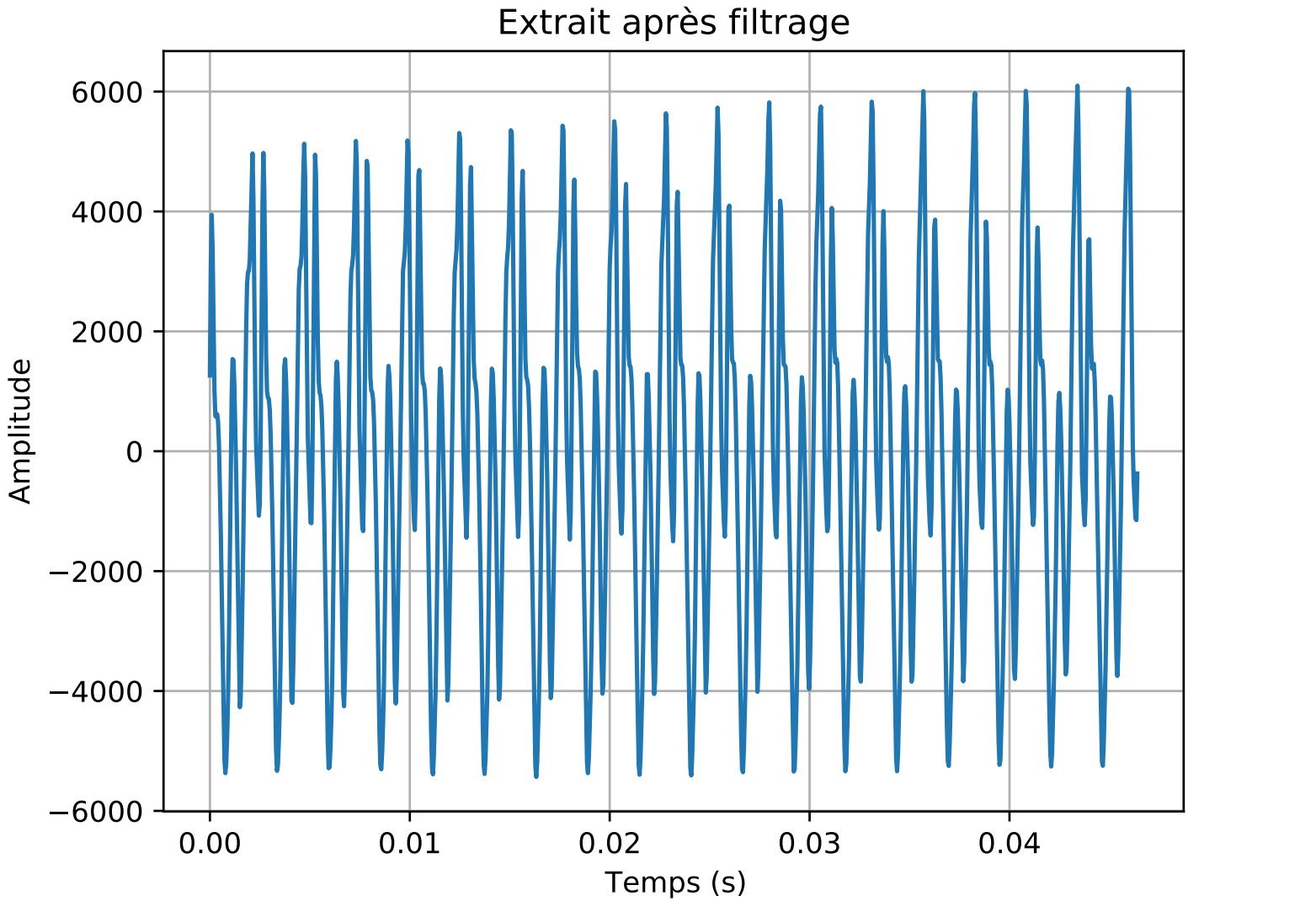


Signal filtré

numpy.fft.irfft fait la transformation inverse :

```python
from numpy.fft import irfft


signal_filtre = irfft(spectre_filtre)


plt.plot(T, signal_filtre)
```

Signal filtré



Moyenne glissante (SII)

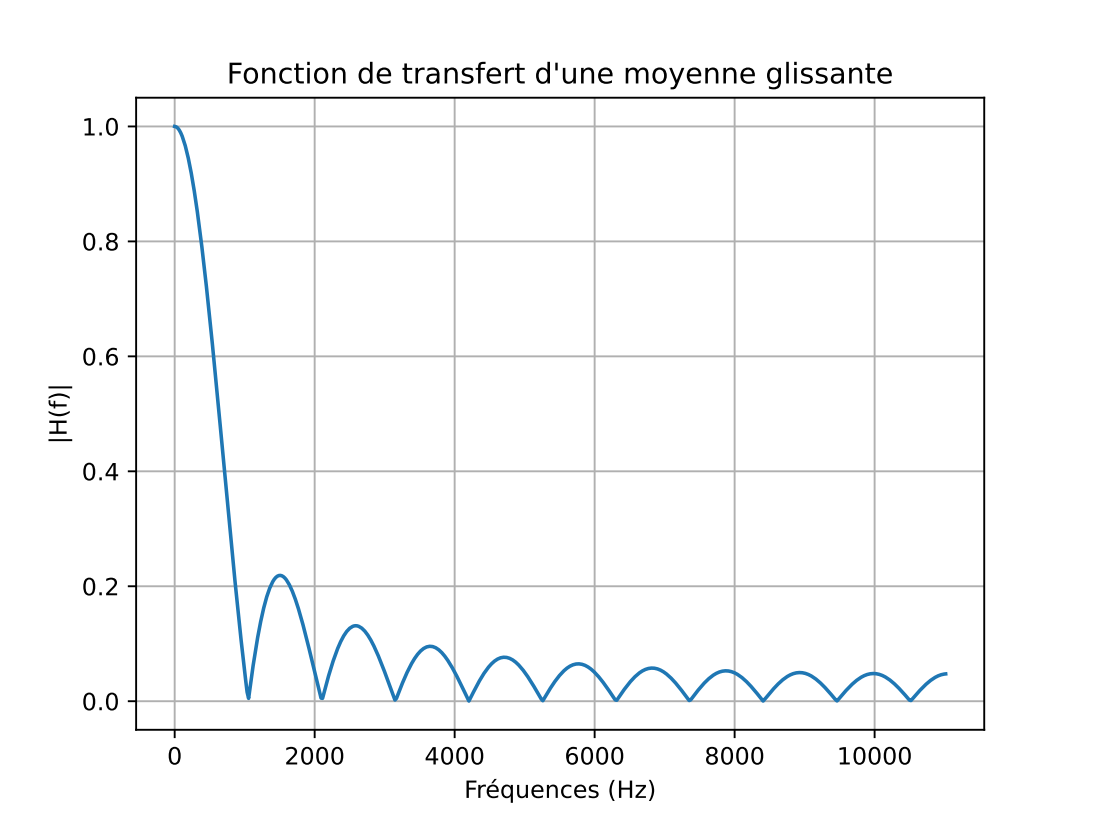
Méthode de filtrage numérique

$$s_i = \frac{1}{2k+1} \sum_{j=i-k}^{j+k} e_i$$

La définition aux bords peut être délicate

Moyenne glissante (SII)

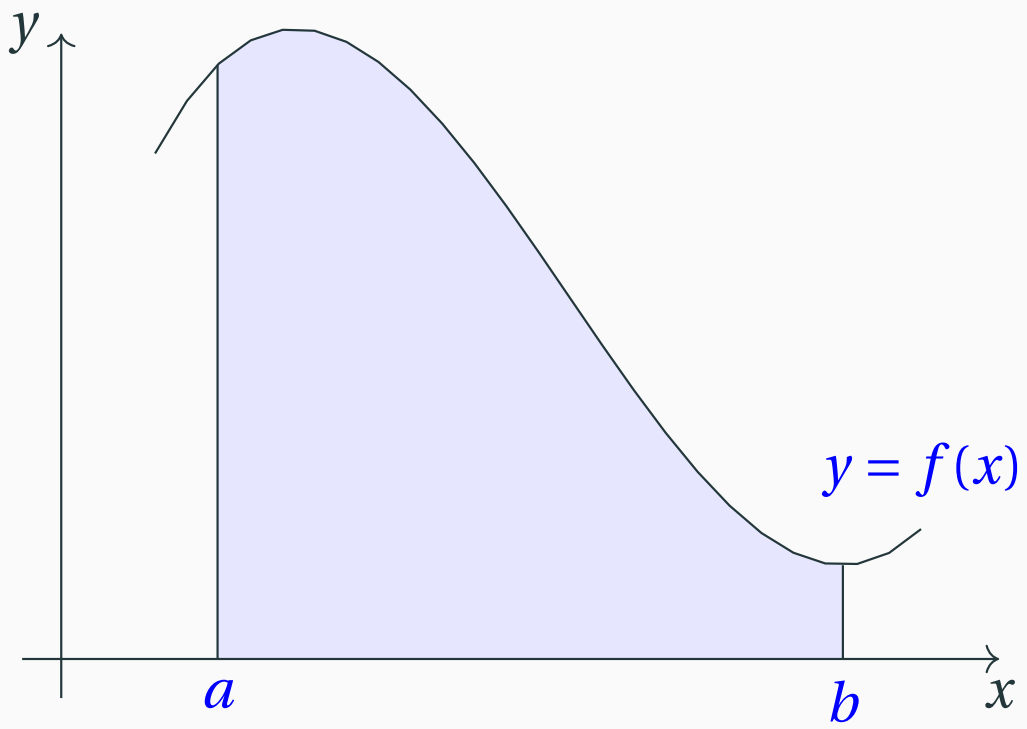
Il s'agit d'une sorte de filtre passe-bas :



Première annulation pour $f = \frac{1}{(2k+1)\,t_{\text{ech.}}}$

On souhaite déterminer $\mathrm{I}_{a,b} = \int_a^b f(u)\,\mathrm{d}u$,

soit l'aire « sous la courbe »



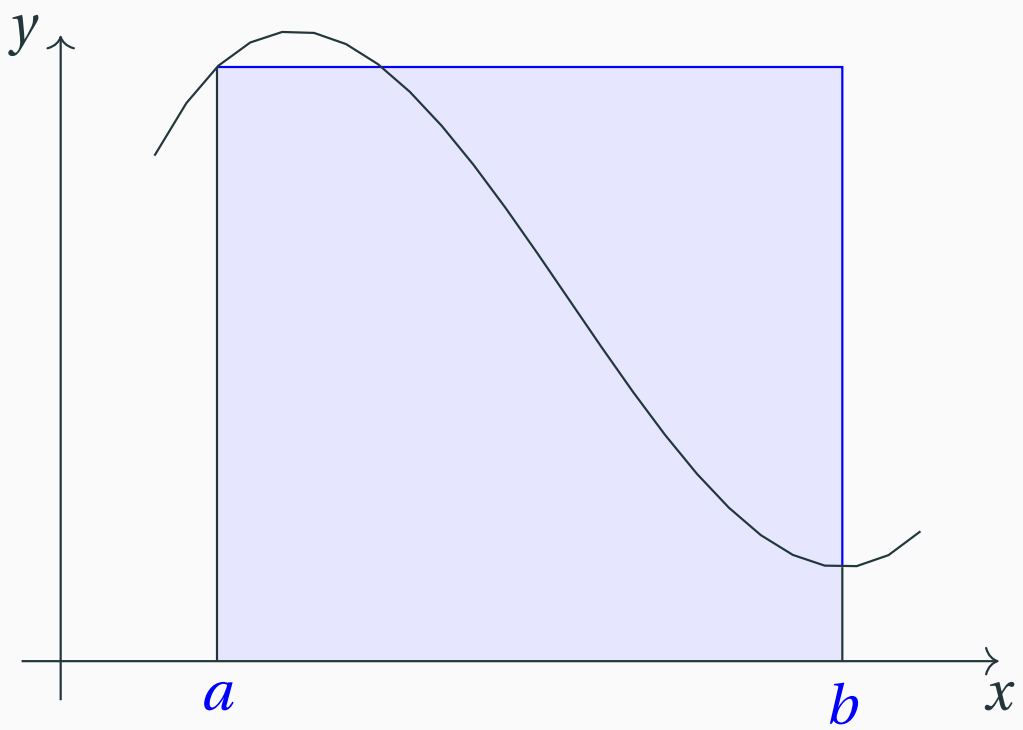
Méthode

On s'efforce d'approcher la surface « sous la courbe » par une forme géométrique simple dont on sait calculer la surface (rectangle, trapèze, etc.)

Dans la suite, on notera $\mathrm{J}_{a,b}$ les estimations de $\mathrm{I}_{a,b} = \int_a^b f(u)\,\mathrm{d}u$.

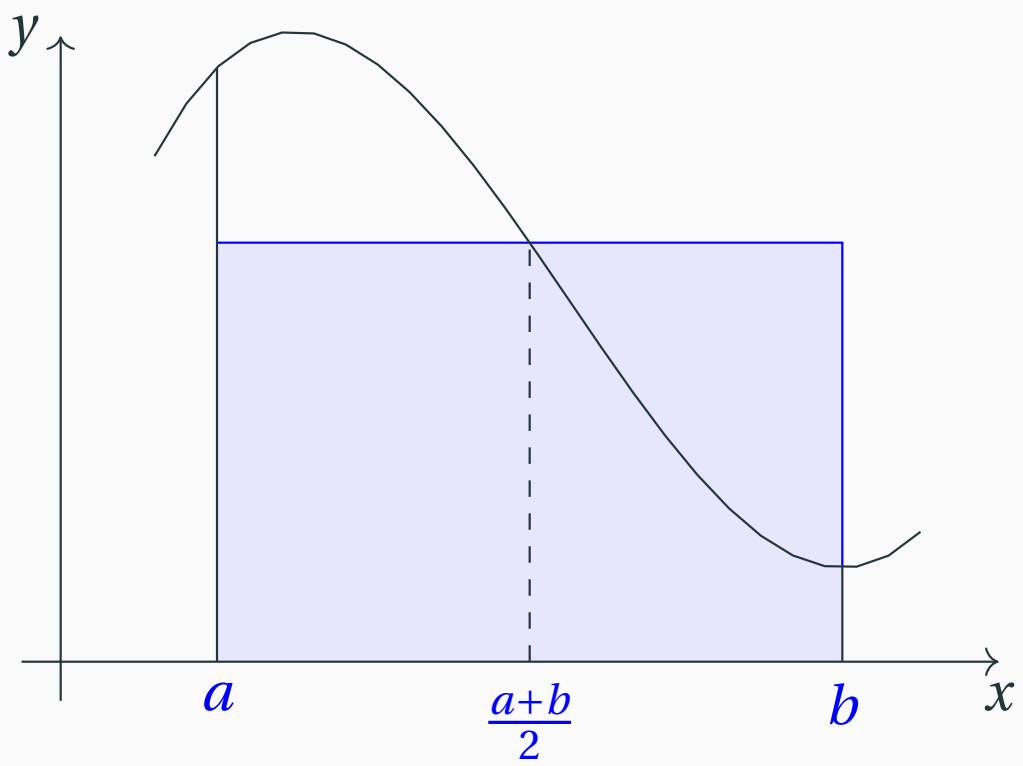
Méthode du rectangle gauche

$$\mathrm{J}_{a,b} = f(a) \times (b - a)$$



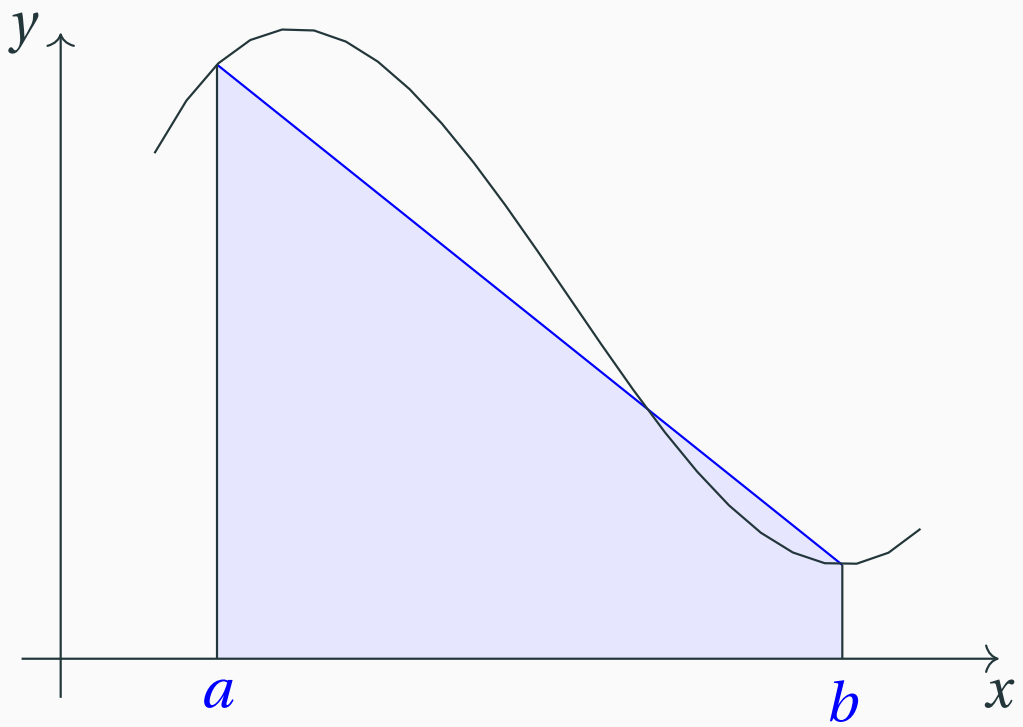
Méthode du point milieu

$$\mathrm{J}_{a,b} = f\left(\frac{a+b}{2}\right) \times (b-a)$$



Méthode du trapèze

$$\mathrm{J}_{a,b} = \frac{f(a) + f(b)}{2} \times (b - a)$$



Aller plus loin

Cela ne donne pas une bonne approximation en général

On utilise alors des méthodes *composites*

On divise l'intervalle $[a, b]$ en n intervalles :

$$a = u_0 < u_1 < u_2 < \ldots < u_{n-1} < u_n = b$$

L'intégrale recherchée est une somme d'intégrales sur ces sous-intervalles

$$\int_a^b f(\xi)\ \mathrm{d}\xi = \int_{u_0}^{u_1} f(\xi)\ \mathrm{d}\xi + \int_{u_1}^{u_2} f(\xi)\ \mathrm{d}\xi + \ldots + \int_{u_{n-1}}^{u_n} f(\xi)\ \mathrm{d}\xi$$

On approche donc l'intégrale en estimant chacun de ces termes avec les méthodes précédentes.

Choix des u_i

On choisit généralement des u_i régulièrement espacés

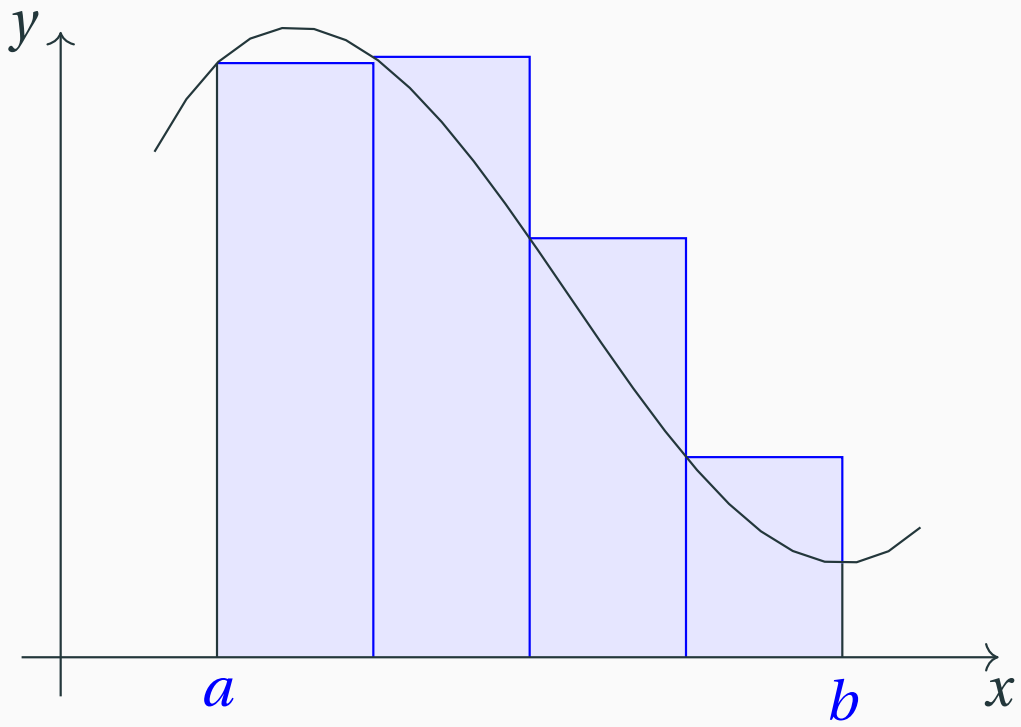
$$u_i = a + \frac{b-a}{n} \times i$$

On a bien, ainsi, n intervalles vérifiant

$$a = u_0 < u_1 < u_2 < \ldots < u_{n-1} < u_n = b$$

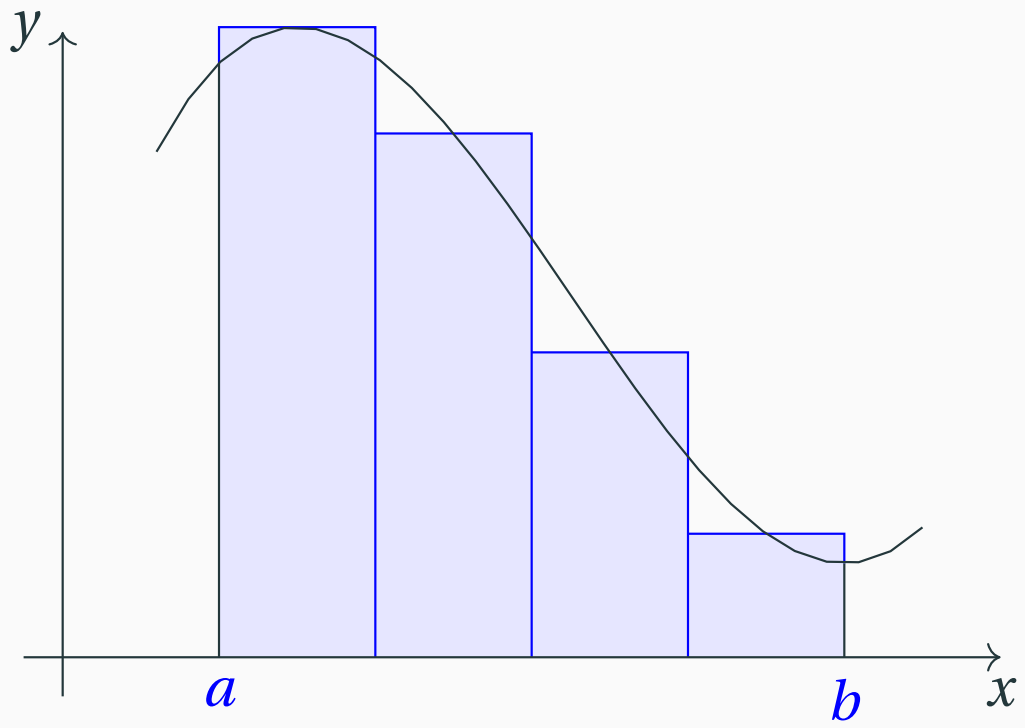
Méthode des rectangles gauches

$$\mathrm{J}_{[a,b],n}(f) = \sum_{i=0}^{n-1} \frac{b-a}{n} f\left(a + i \times \frac{b-a}{n}\right)$$



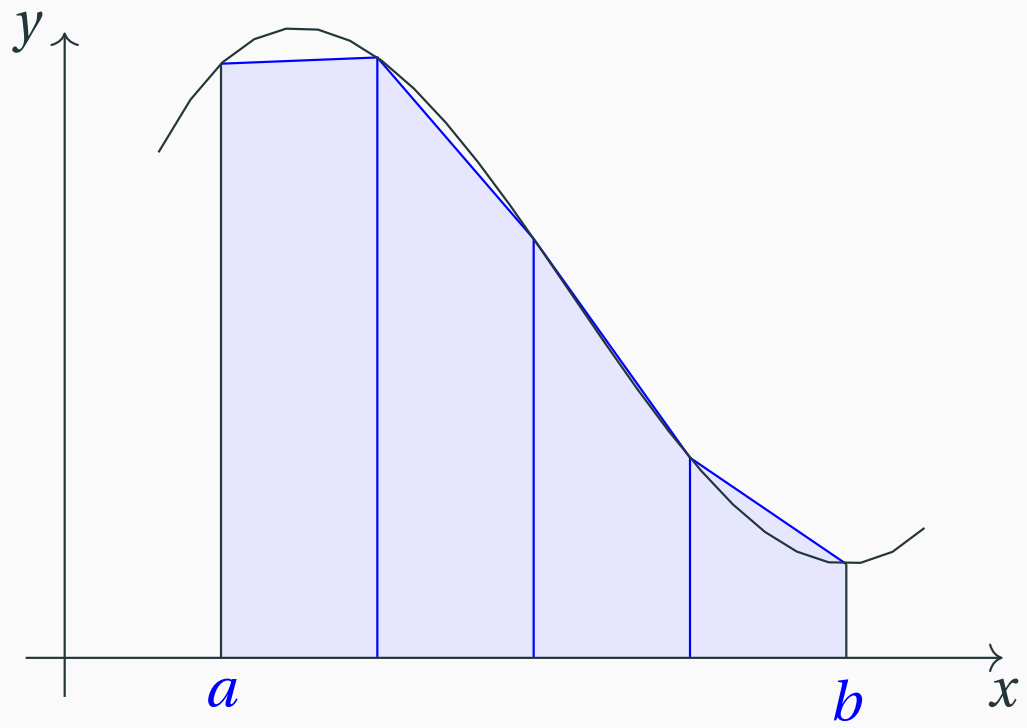
Méthode des points milieux

$$\mathrm{J}_{[a,b],n}(f) = \sum_{i=0}^{n-1} \frac{b-a}{n} f\left(a + \left(i + \frac{1}{2}\right) \times \frac{b-a}{n}\right)$$



Méthode des trapèzes

$$\mathrm{J}_{[a,b],n}(f) = \sum_{i=0}^{n-1} \frac{b-a}{2n} \times \left[f\left(a + i \times \frac{b-a}{n}\right) + f\left(a + (i+1) \times \frac{b-a}{n}\right) \right]$$



Intégrale d'une fonction

$$\mathrm{J}_{[a,b],n}(f) = \frac{b-a}{n} \times \left[\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{n-1} f\left(a + i \times \frac{b-a}{n}\right) \right]$$

```python
def trapezes(f, a, b, N):
    I = f(a)/2 + f(b)/2

    d = (b-a)/N

    for i in range(1, N):
        I = I + f(a+i*d)

    return I*d
```

Intégrale d'un signal

Si un signal se trouve dans un tableau `signal`

```python
def integrale(signal, dt):
    I = 0
    for i in range(len(signal)-1):
        I = I + (signal[i]+signal[i+1])/2  # trapèzes
    return I*dt
```

Primitive d'un signal

Si un signal se trouve dans un tableau `signal`

```python
def primitive(signal, dt, c0):
    Pr = [c0]
    for i in range(len(signal)-1):
        delta_y = (signal[i]+signal[i+1])/2 * dt
        Pr.append(Pr[-1] + delta_y)
    return Pr
```

Dérivation numérique

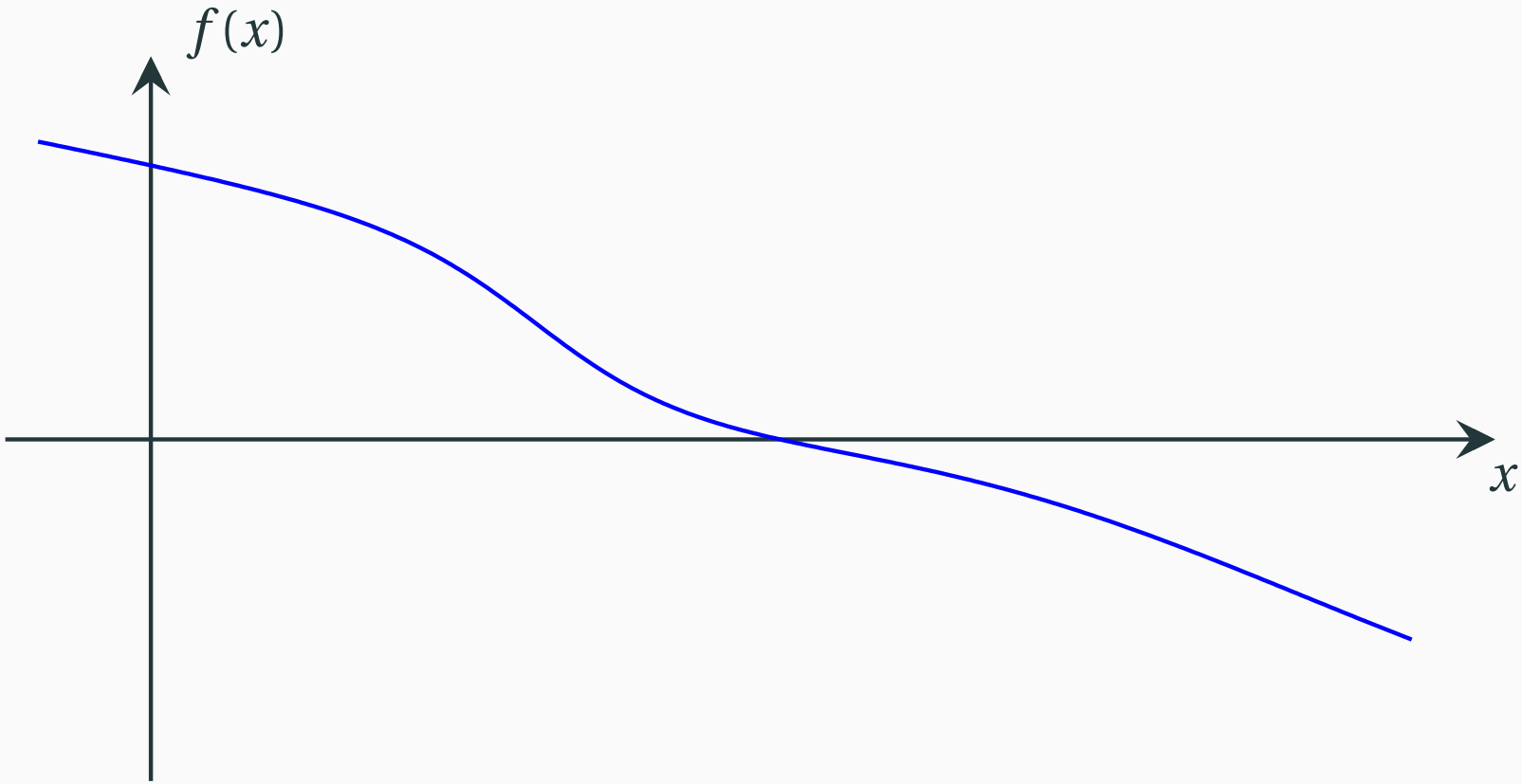
Si un signal se trouve dans un tableau `signal`

```python
def derivee(signal, dt):
    Der = []
    for i in range(len(signal)-1):
        Der.append((signal[i+1]-signal[i]) / dt)
    return Der
```

Attention, un point de moins !

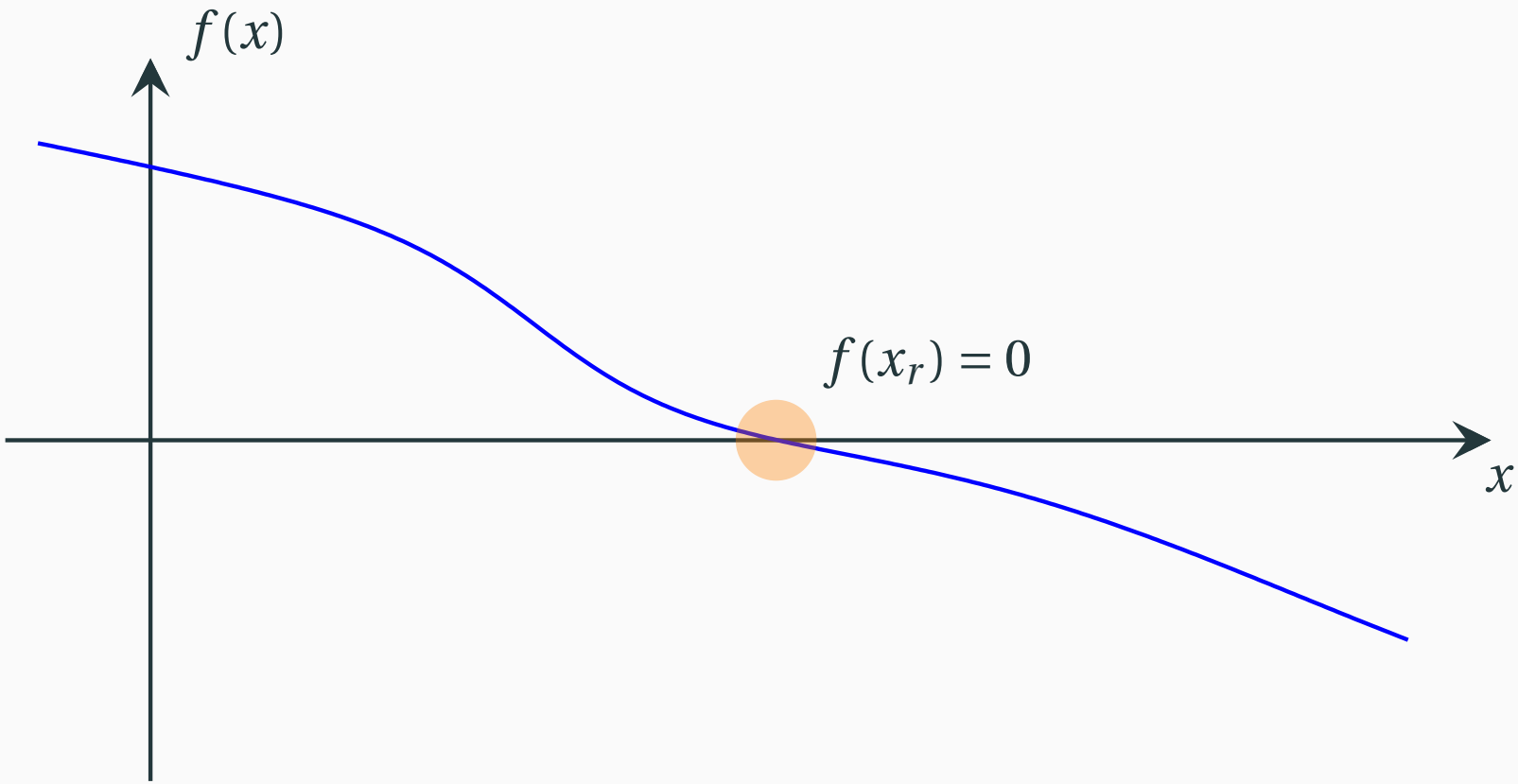
Objectifs

On considère une fonction f réelle d'une variable réelle



Objectifs

On considère une fonction f réelle d'une variable réelle



On cherche une approximation $\hat{x}_r$ d'un de ses « zéros » x_r

Recherches par encadrement

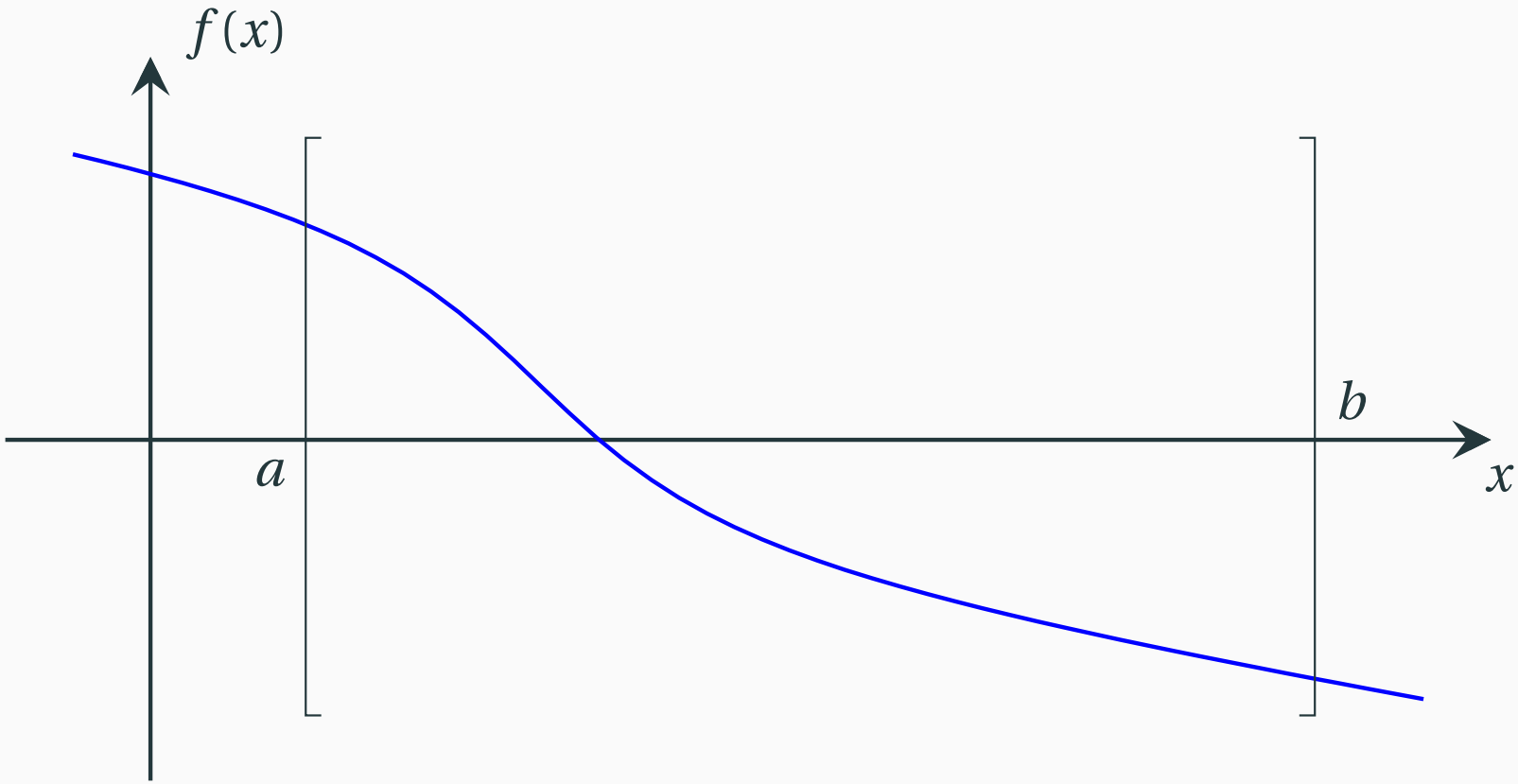
On suppose :

- on connaît a et b tels que $f(a) \times f(b) \leqslant 0$
- f est continue sur $[a, b]$

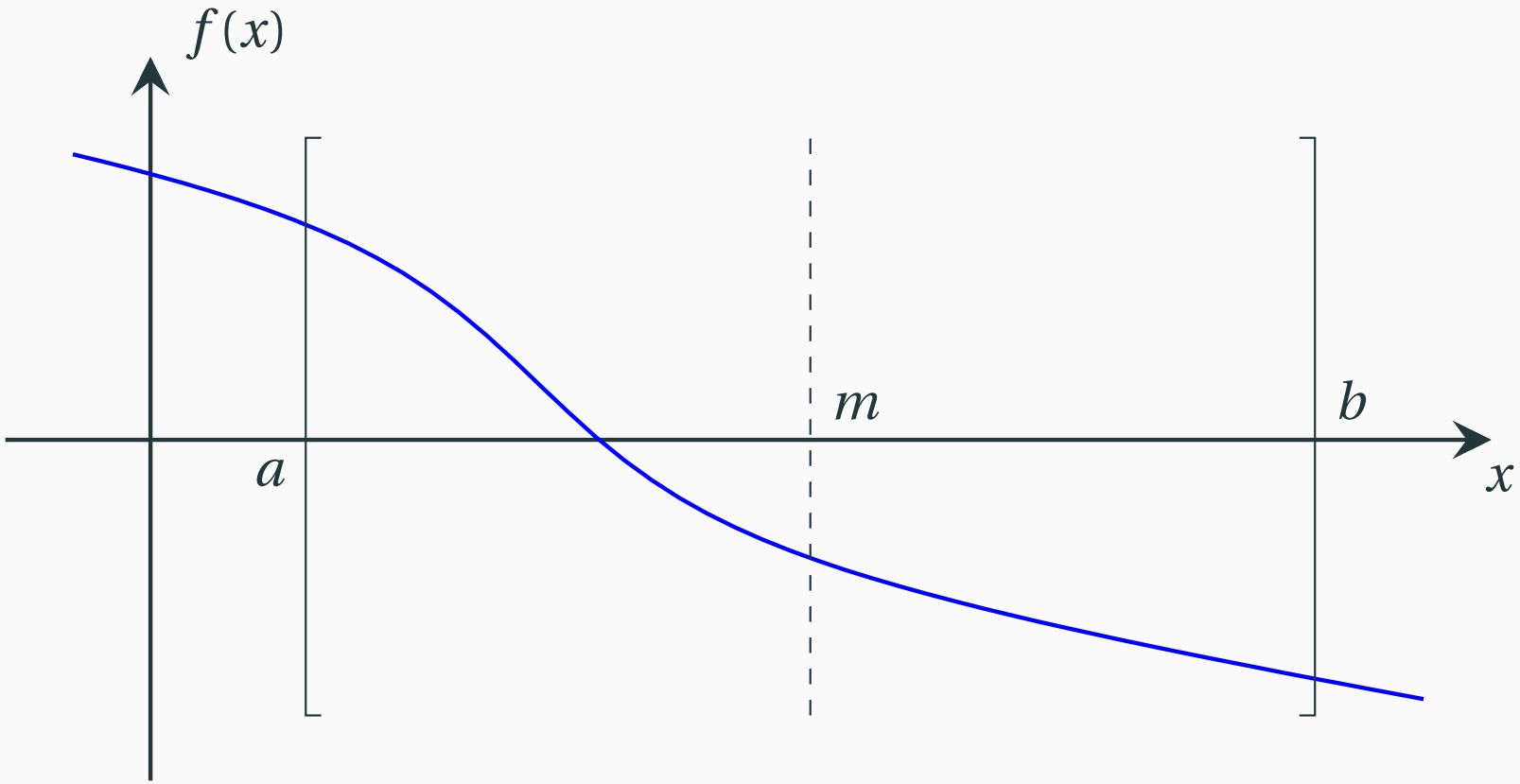
D'après le T.V.I., il y a au moins un zéro dans $[a, b]$

On va essayer d'affiner l'encadrement d'un zéro

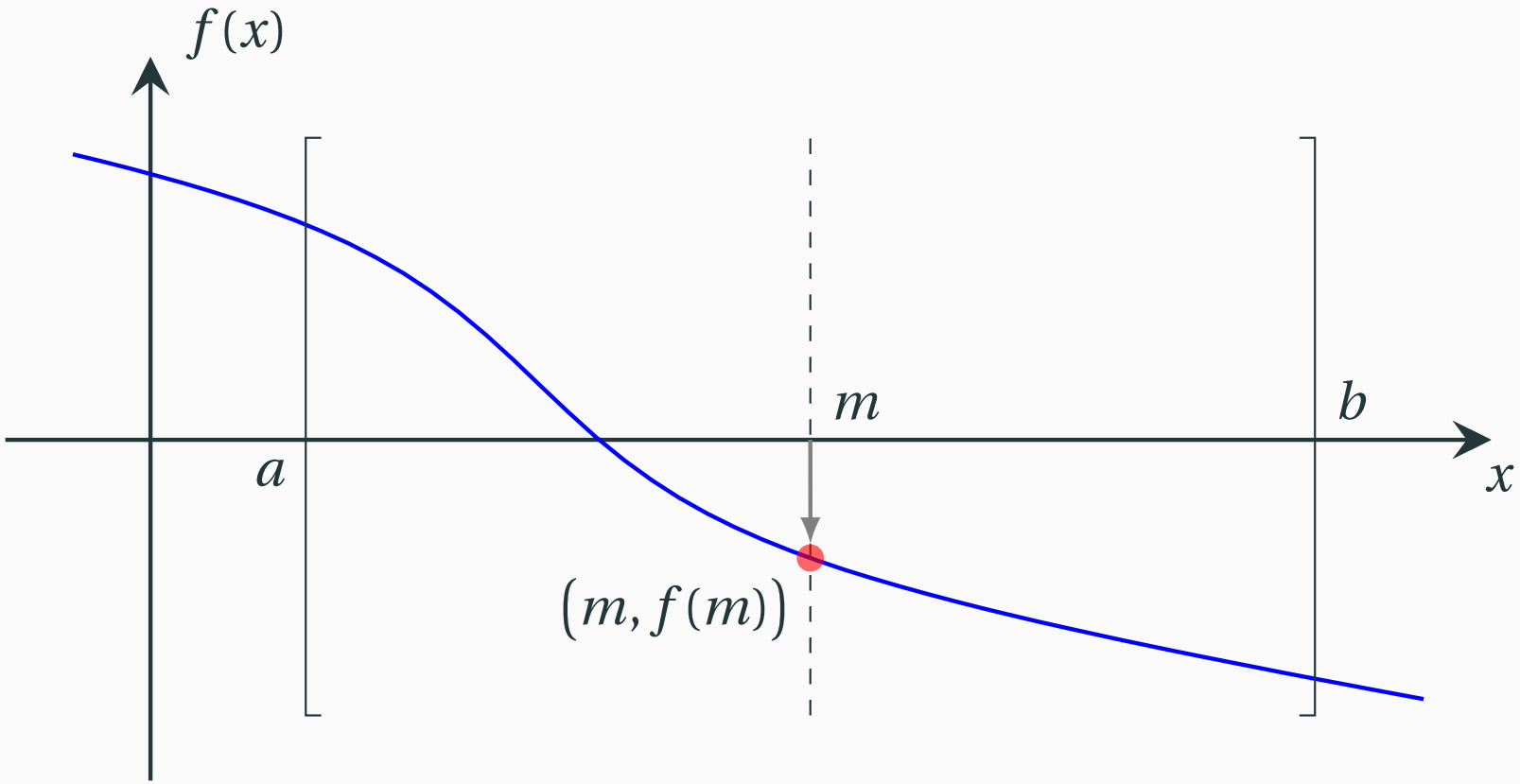
Recherche dichotomique



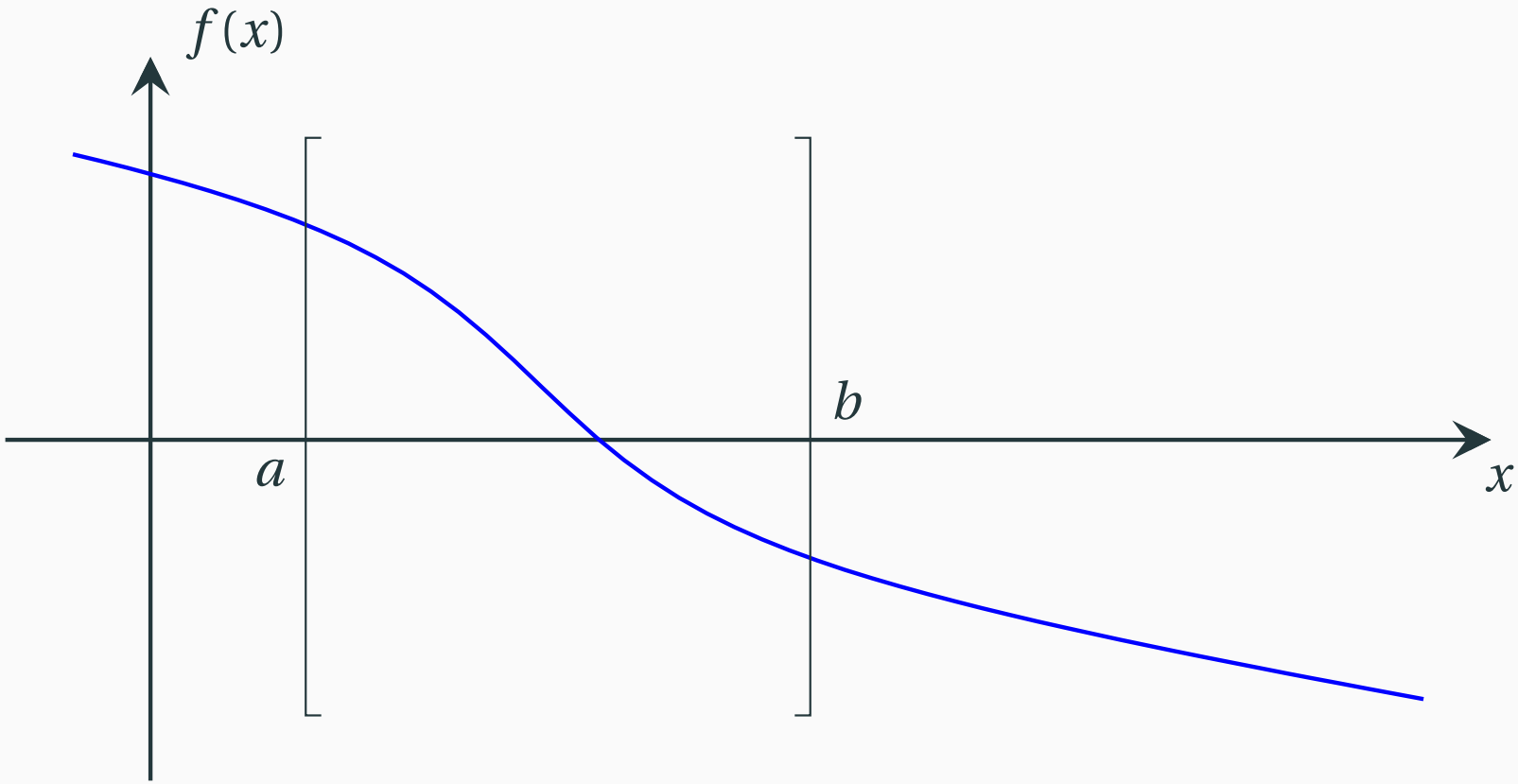
Recherche dichotomique



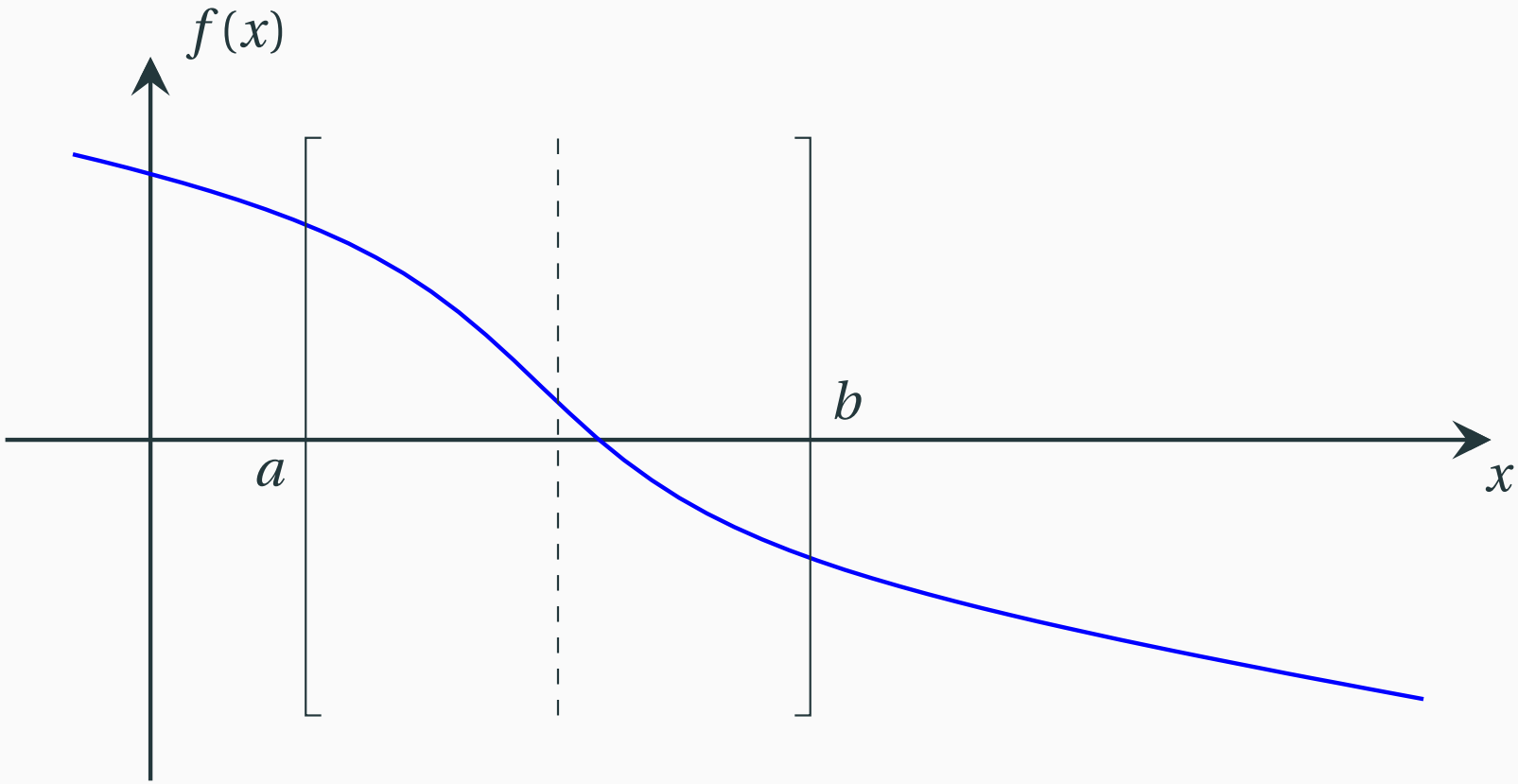
Recherche dichotomique



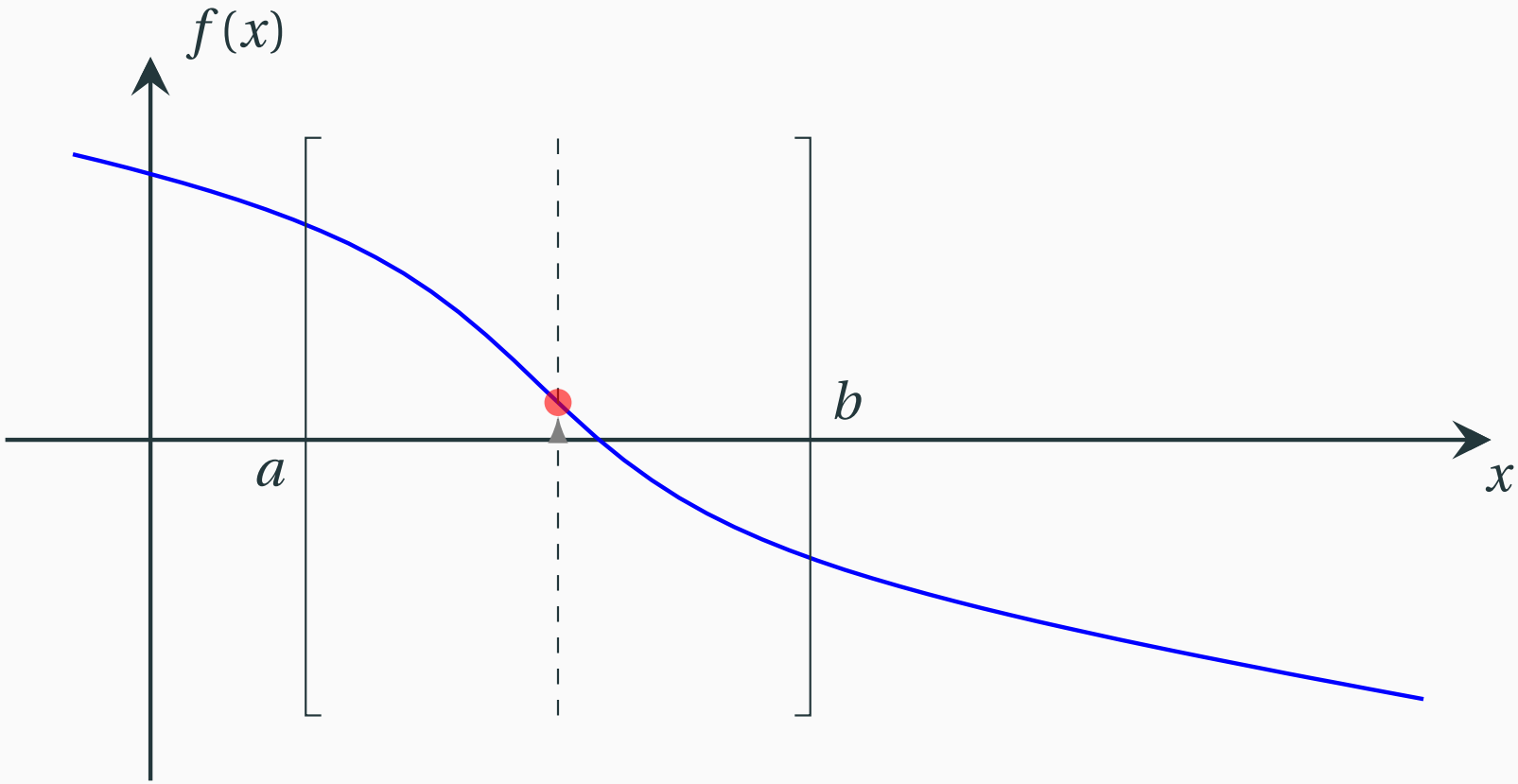
Recherche dichotomique



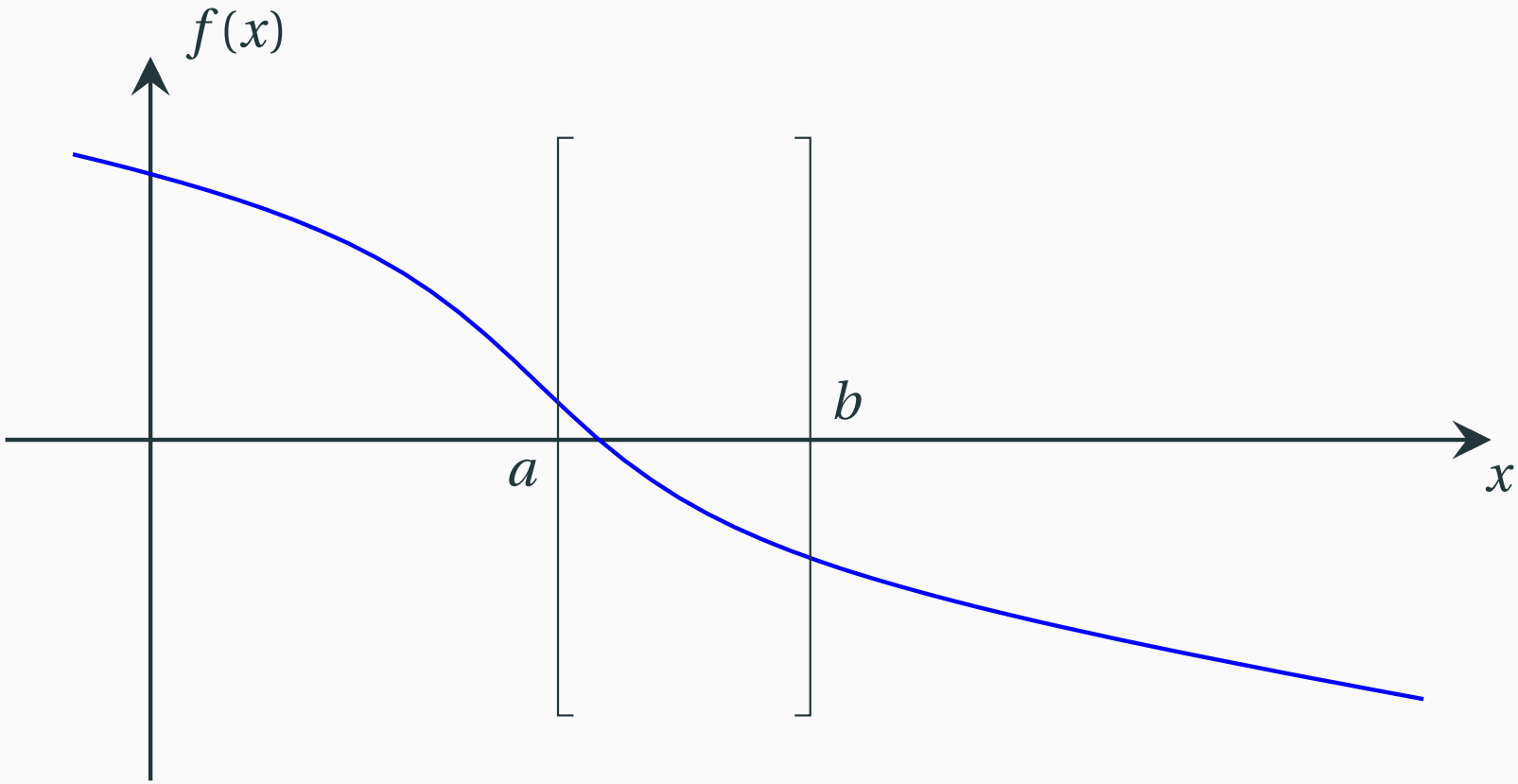
Recherche dichotomique



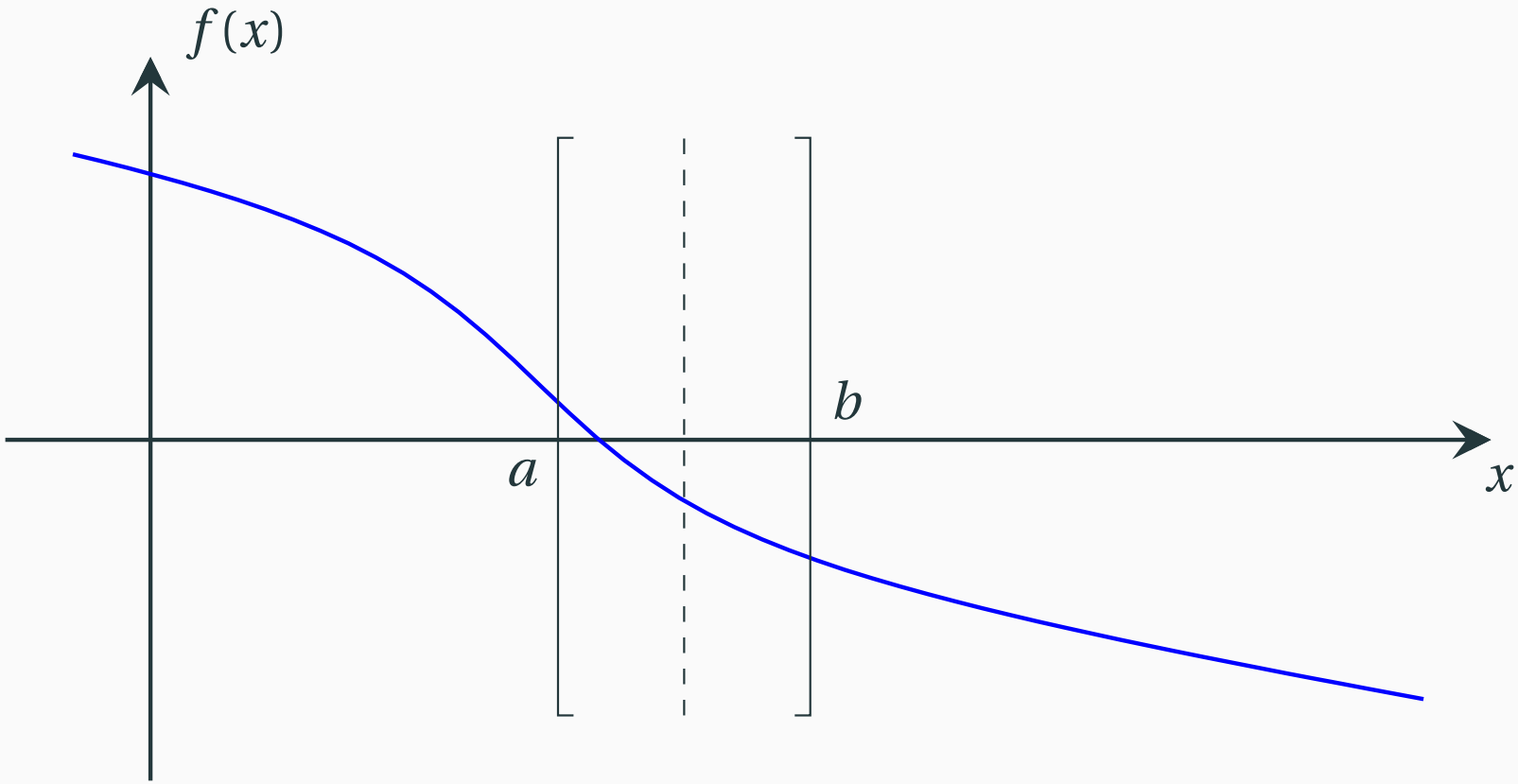
Recherche dichotomique



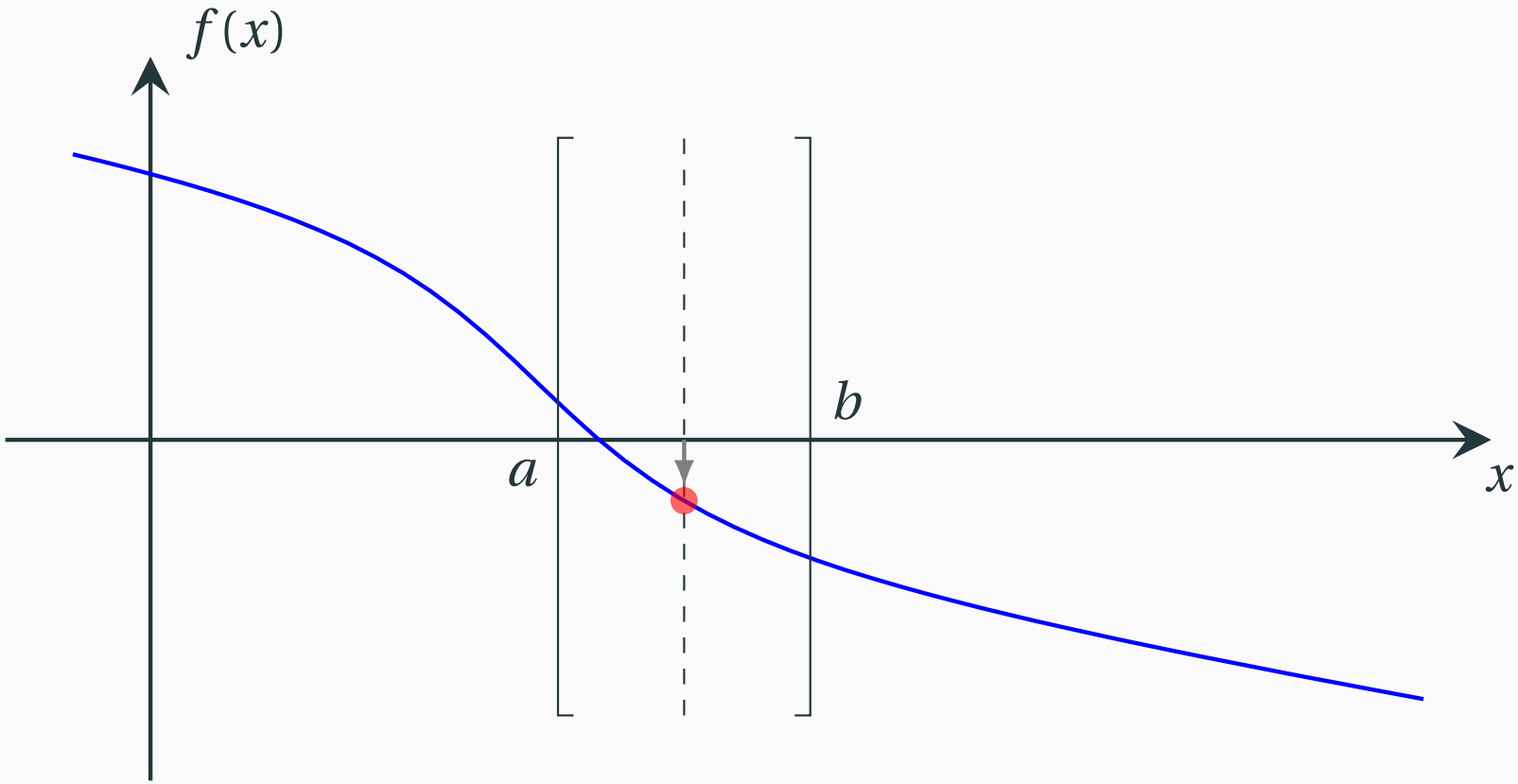
Recherche dichotomique



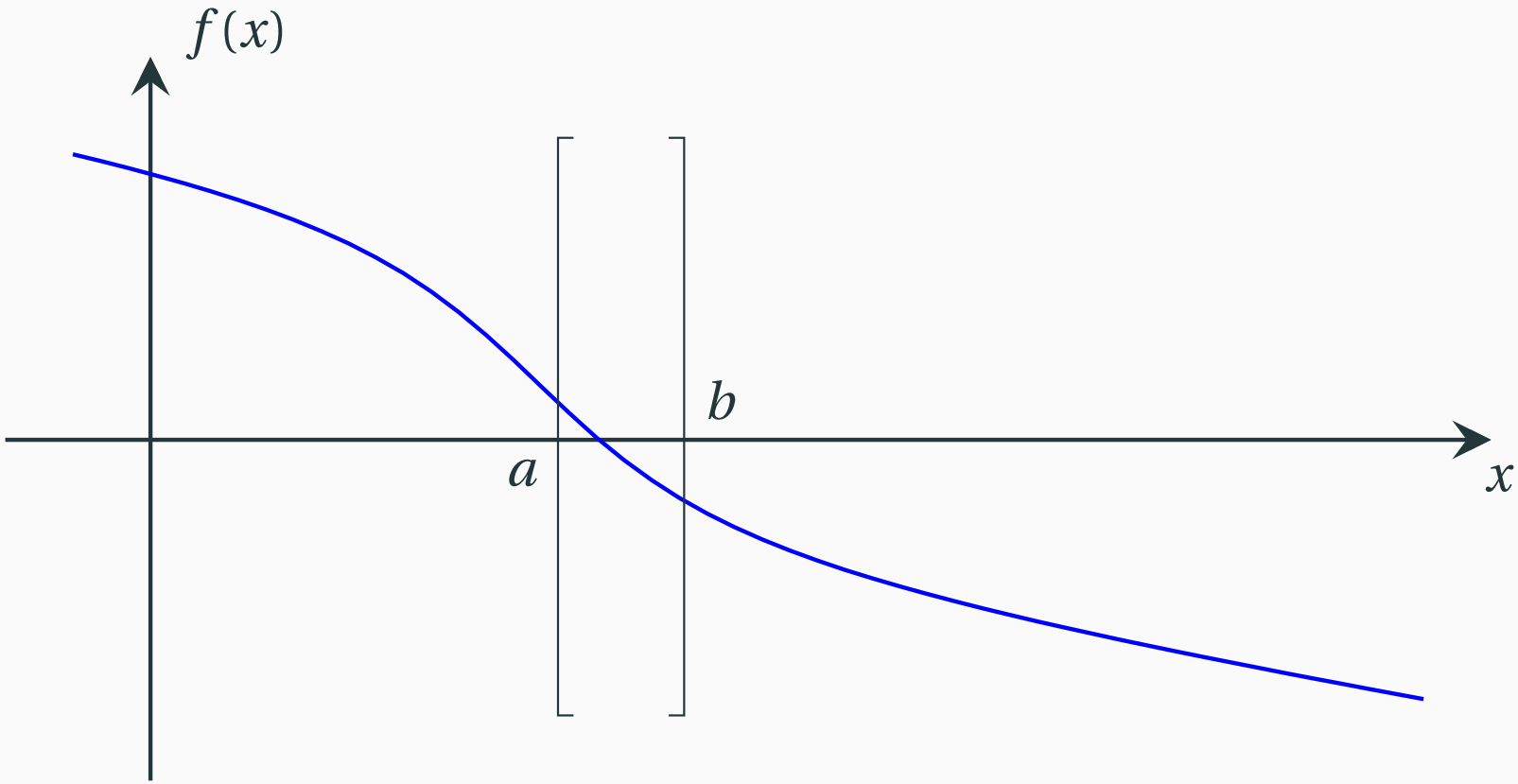
Recherche dichotomique



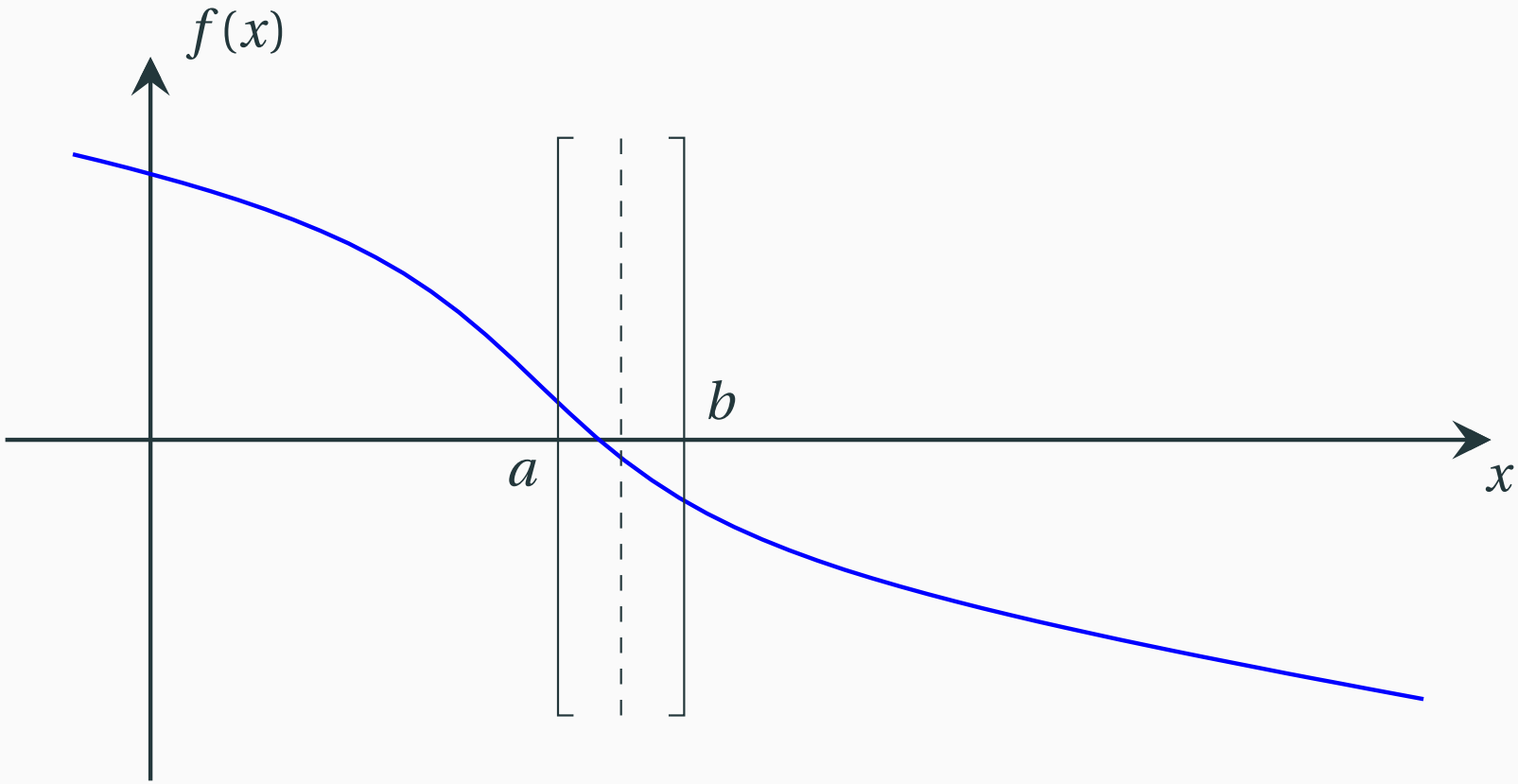
Recherche dichotomique



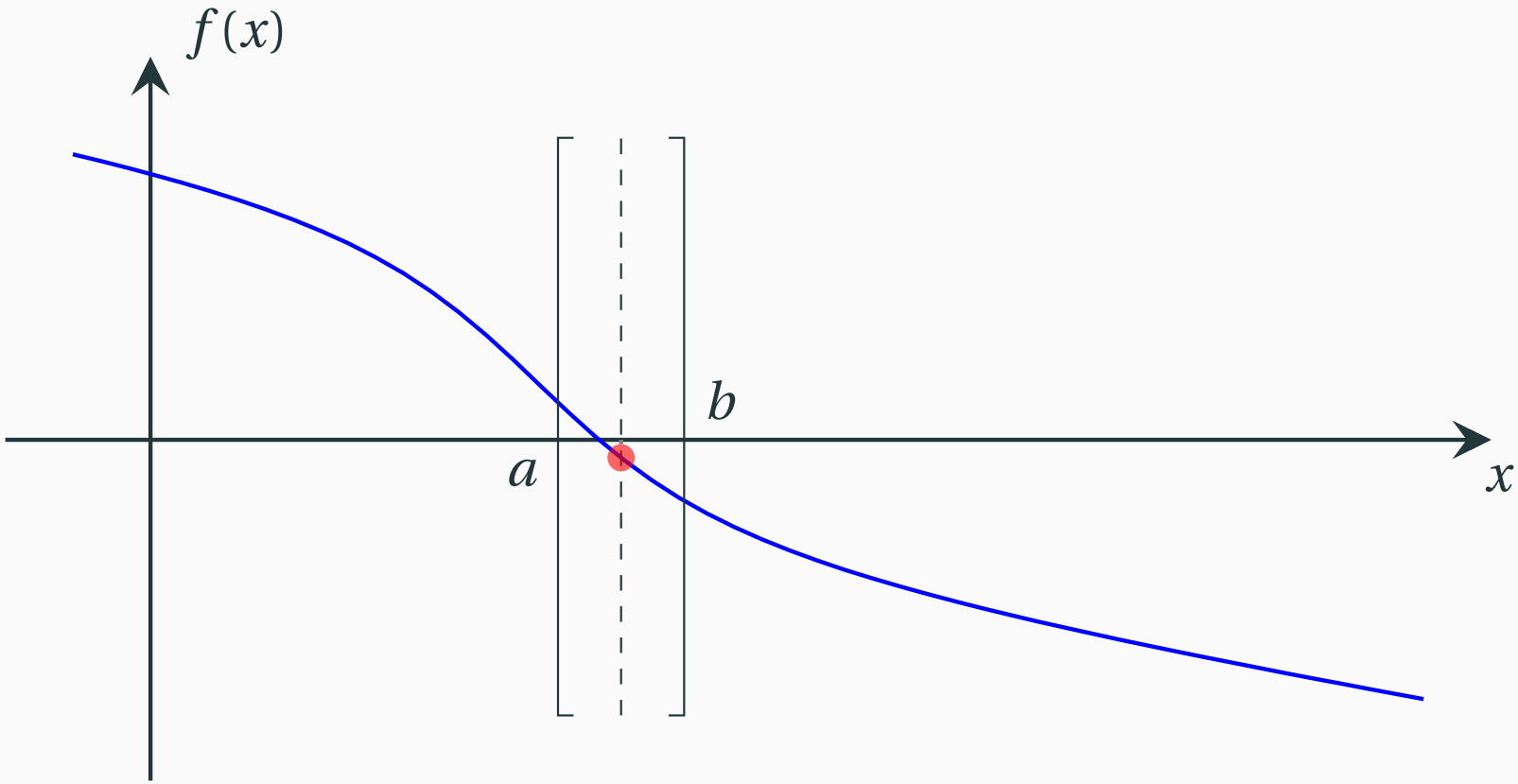
Recherche dichotomique



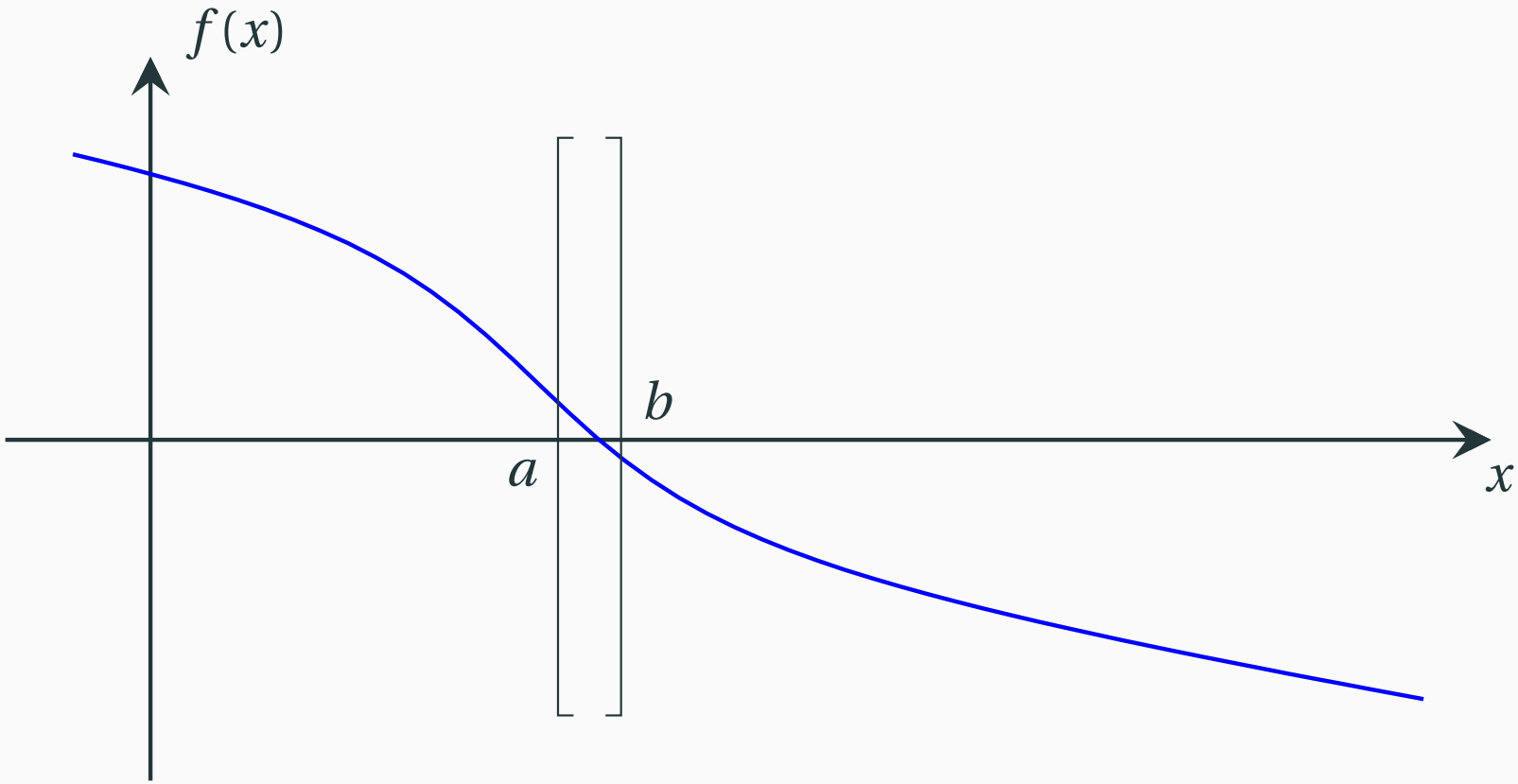
Recherche dichotomique



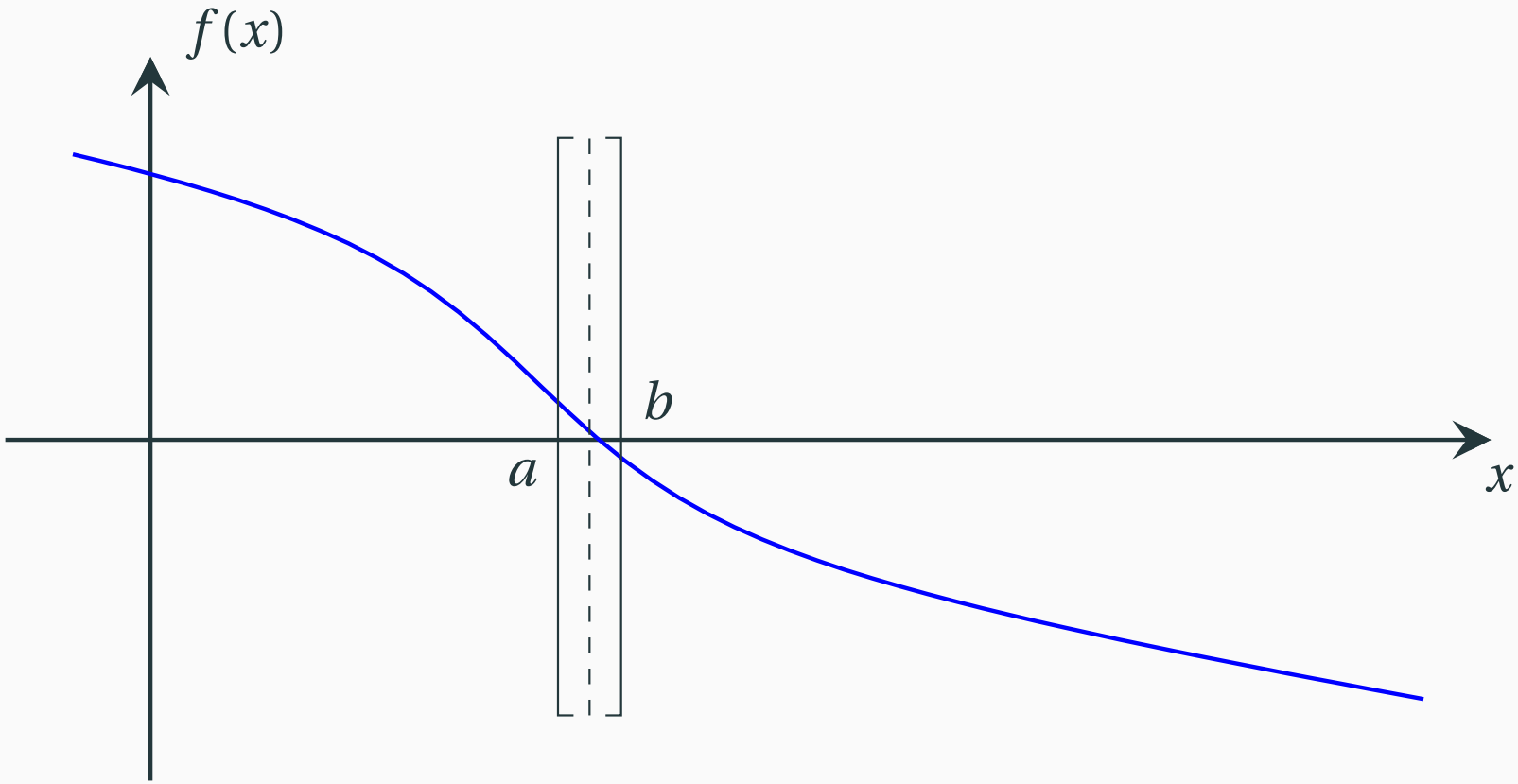
Recherche dichotomique



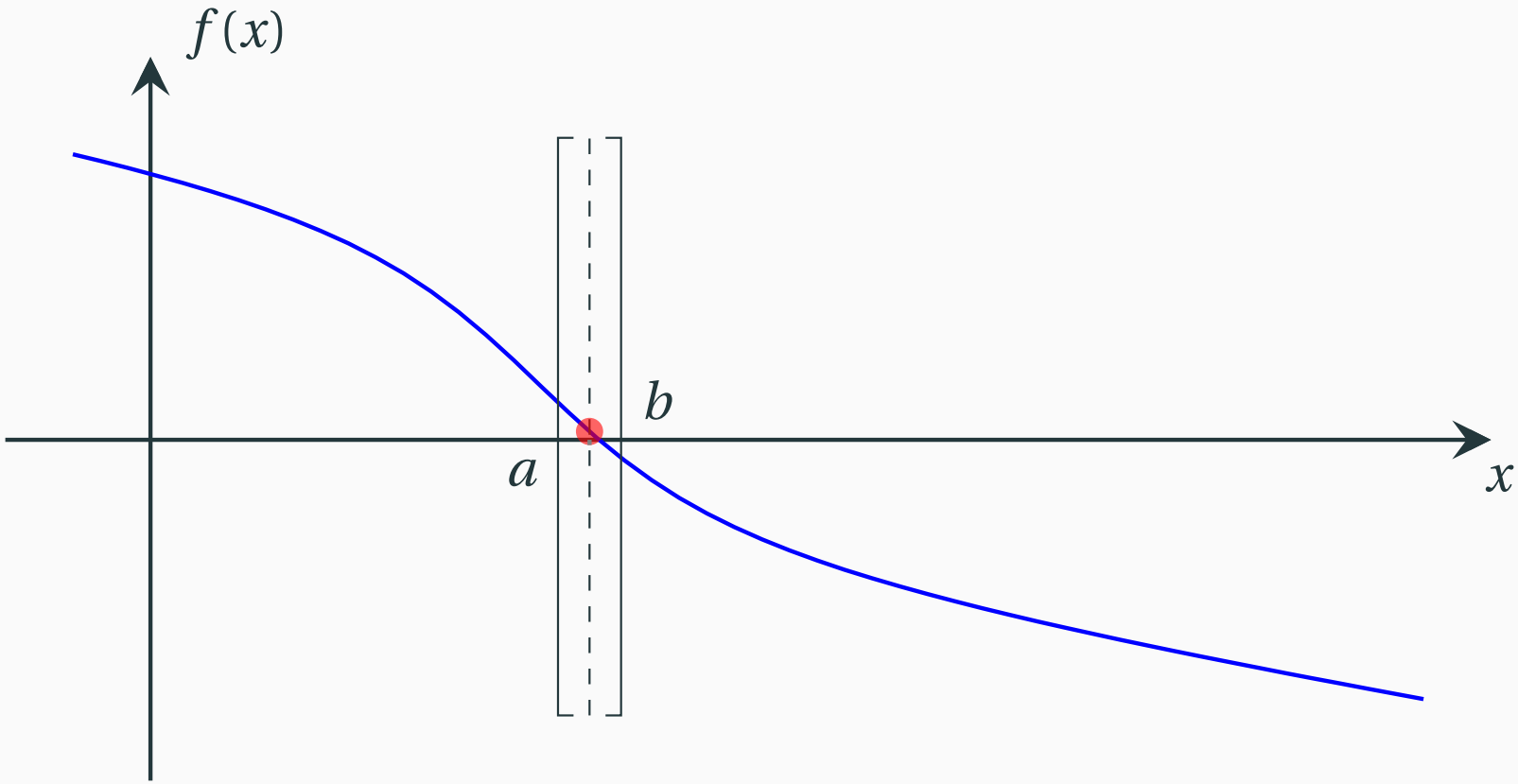
Recherche dichotomique



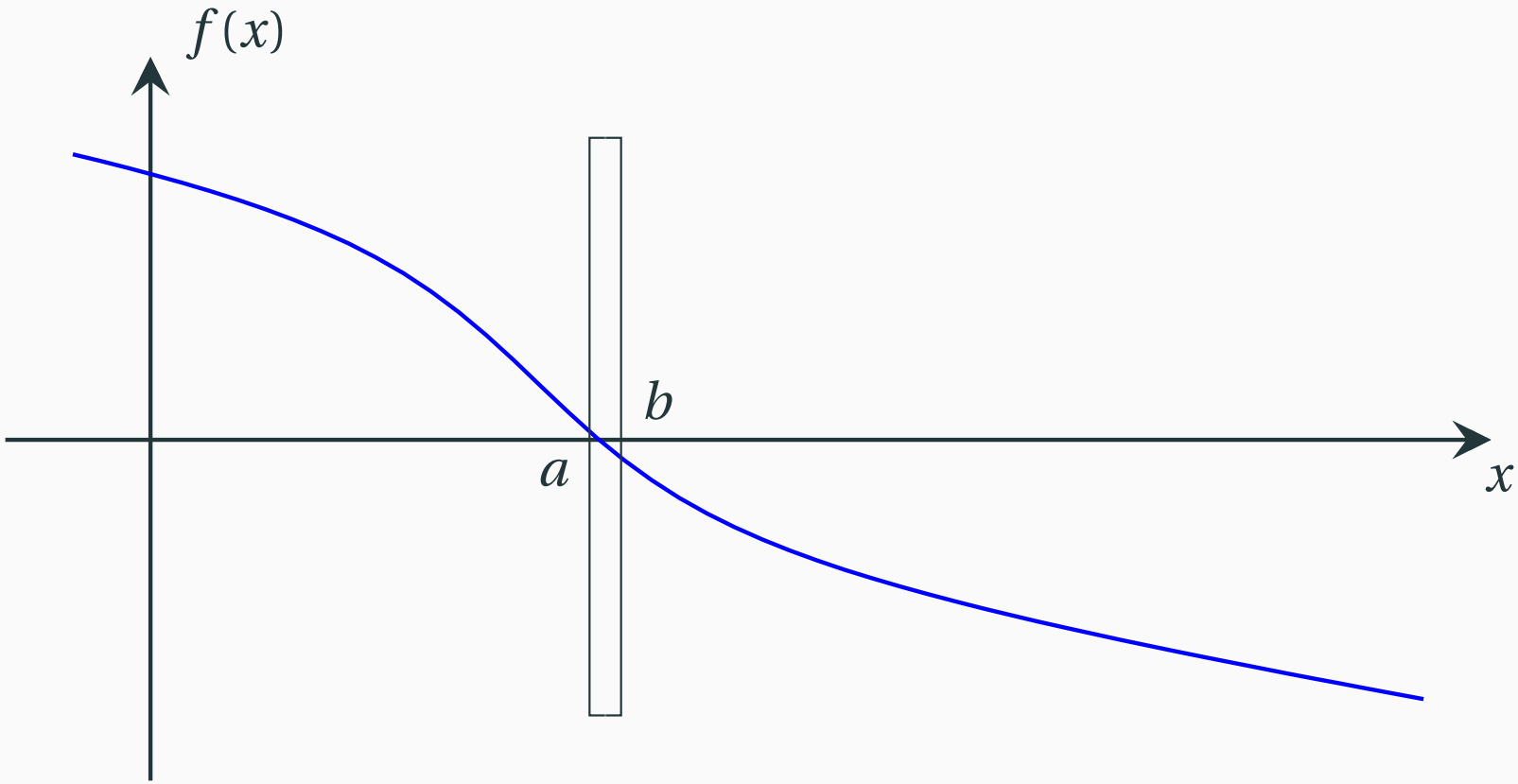
Recherche dichotomique



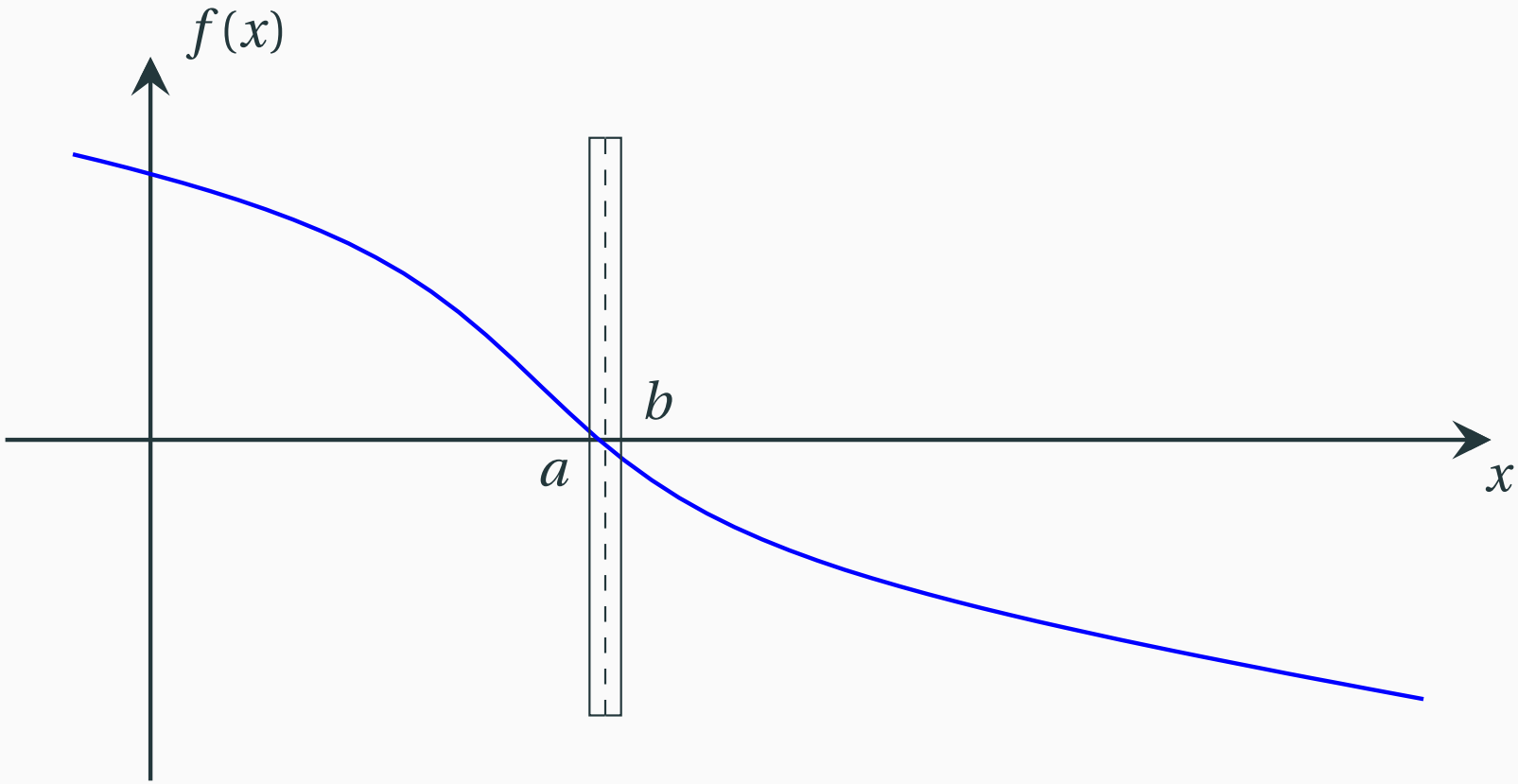
Recherche dichotomique



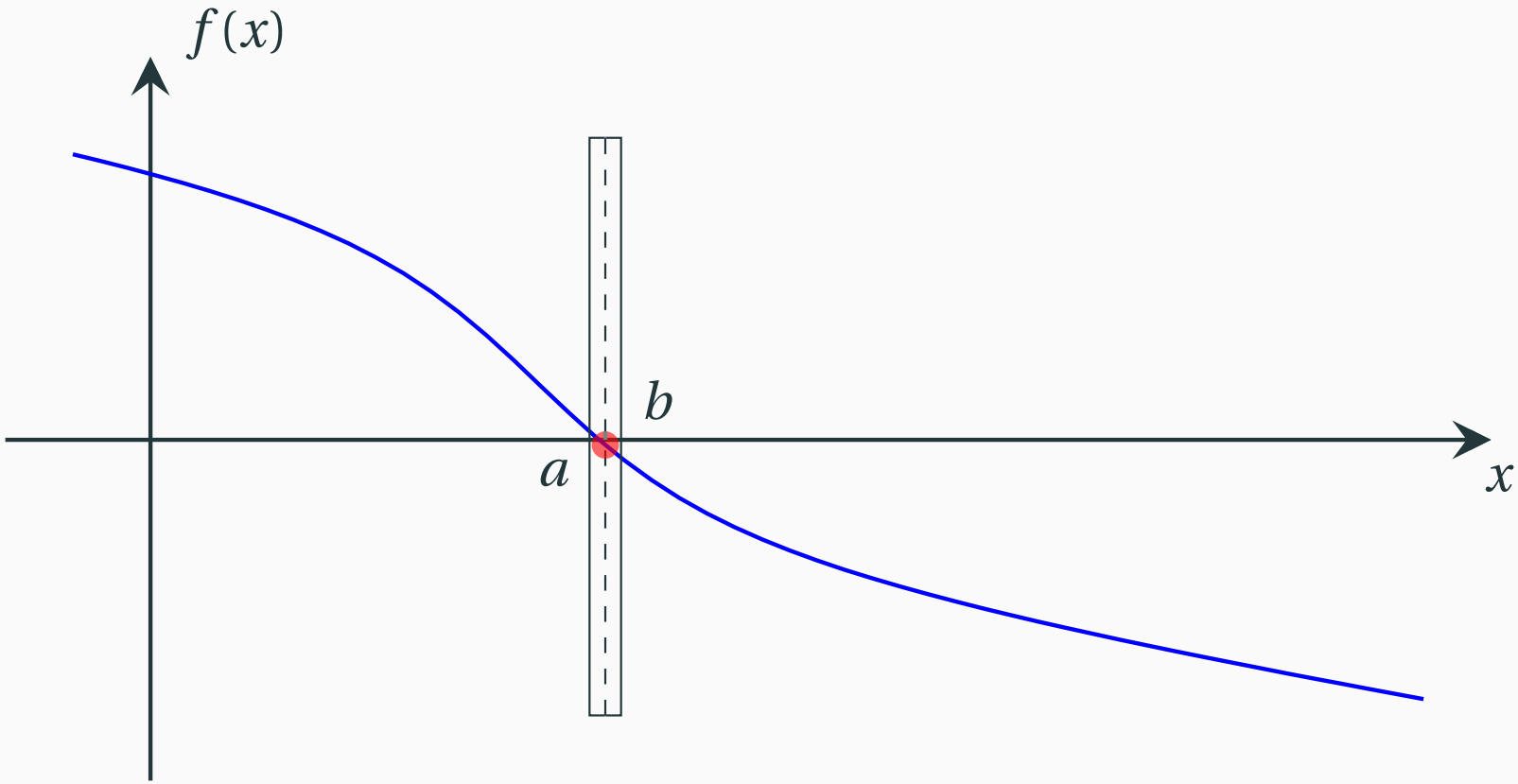
Recherche dichotomique



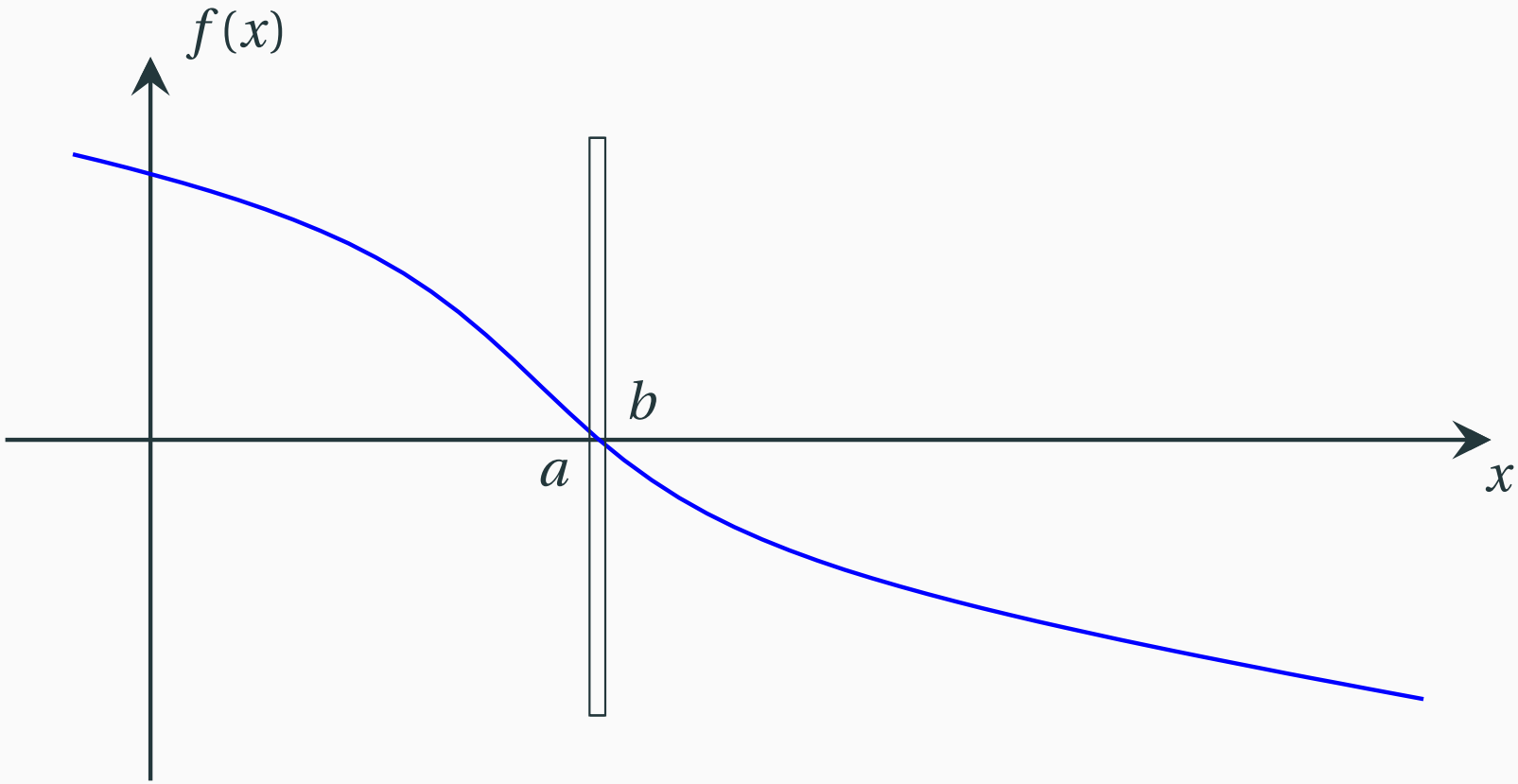
Recherche dichotomique



Recherche dichotomique



Recherche dichotomique



Recherche dichotomique

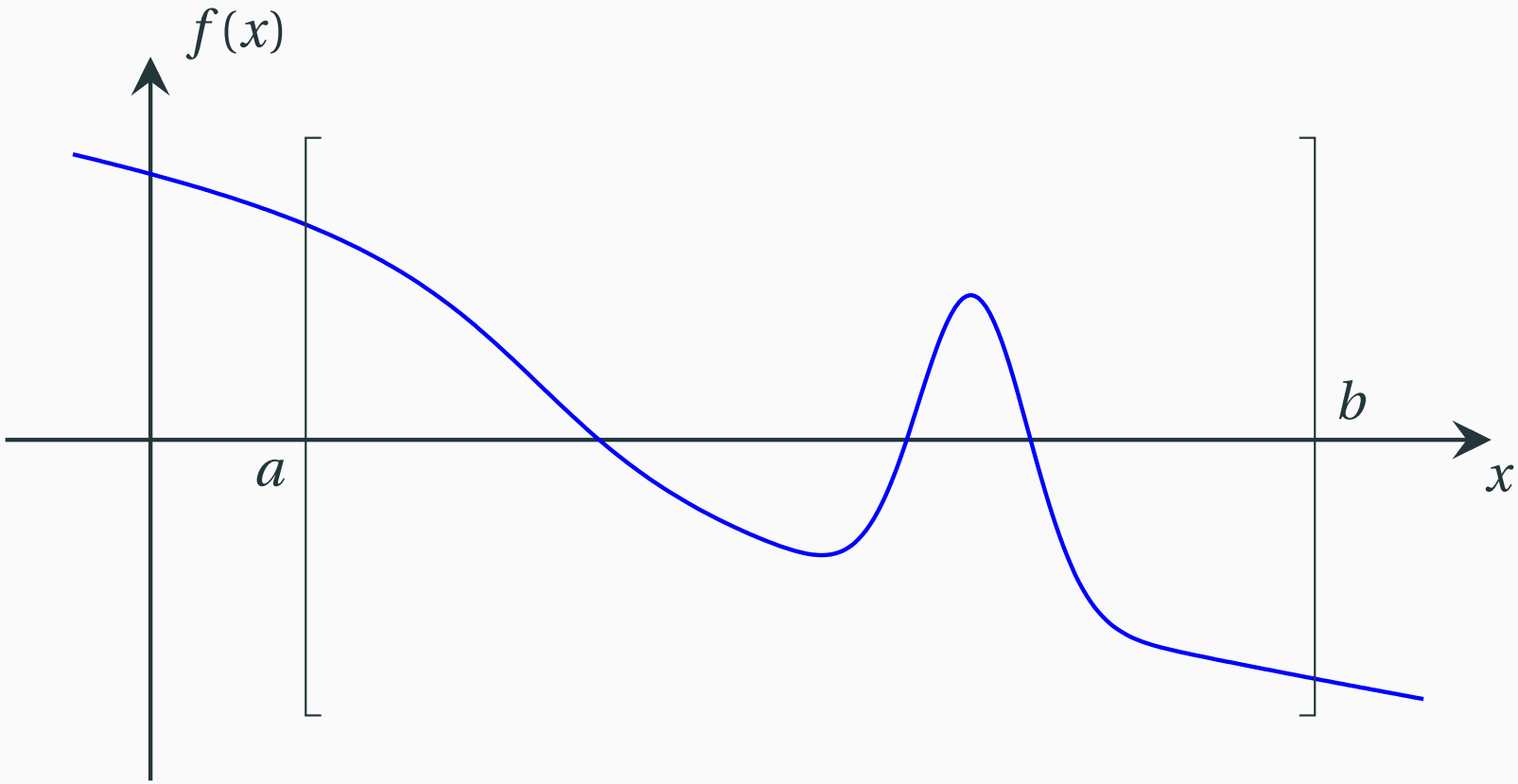
On conserve une moitié d'intervalle contenant à coup sûr un zéro

- $[a, m]$ si $f(a) \times f(m) \leqslant 0$
- $[m, b]$ si $f(a) \times f(m) > 0$ (donc nécessairement $f(m) \times f(b) \leqslant 0$)

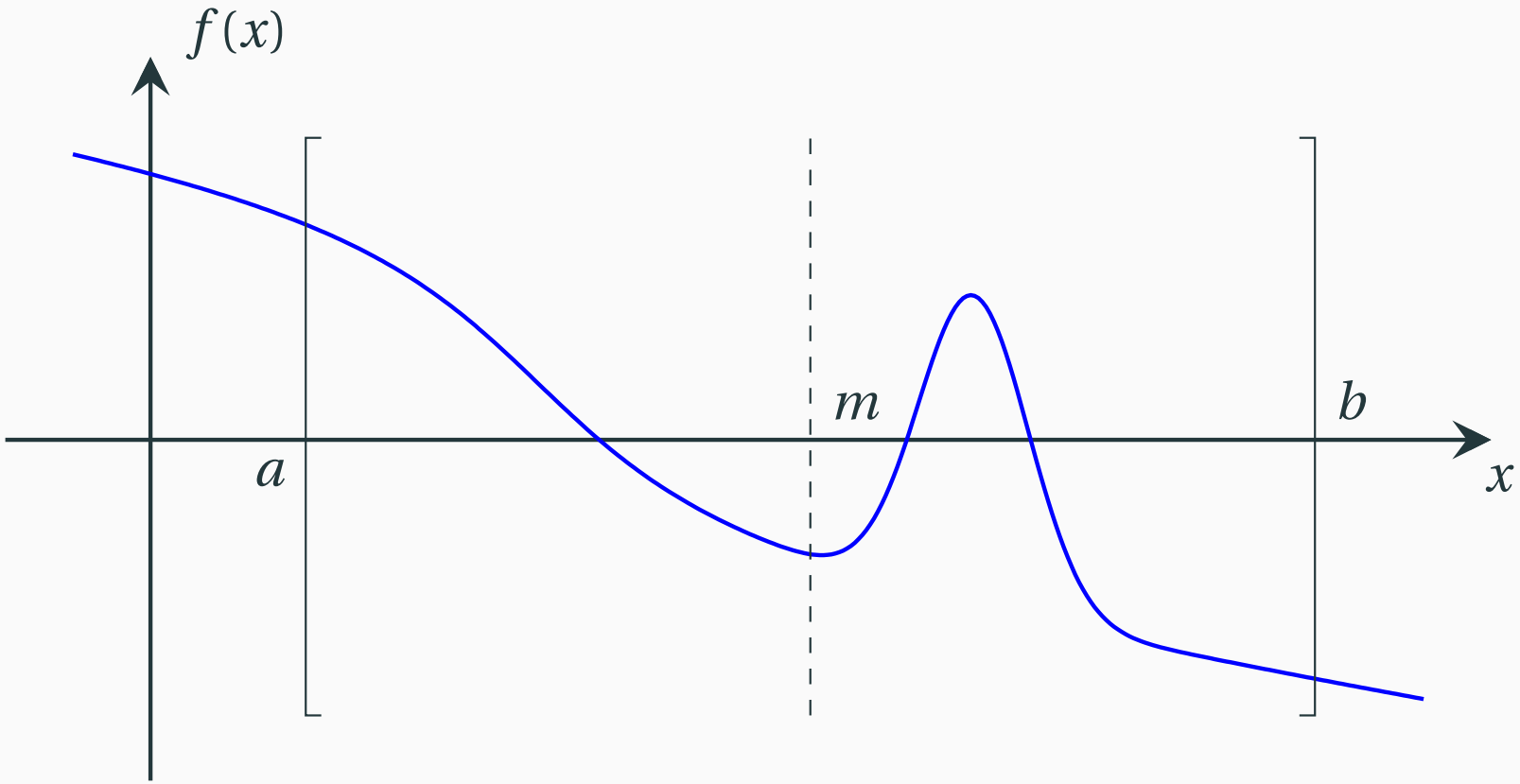
$f(a) \times f(b) \leqslant 0$ est un *invariant de boucle*

Il peut y avoir des zéros de part et d'autre de m !

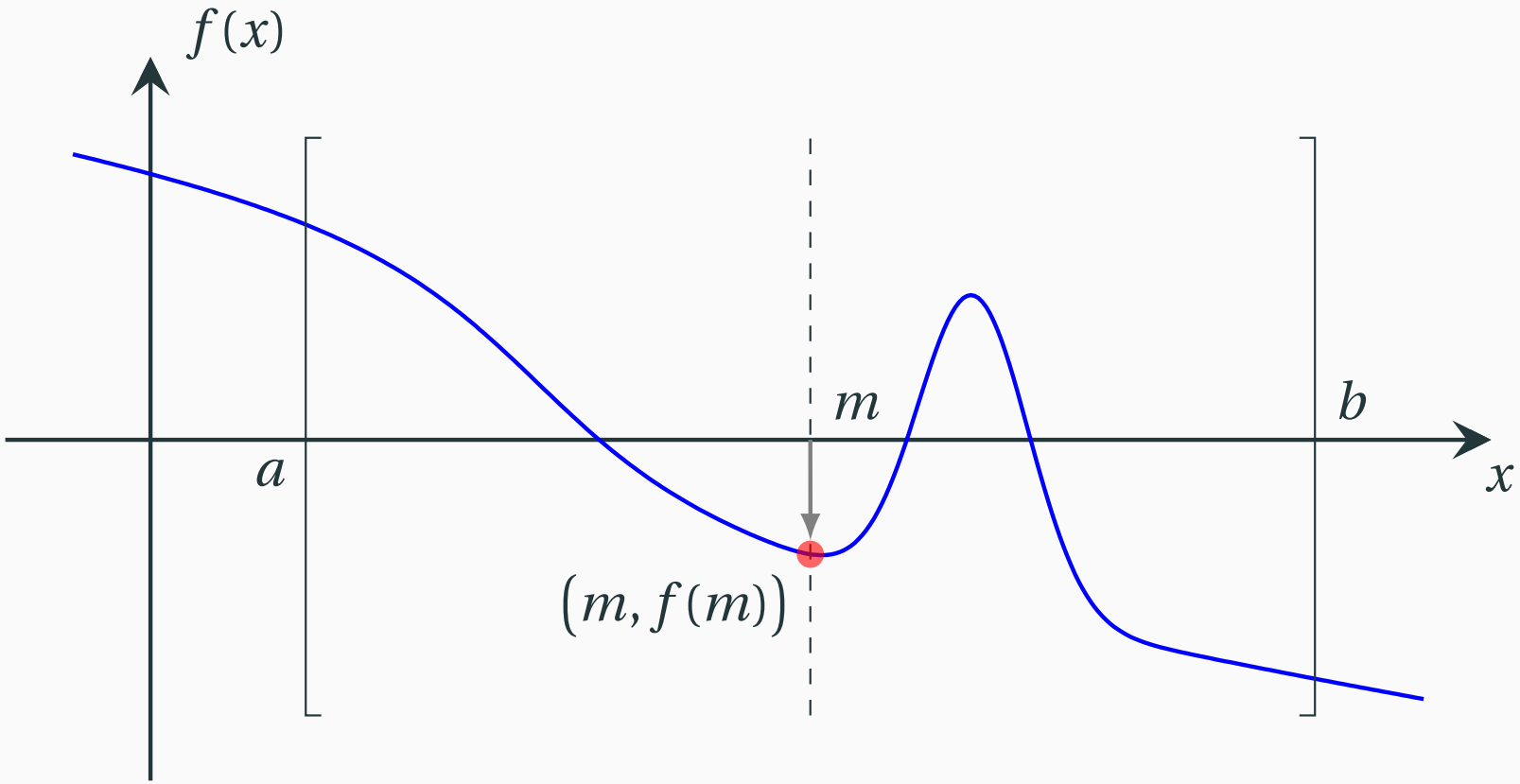
Recherche dichotomique



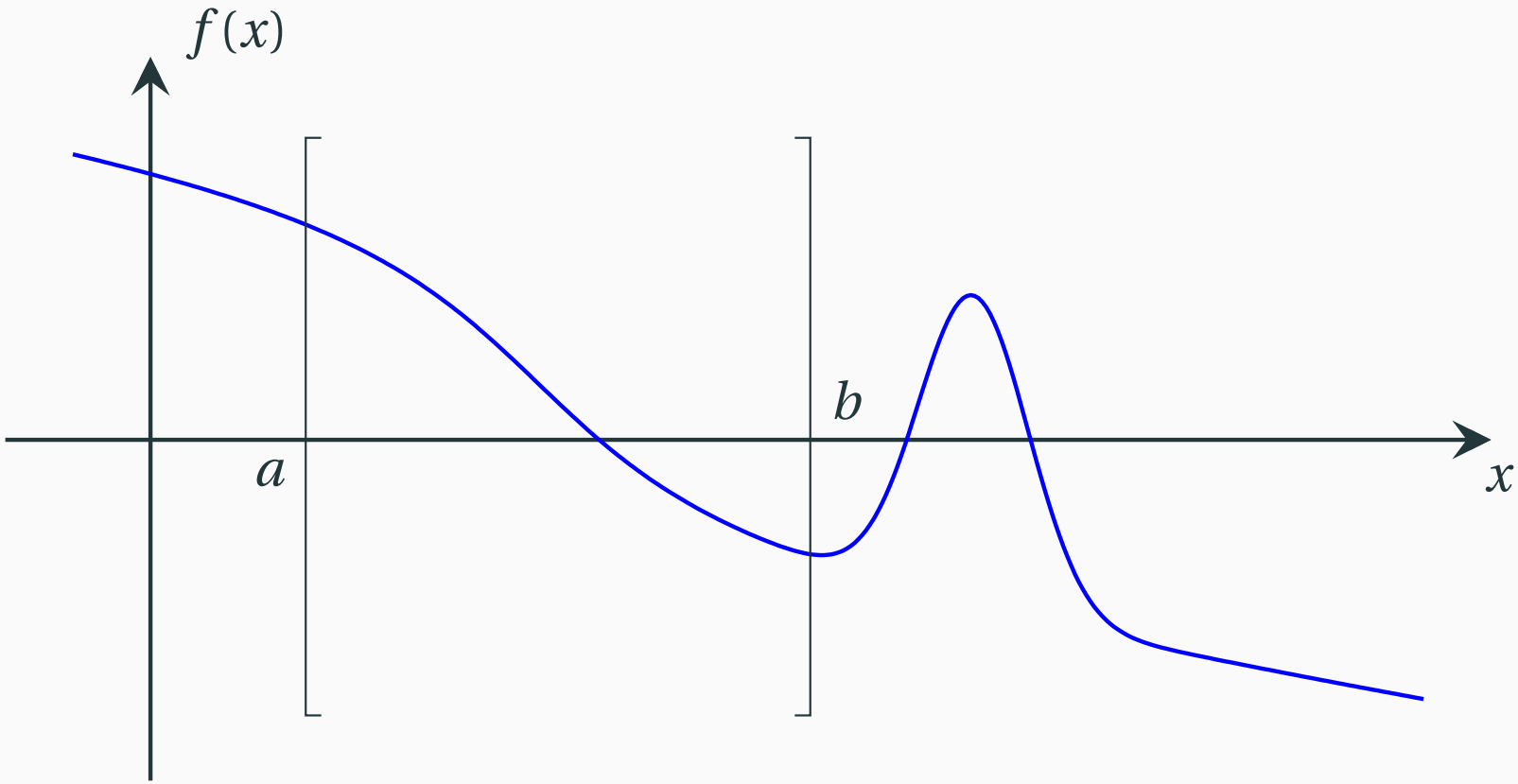
Recherche dichotomique



Recherche dichotomique



Recherche dichotomique



Recherche dichotomique

Si l'on veut une précision de ϵ, on s'arrête lorsque $|b-a| \leqslant 2\epsilon$

On est certain d'obtenir une approximation d'un zéro à ϵ près

Après n itérations, la largeur de l'encadrement est $\frac{|b-a|}{2^n}$

Il faut donc $n = \left\lceil \log_2\left(\frac{b-a}{\epsilon}\right)\right\rceil - 1$ itérations

Recherche dichotomique

```python
def RechercheDichotomique(f, a, b, eps) :
    if f(a)*f(b) > 0:
        raise ValueError("Mauvais encadrement")

    while abs(b-a) > 2*eps:
        m = (a+b)/2
        if f(a)*f(m) <= 0:
            b = m
        else:
            a = m

    return (a+b)/2
```

Recherche dichotomique

Plutôt qu'un test pour vérifier le changement de signe, on peut utiliser une assertion

```
...
assert( f(a)*f(b) ) <= 0
...
```

Possiblement ignoré à l'exécution (mode *release*)

Un exemple concret

Dosage de CH_3COOH par OH^-

$$AH + H_20 \leftrightarrow A^- + H_3O^+ \qquad K_a$$

et

$$2H_20 \leftrightarrow H_3O^+ + HO^- \qquad K_e$$

$$x^3 + (c_2 v + K_a (V_0 + v) c_0) x^2 - K_a K_e (V_0 + v)^3 c_0^3$$

$$+ \left(K_a (V_0 + v) c_0 c_2 v - K_e (V_0 + v)^2 c_0^2 - K_a (V_0 + v) c_0 c_1 V_0\right) x = 0$$

où $x = [H_3O^+] = 10^{-pH}$

Un exemple concret

```
c0 = 1.0  # mol/L
c1 = 0.05 # concentration acide, mol/L
c2 = 1.0  # concentration soude, mol/L
V0 = 0.2  # L

v = 0.01  # soude versée, L

Ka = 10**(-4.8)
Ke = 10**(-14)

def f(pH):
    x = 10**-pH
    return (x**3 + (c2 * v + Ka * (V0 + v) * c0) * x**2.0
            - Ka * Ke * (V0 + v)**3.0 * c0**3.0
            + (Ka * (V0 + v) * c0 * c2 * v - Ke * (V0 + v)**2.0 * c0**2.0
            - Ka * (V0 + v) * c0 * c1 * V0) * x)
```

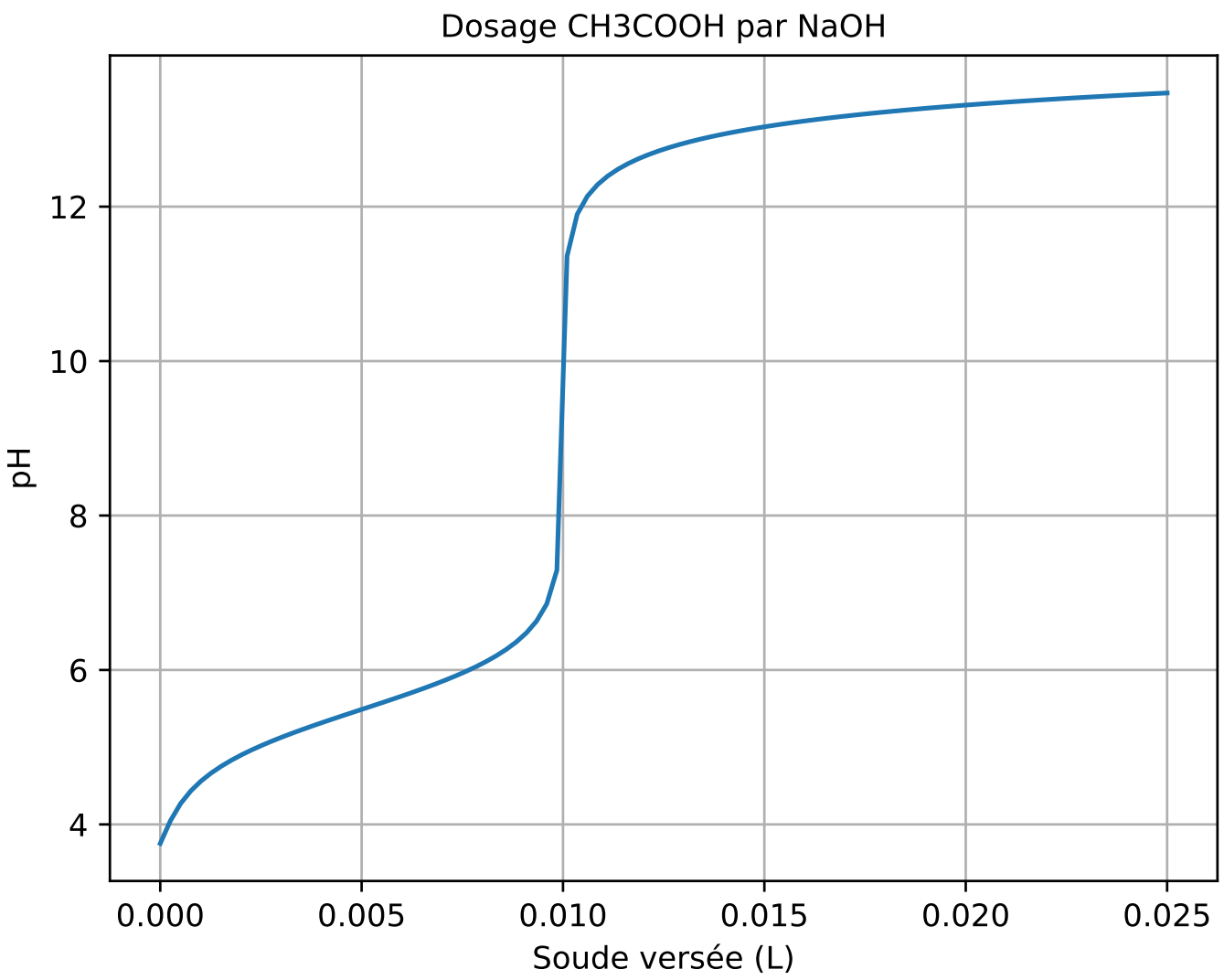
Un exemple concret

```
In []: import scipy.optimize as opt

In []: f(0.0)
Out[]: 1.0100033282757037

In []: f(14.0)
Out[]: -1.4677695889479051e-21

In []: opt.bisect(f, 0.0, 14.0)
Out[]: 9.416718335314954
```

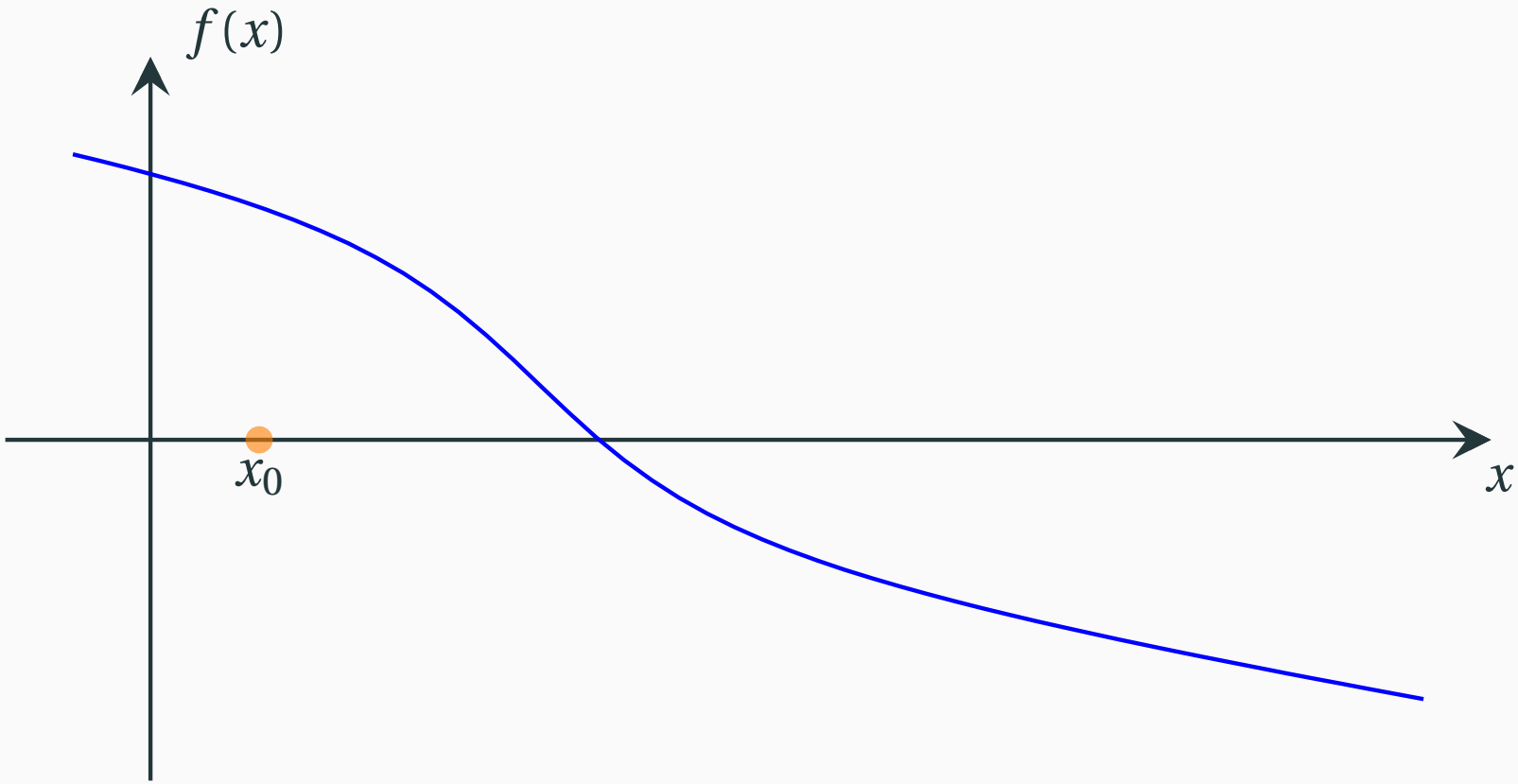
Un exemple concret

```
V = np.linspace(0.0, 0.025, 100)
PH = []
for v in V:
    PH.append(opt.bisect(f, 0.0, 14.0))

plt.plot(V, PH)
```

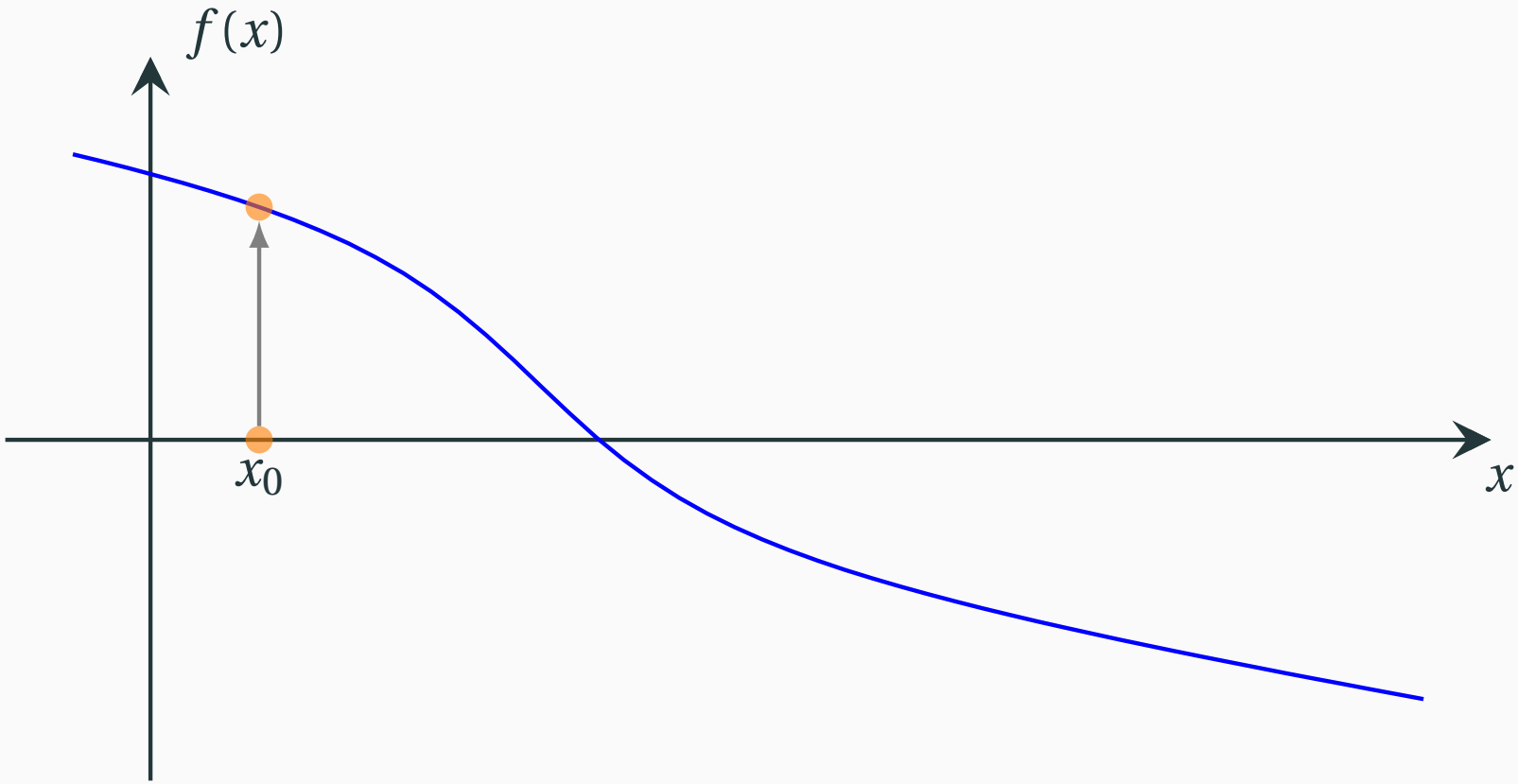
Un exemple concret



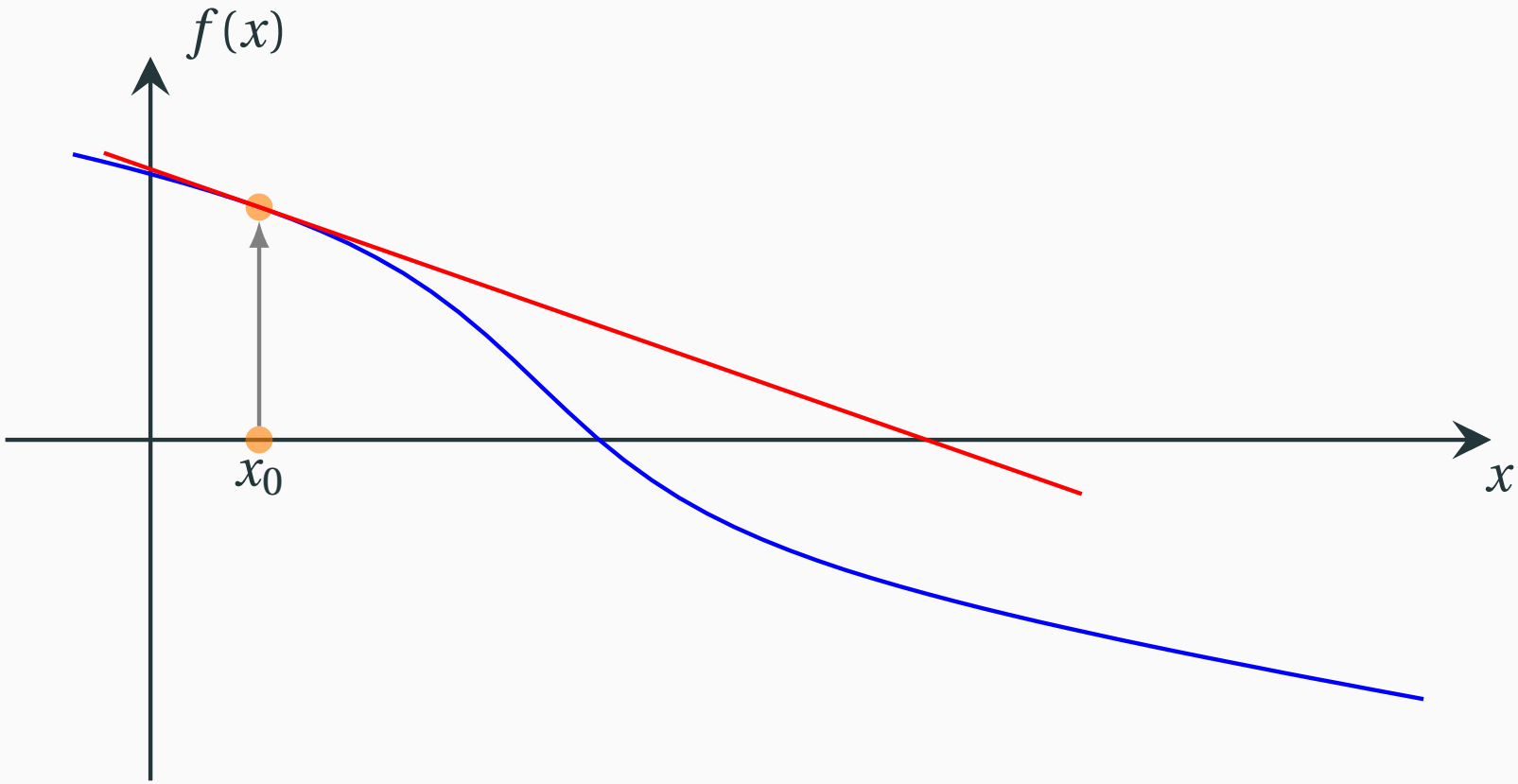
Méthode de Newton-Raphson



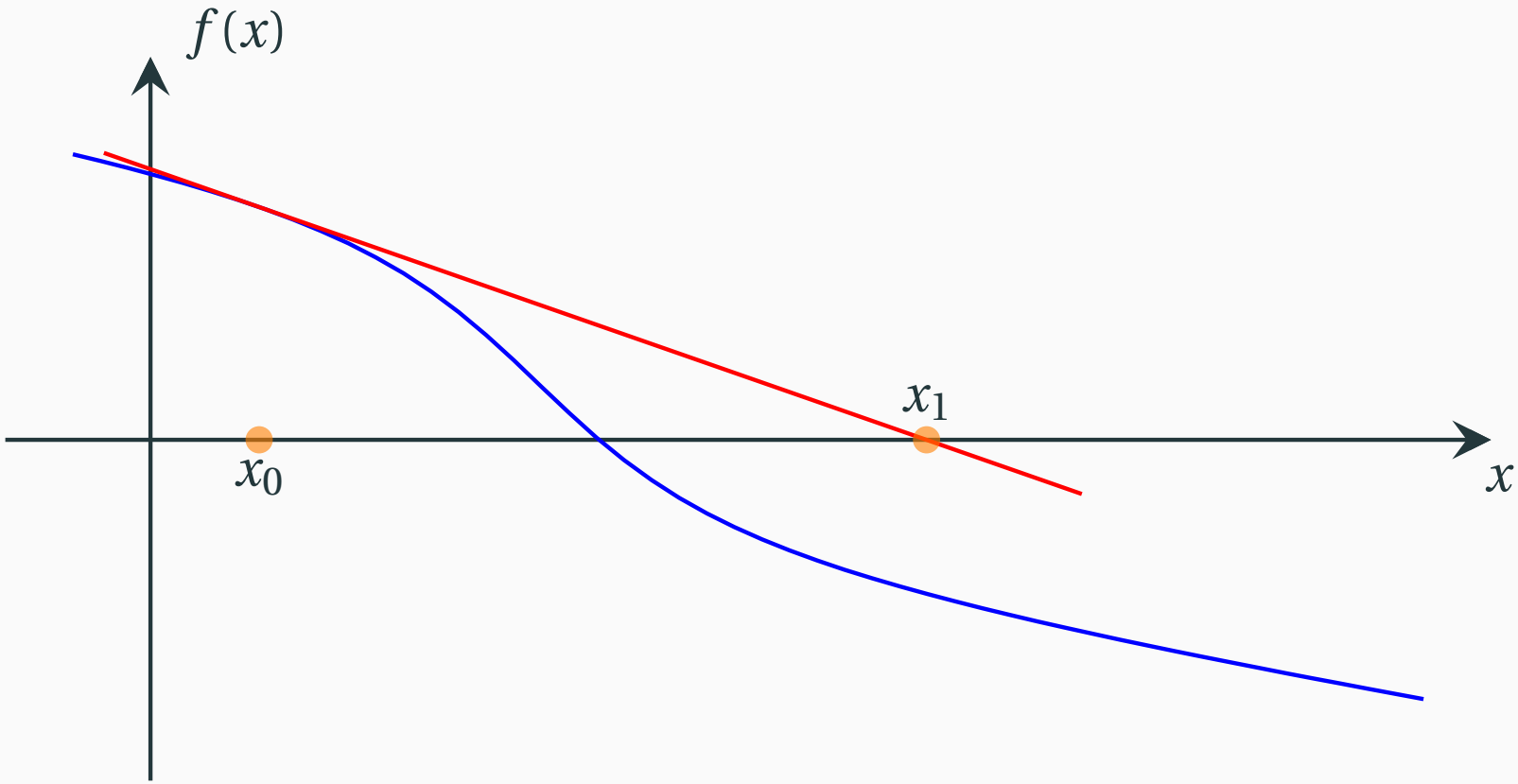
Méthode de Newton-Raphson



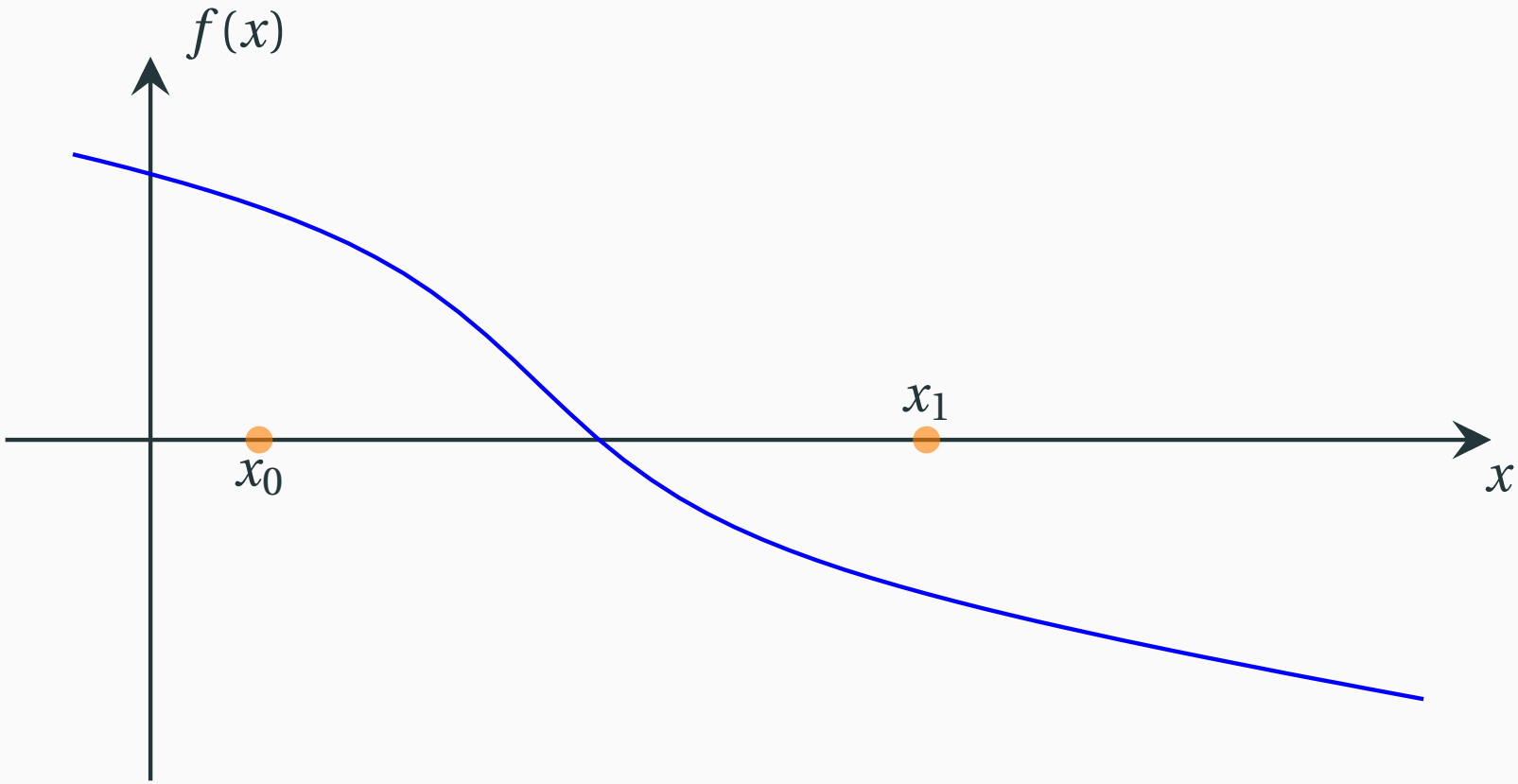
Méthode de Newton-Raphson



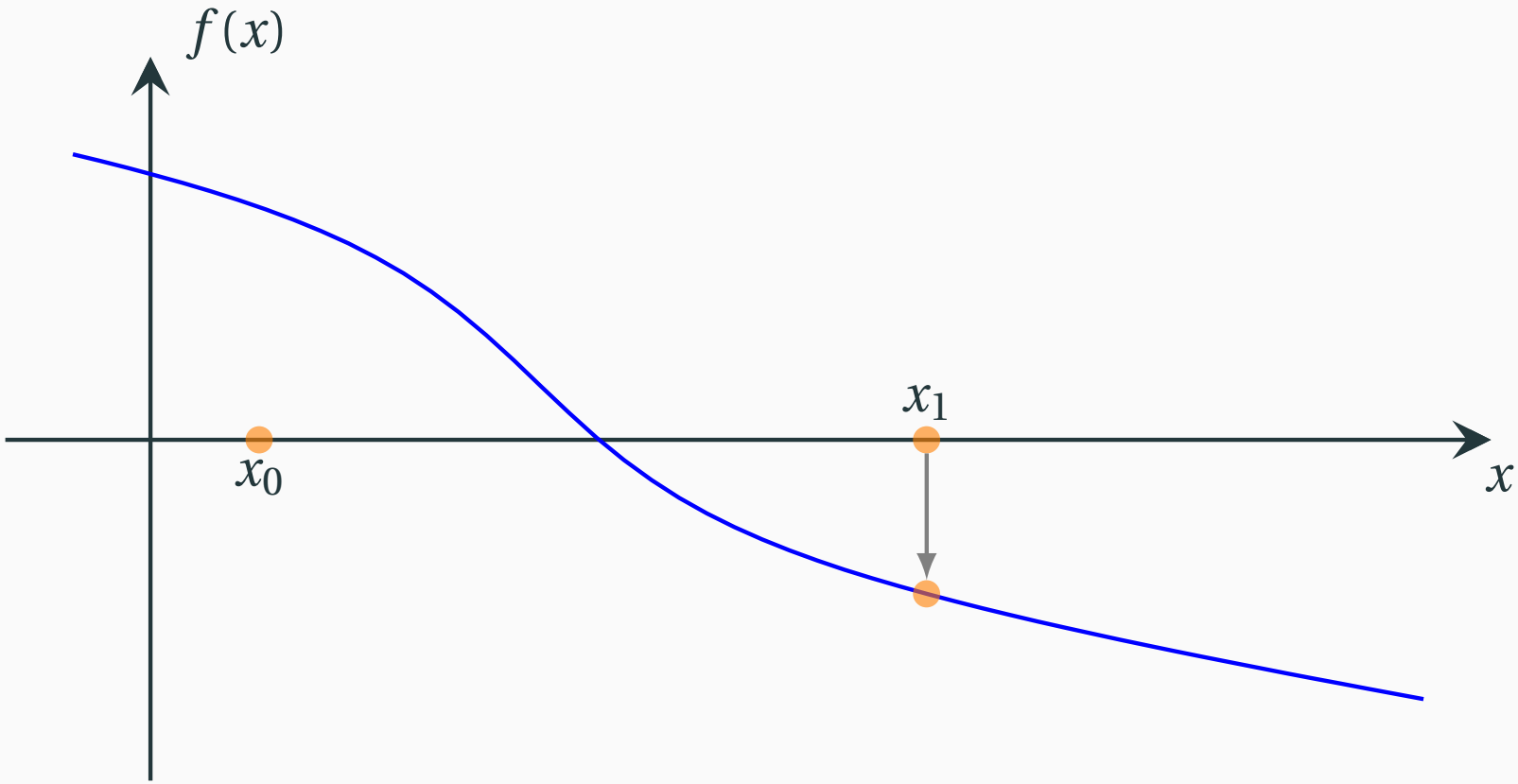
Méthode de Newton-Raphson



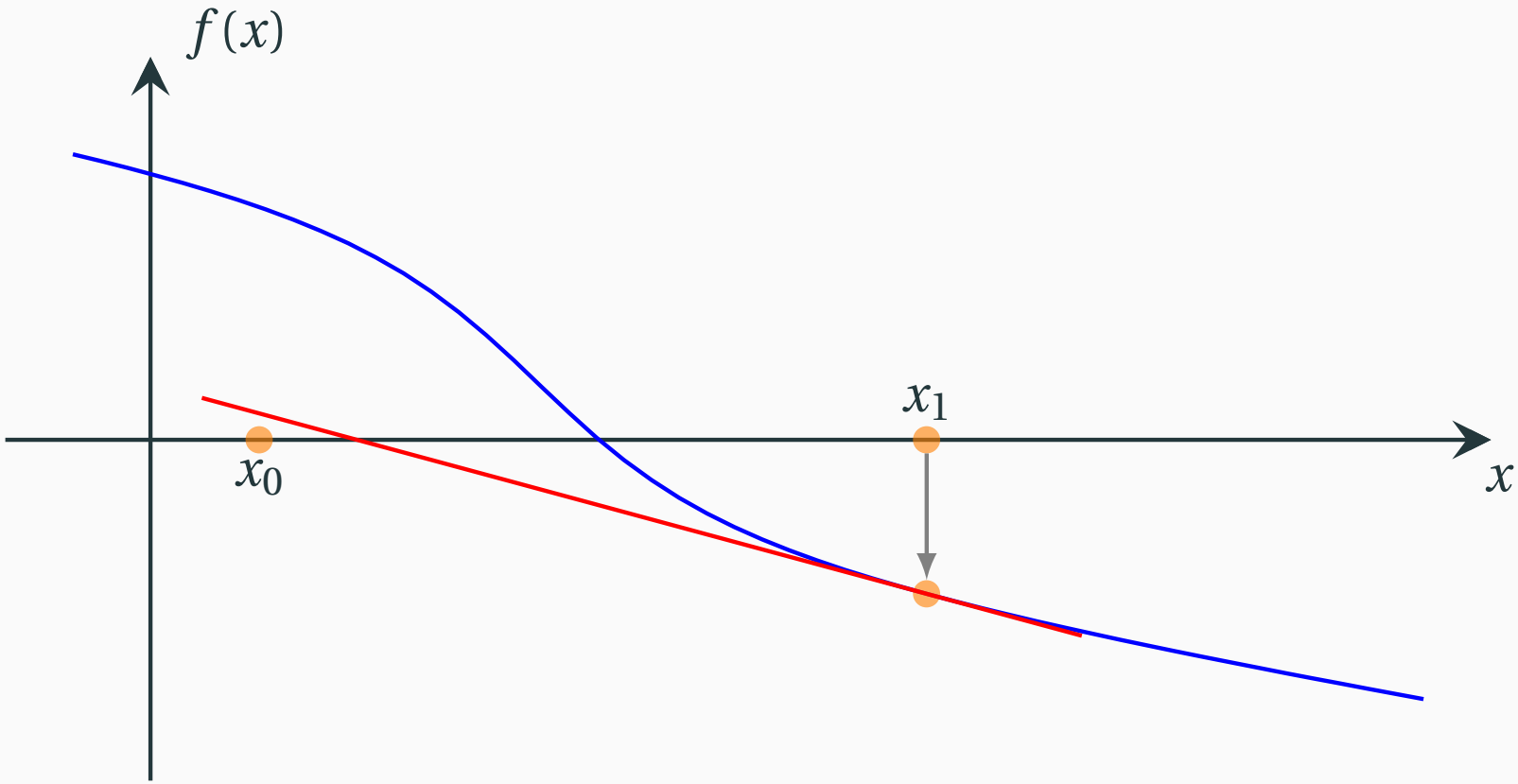
Méthode de Newton-Raphson



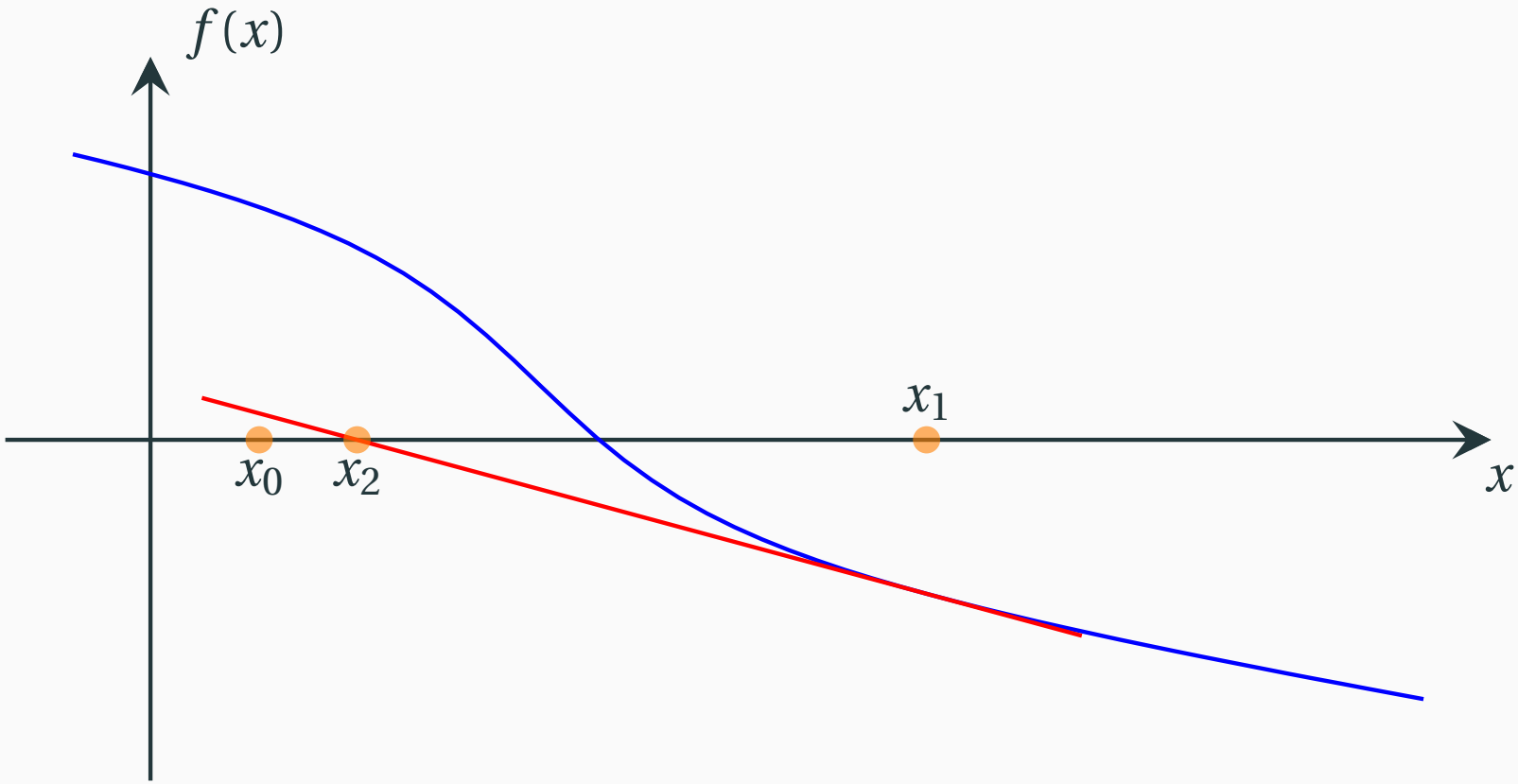
Méthode de Newton-Raphson



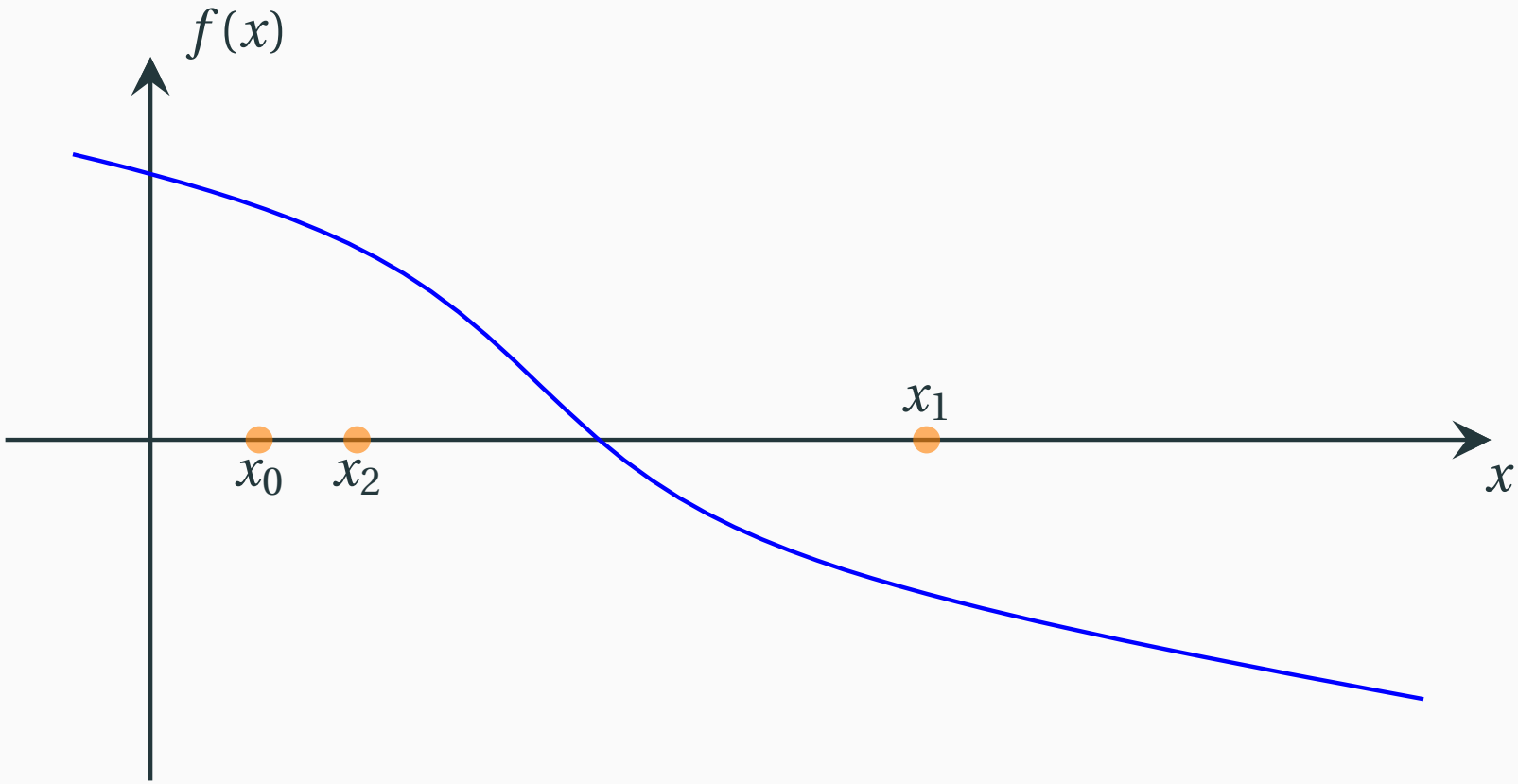
Méthode de Newton-Raphson



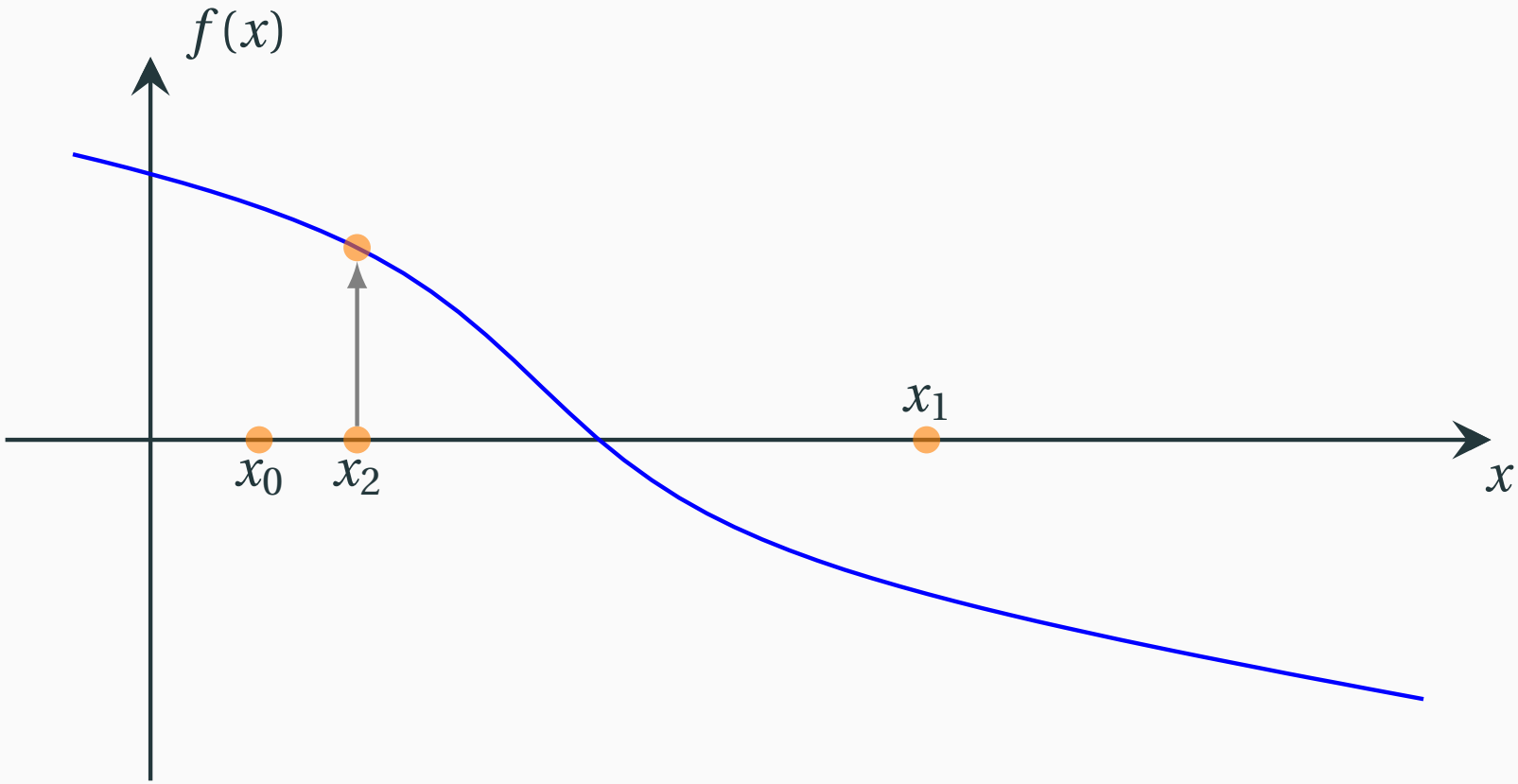
Méthode de Newton-Raphson



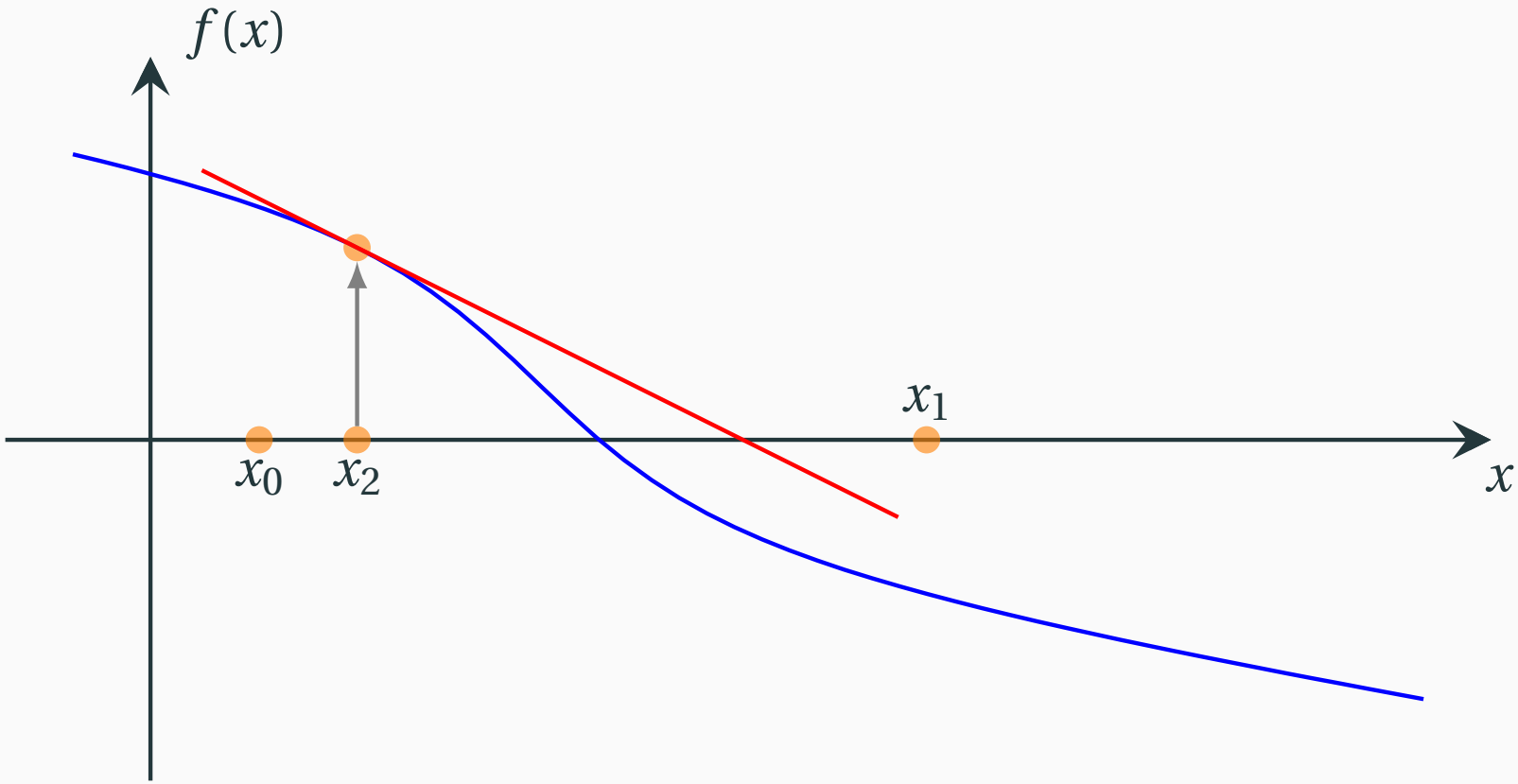
Méthode de Newton-Raphson



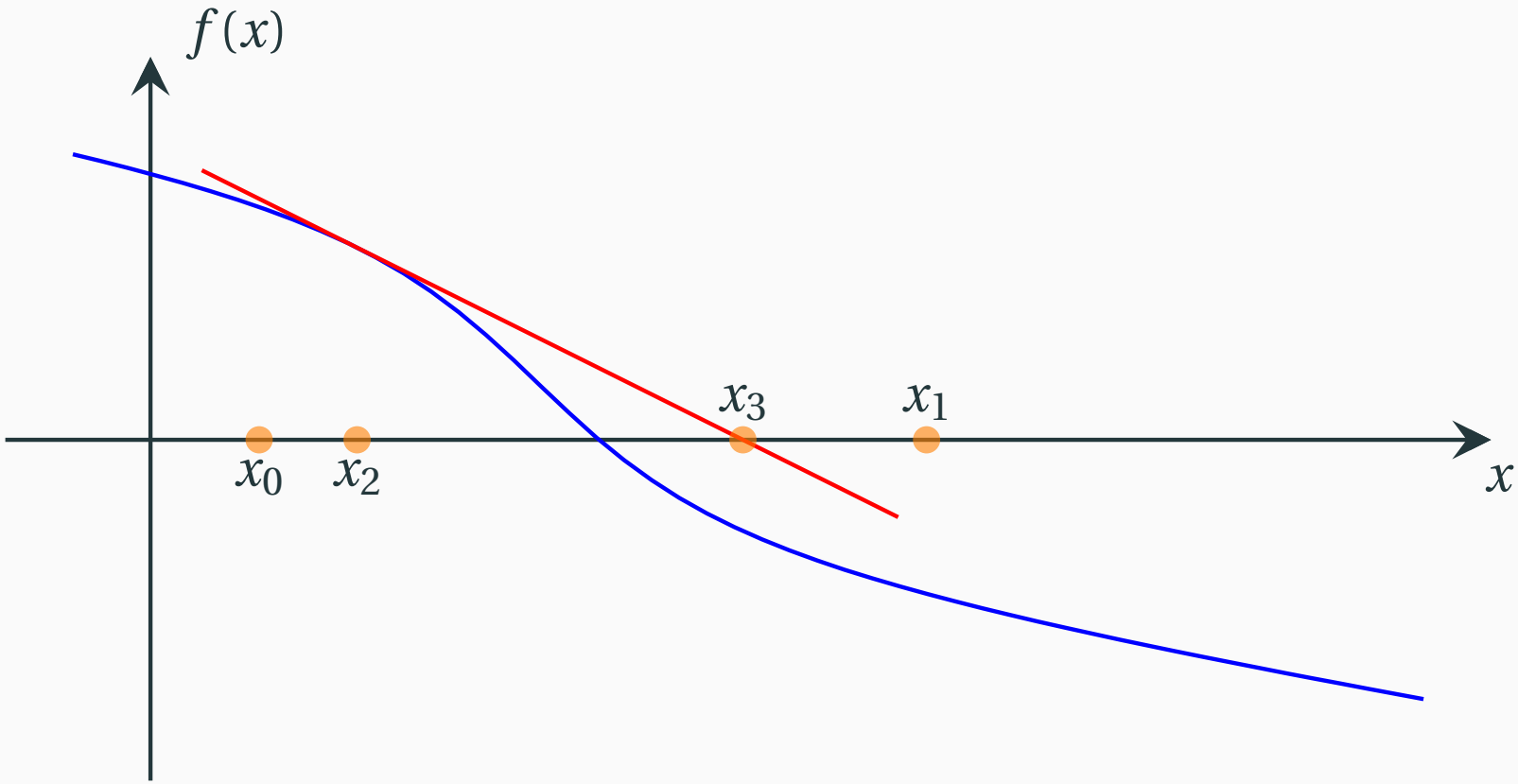
Méthode de Newton-Raphson



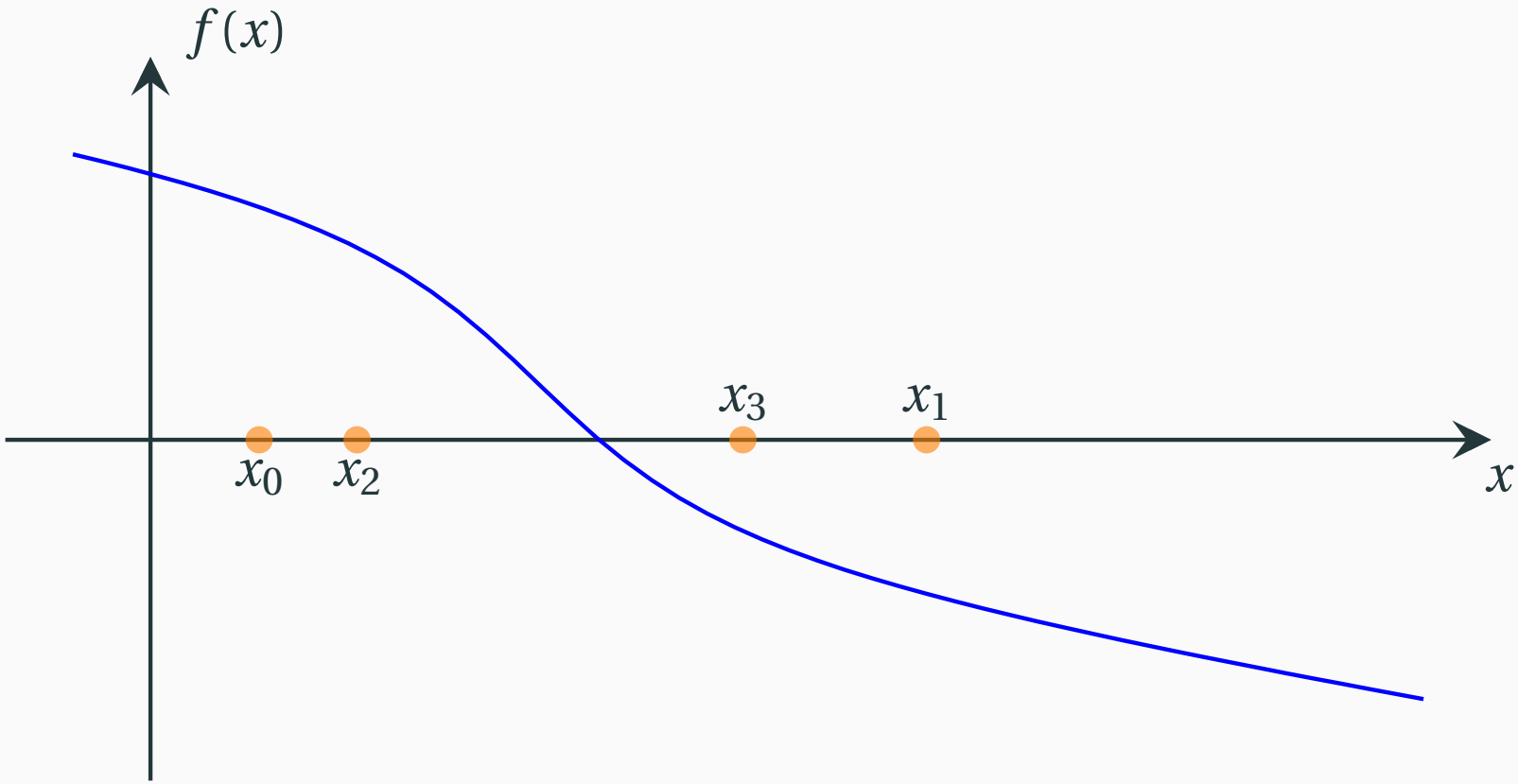
Méthode de Newton-Raphson



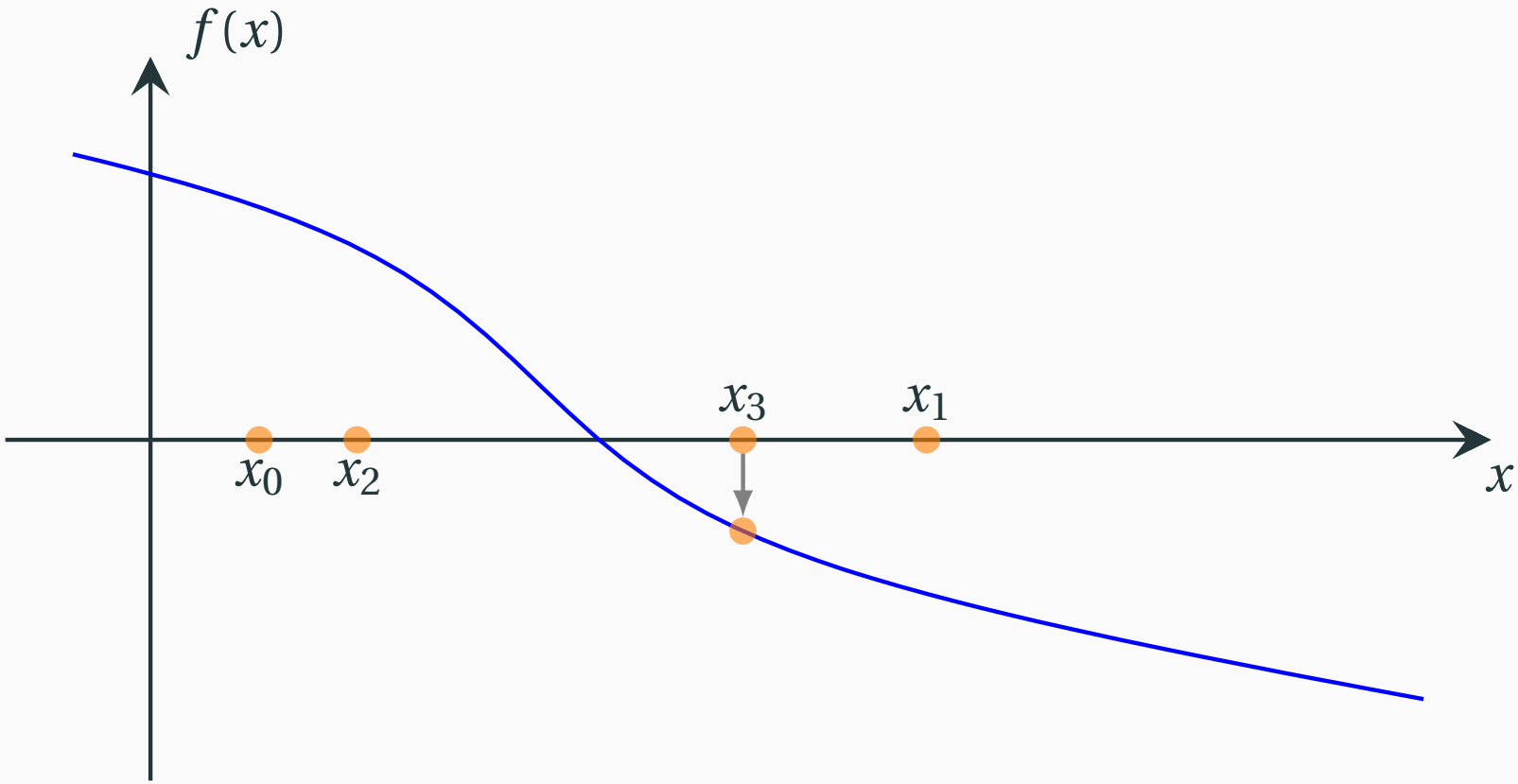
Méthode de Newton-Raphson



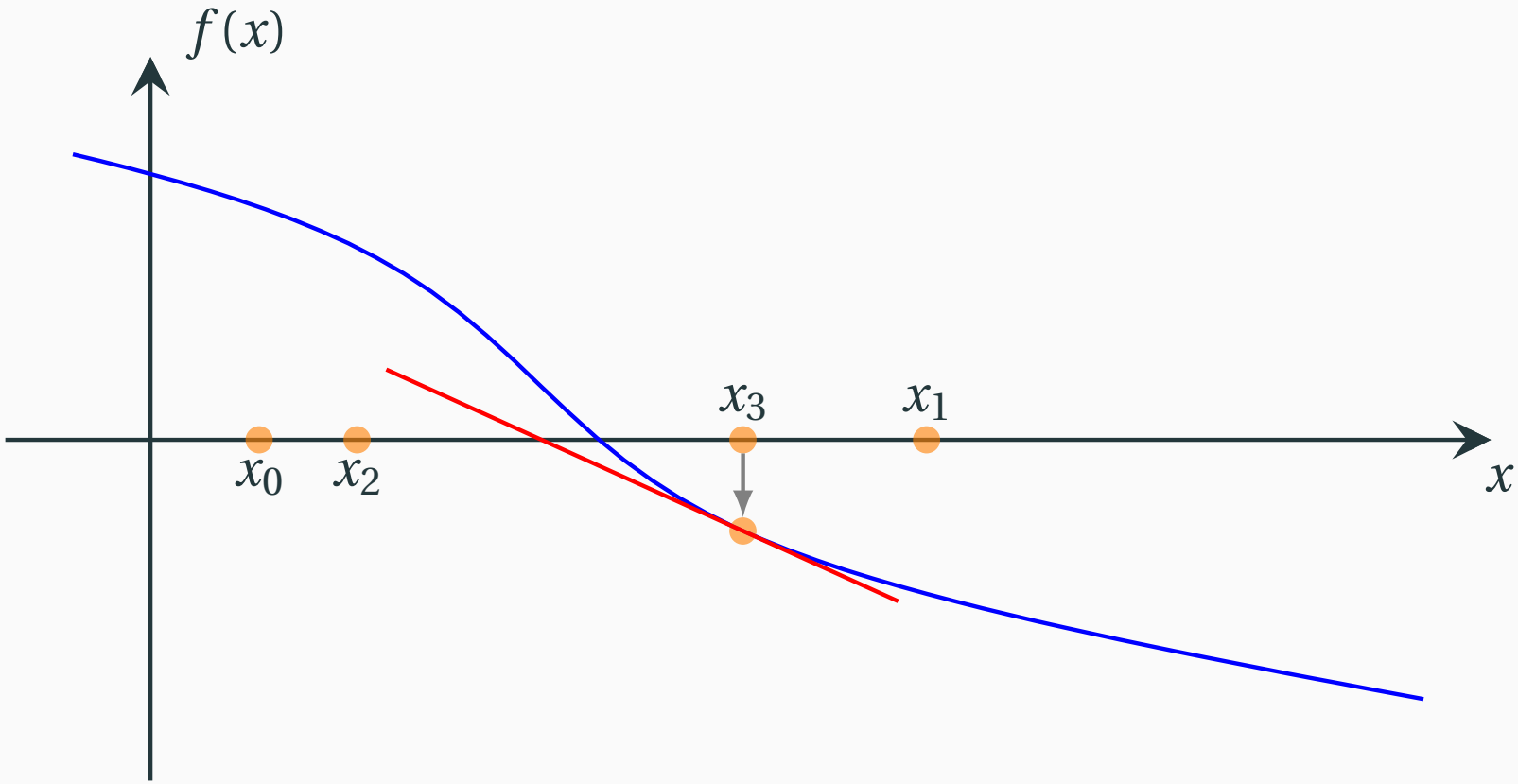
Méthode de Newton-Raphson



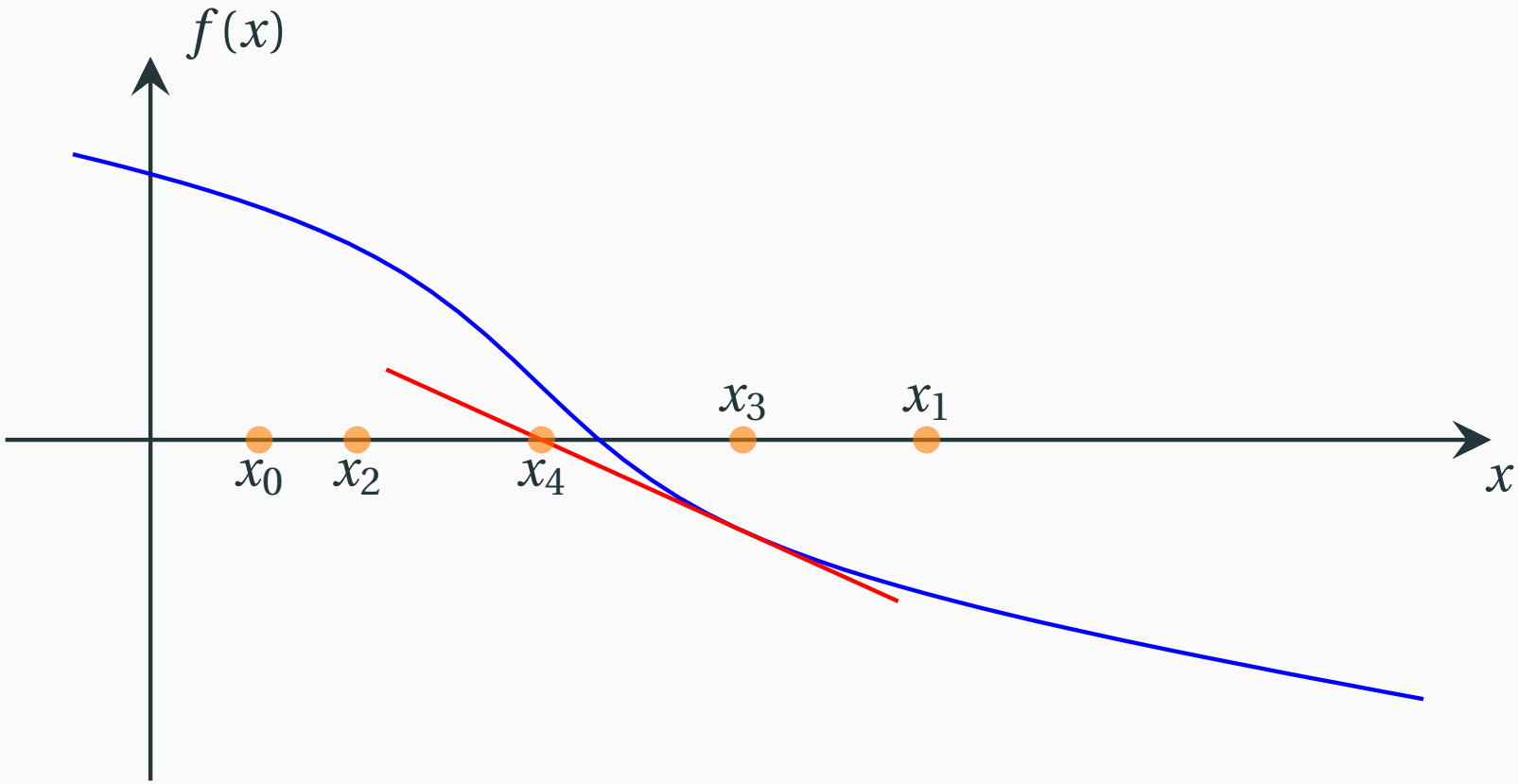
Méthode de Newton-Raphson



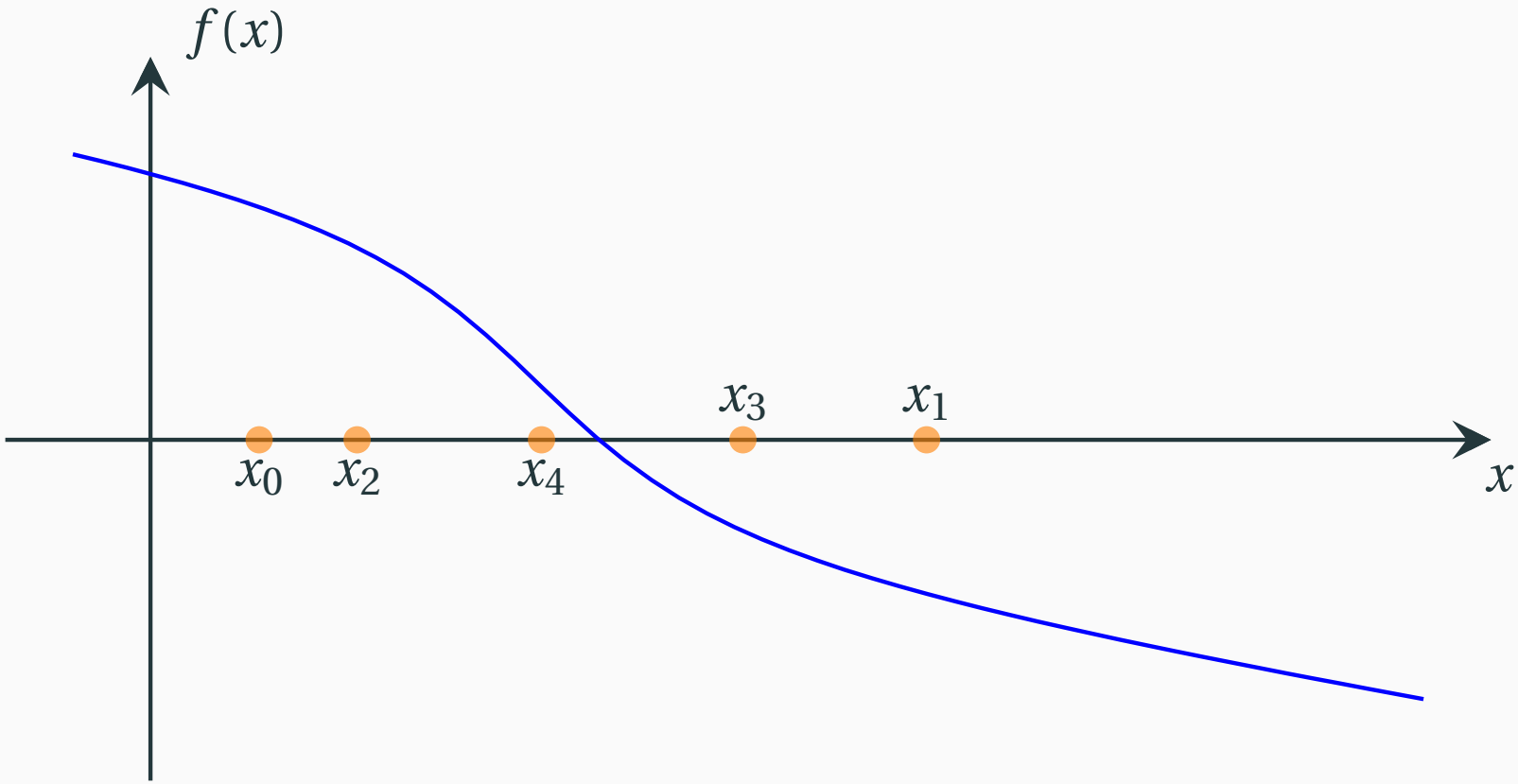
Méthode de Newton-Raphson



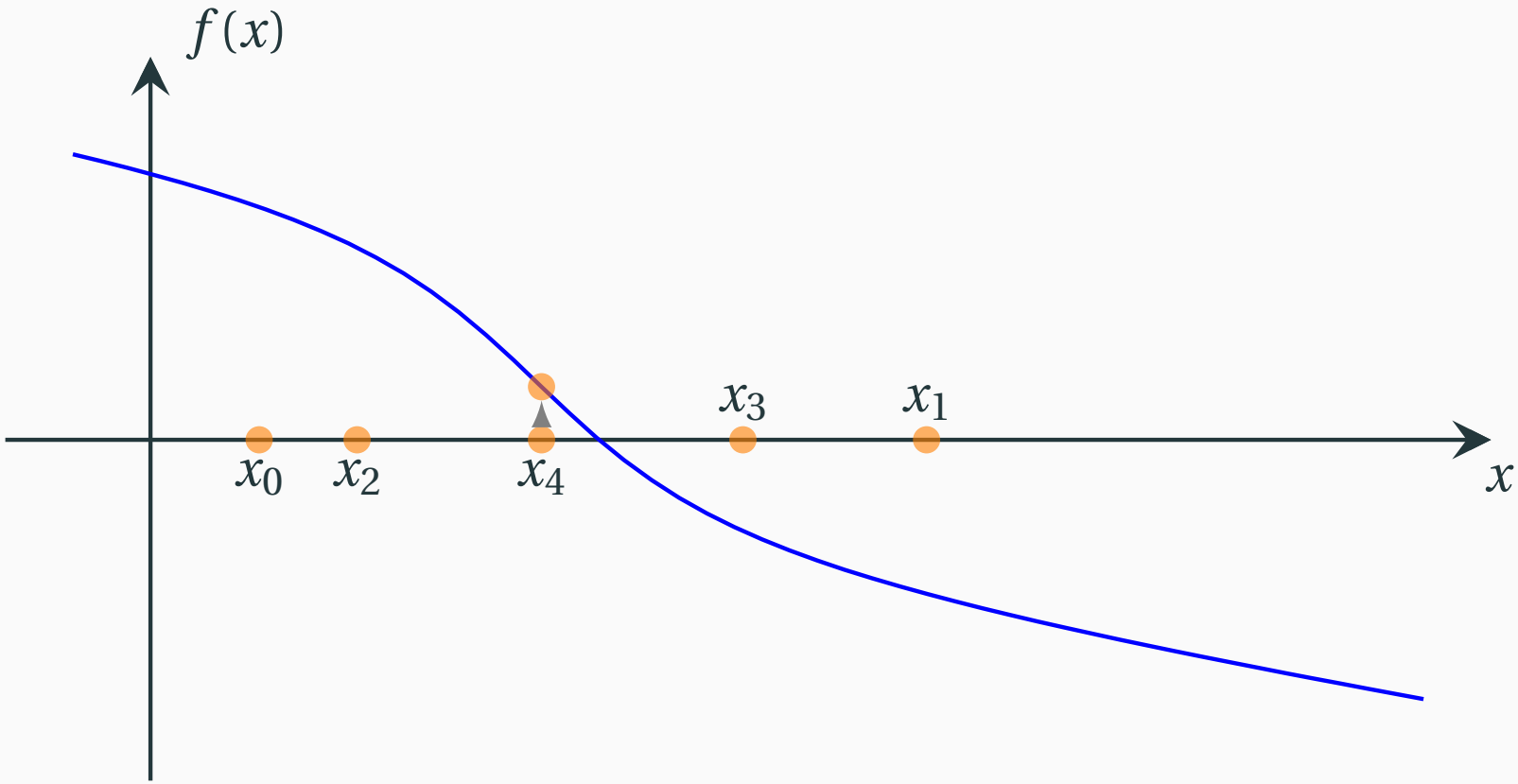
Méthode de Newton-Raphson



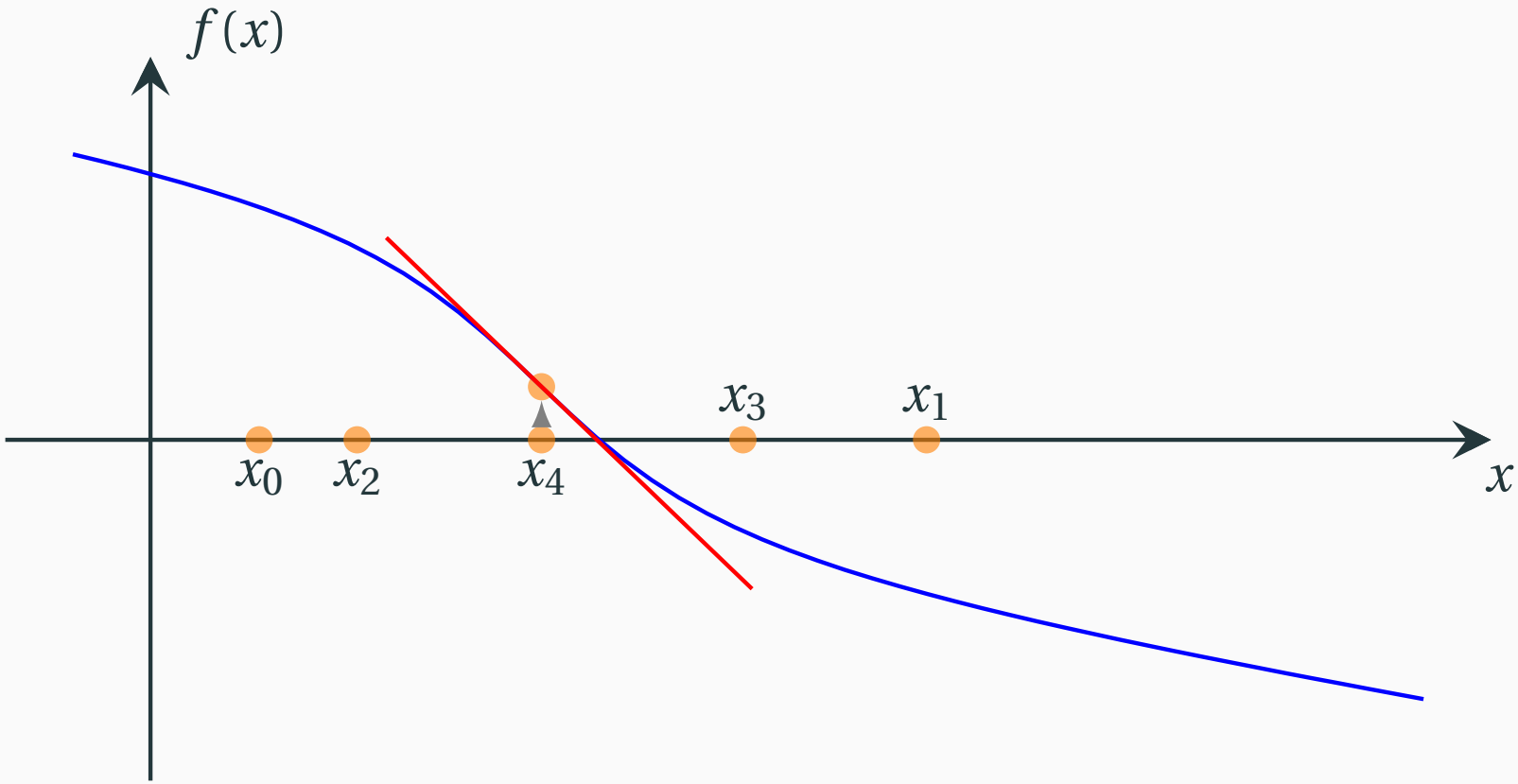
Méthode de Newton-Raphson



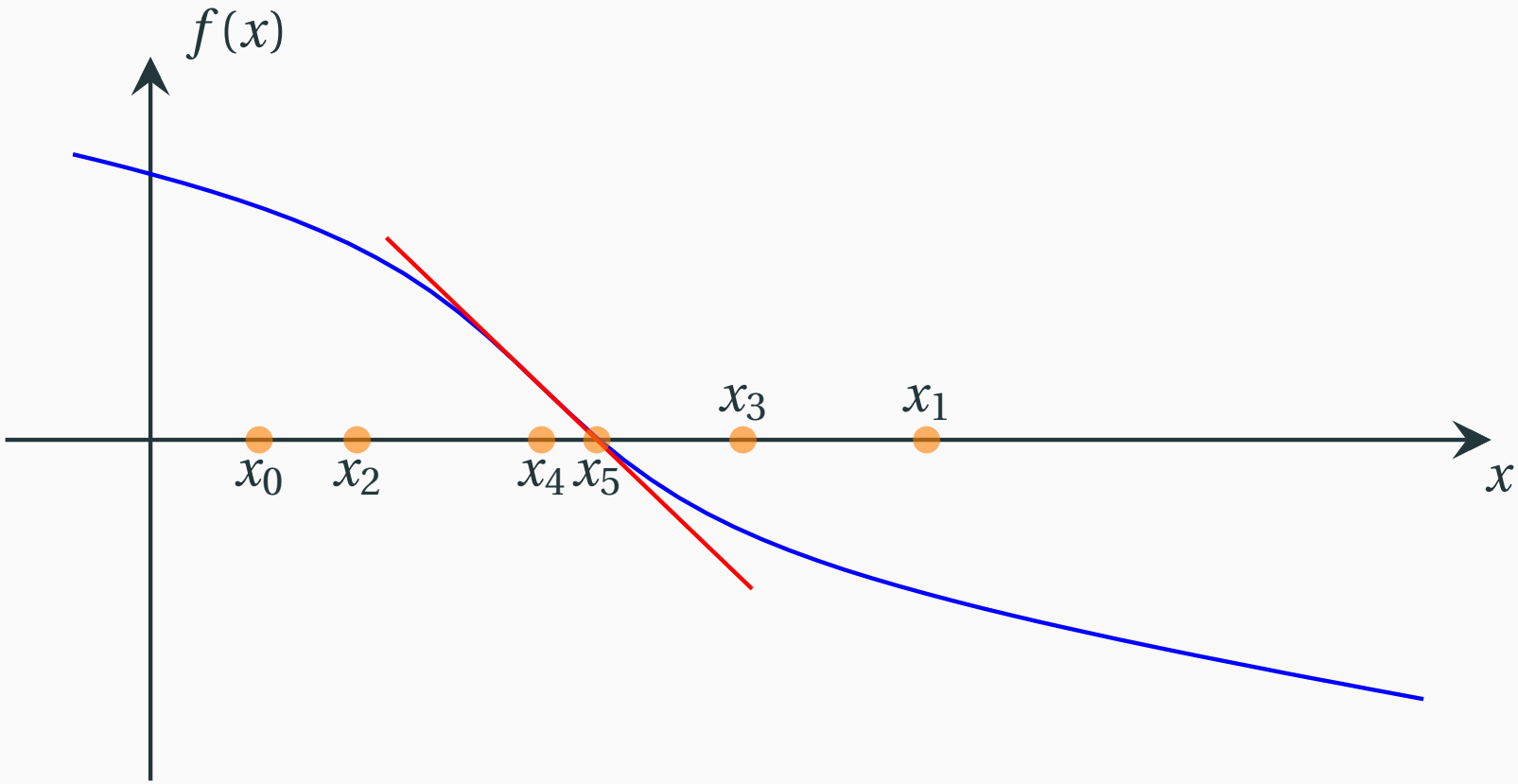
Méthode de Newton-Raphson



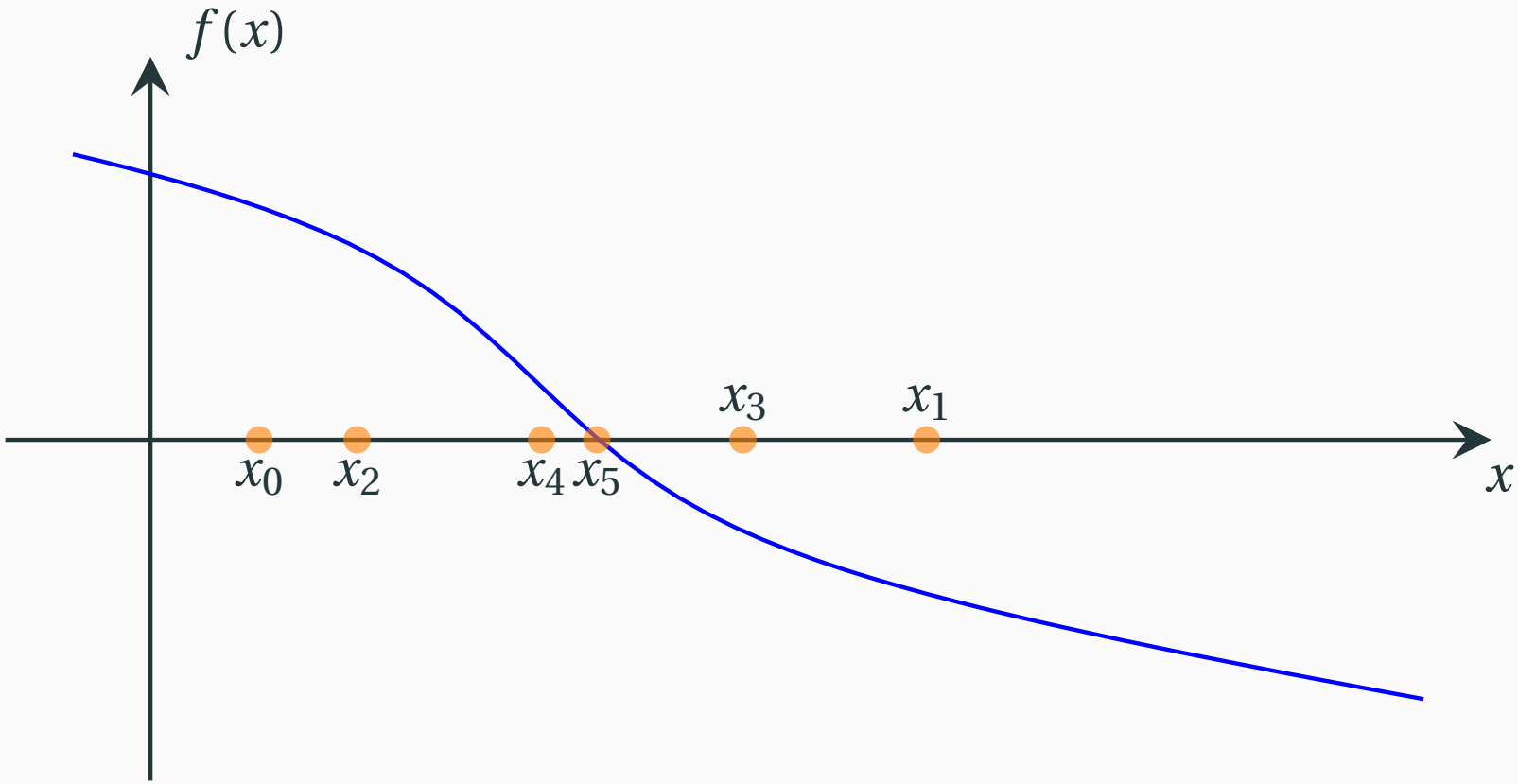
Méthode de Newton-Raphson



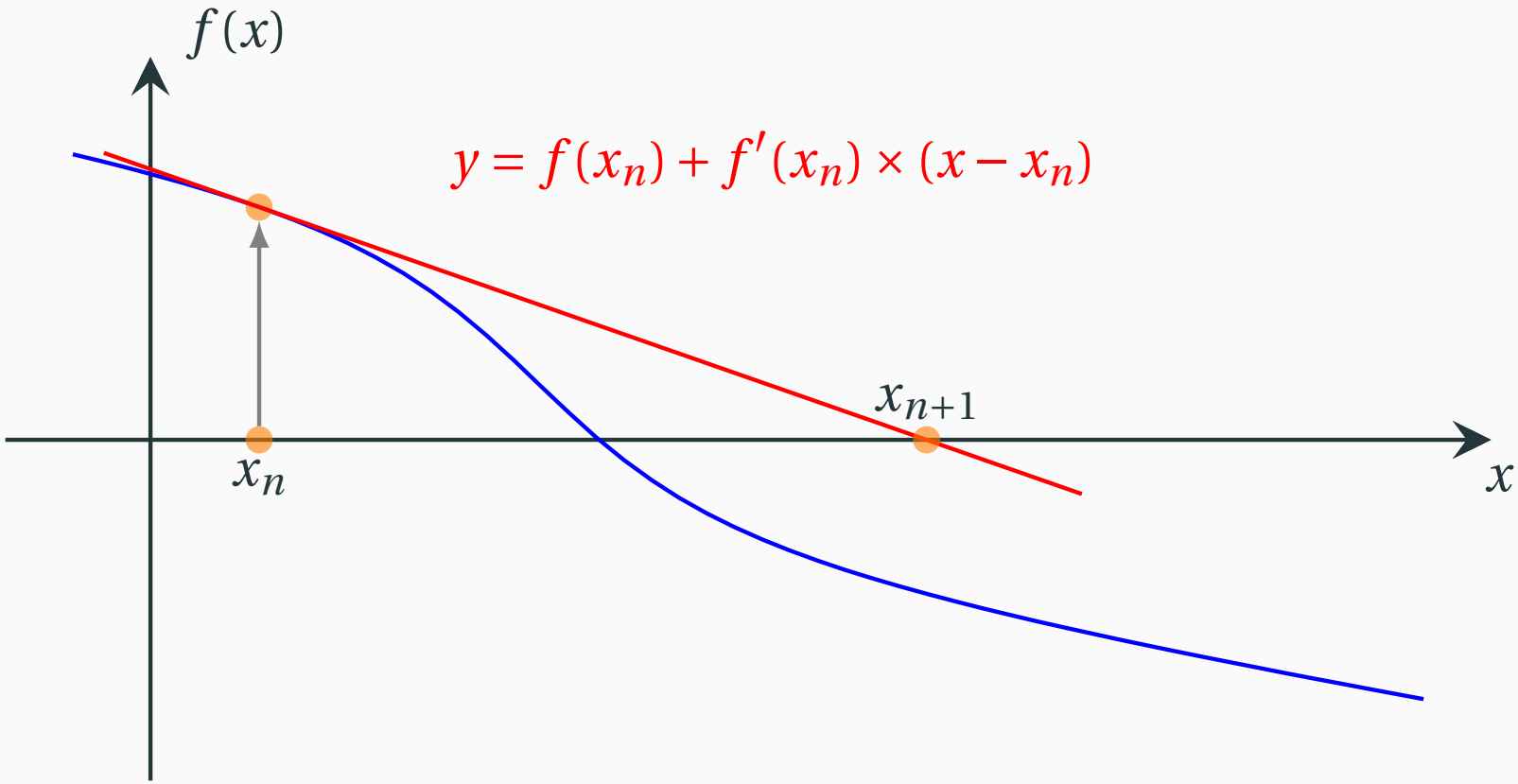
Méthode de Newton-Raphson



Méthode de Newton-Raphson



Méthode de Newton-Raphson



Implémentation

L'équation de la tangente en x_n est

$$y(x) = f(x_n) + f'(x_n) \times (x - x_n)$$

On détermine x_{n+1} avec la relation $y(x_{n+1}) = 0$

Cela conduit à

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

Implémentation

```python
def Newton(f, fprime, x, eps, maxiter) :
    for _ in range(maxiter) :
        # Détermination du terme suivant dans la suite
        nx = x - f(x) / fprime(x)

        # Arrêt car la suite n'évolue plus
        if abs(nx-x) < eps :
            return nx

        # Mise à jour de x
        x = nx

    # Arrêt pour nombre maximal d'itérations
    return x
```

Avec scipy

Solution de $x^2 - 2 = 0$ au voisinage de $x_0 = 1$:

```
In []: from scipy.optimize import newton

In []: def f(x):
  ...:     return x**2.0 - 2.0

In []: def fprime(x):
  ...:     return 2.0*x

In []: newton(f, 1.0, fprime)
Out[]: 1.4142135623730951
```

On peut omettre `fprime` (= méthode de la sécante)

Avantages et inconvénients

Avantages :

- on n'a pas besoin d'un encadrement
- il n'est pas nécessaire que la fonction change de signe
- la convergence est rapide

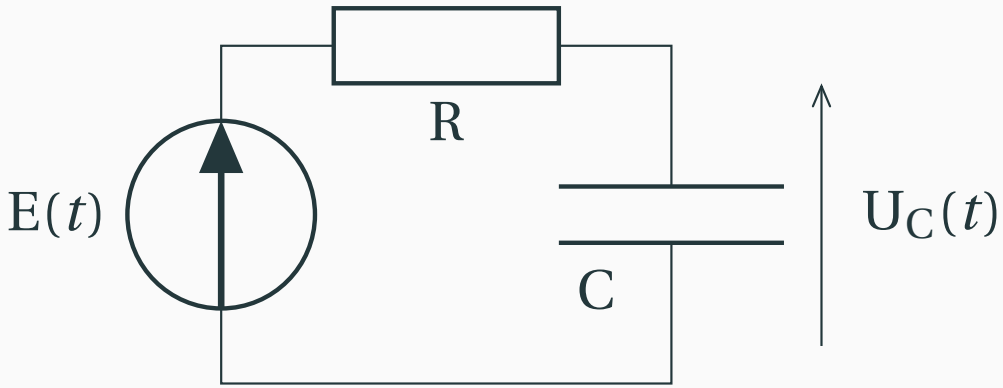
Inconvénients :

- la méthode de Newton-Raphson nécessite de connaître f'
- la convergence n'et pas garantie
- forte sensibilité à x_0

Déroulement de la présentation

Résolution d'équations différentielles

Considérons un circuit :



On prendra $C = 1{,}0\,\mu F$, $R = 1{,}0\,k\Omega$

Le temps caractéristique est donc $\tau = RC = 1{,}0\,ms$

Résolution d'équations différentielles

On suppose que le générateur délivre :

- une tension nulle si $t < 0$
- une tension $\mathrm{E}_0 = 5\,\mathrm{V}$ si $t \geqslant 0$

On cherche à résoudre $\dfrac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$

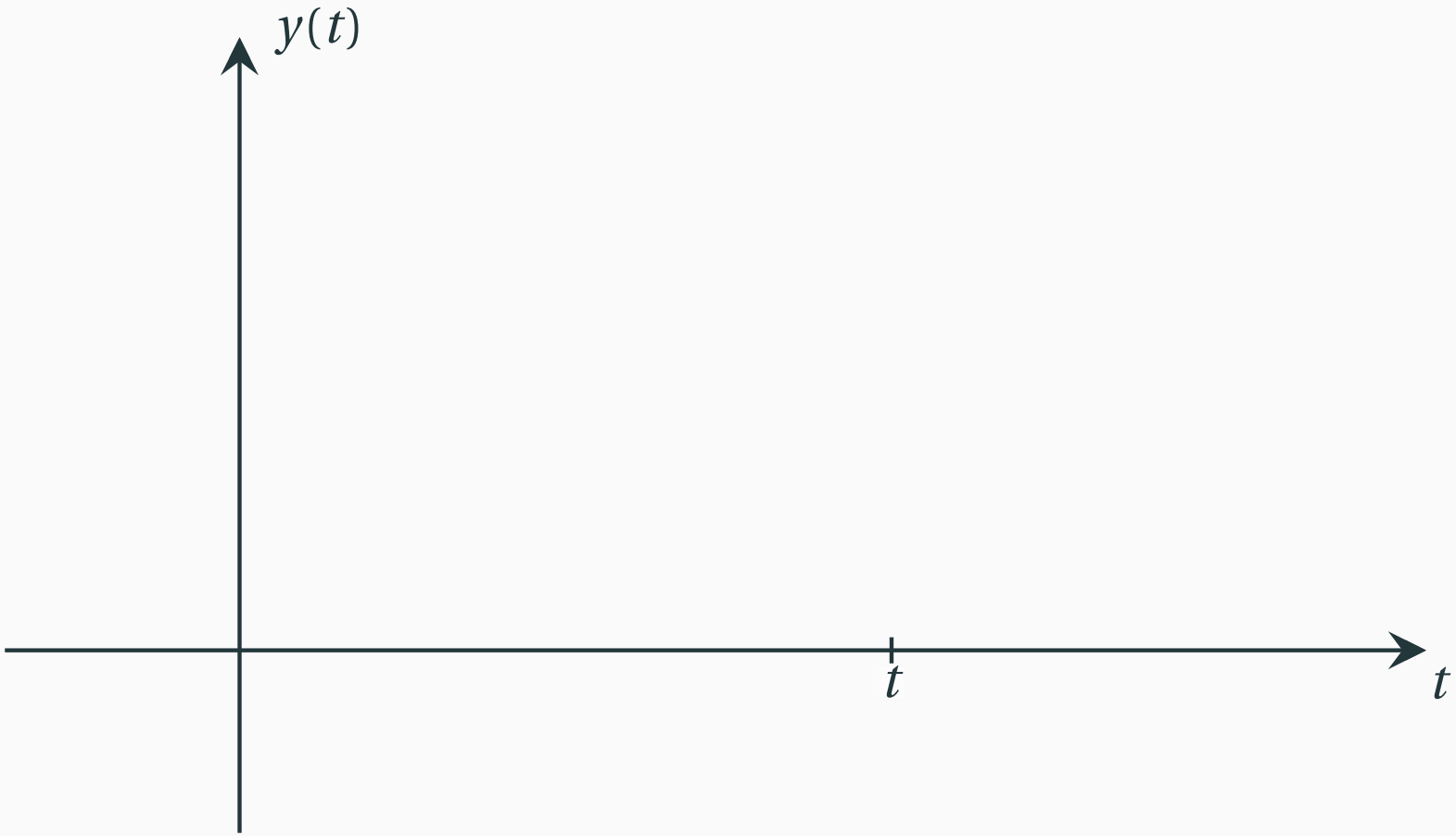
avec $y(t) = \mathrm{U_C}(t)$ et $f(\mathrm{U_C}(t), t) = \frac{1}{\mathrm{RC}}(\mathrm{E}(t) - \mathrm{U_C}(t))$

Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$

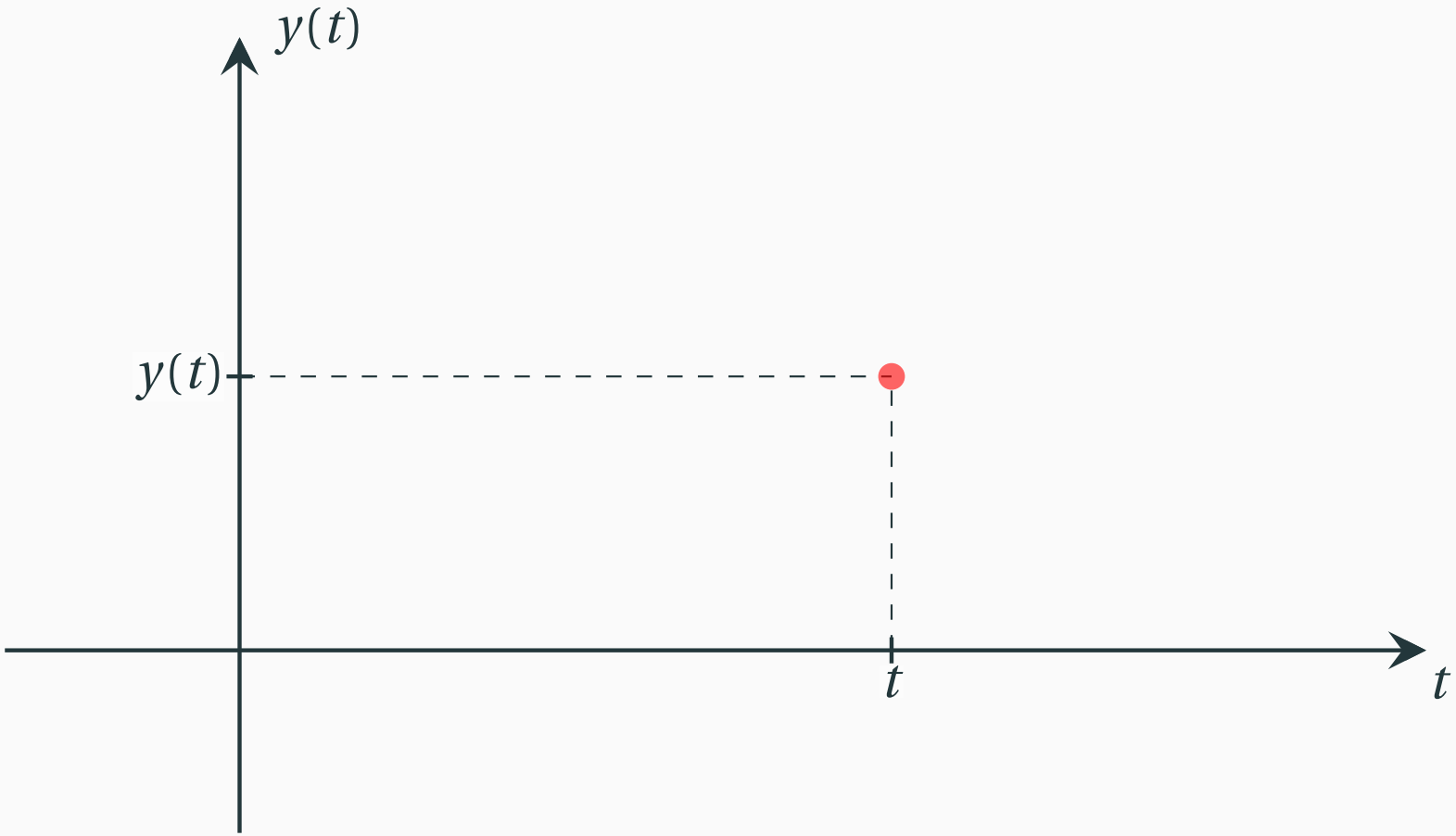
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



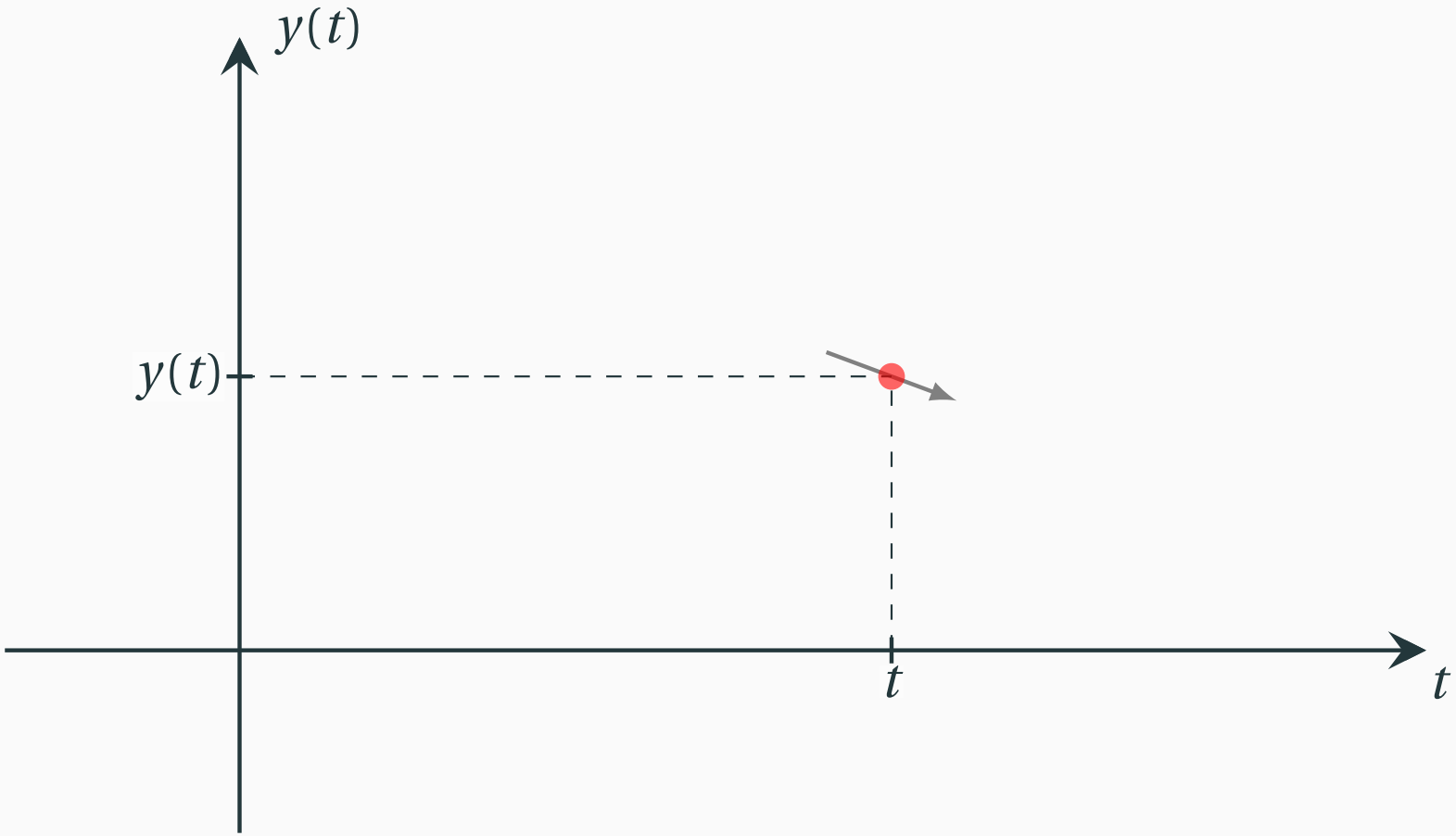
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



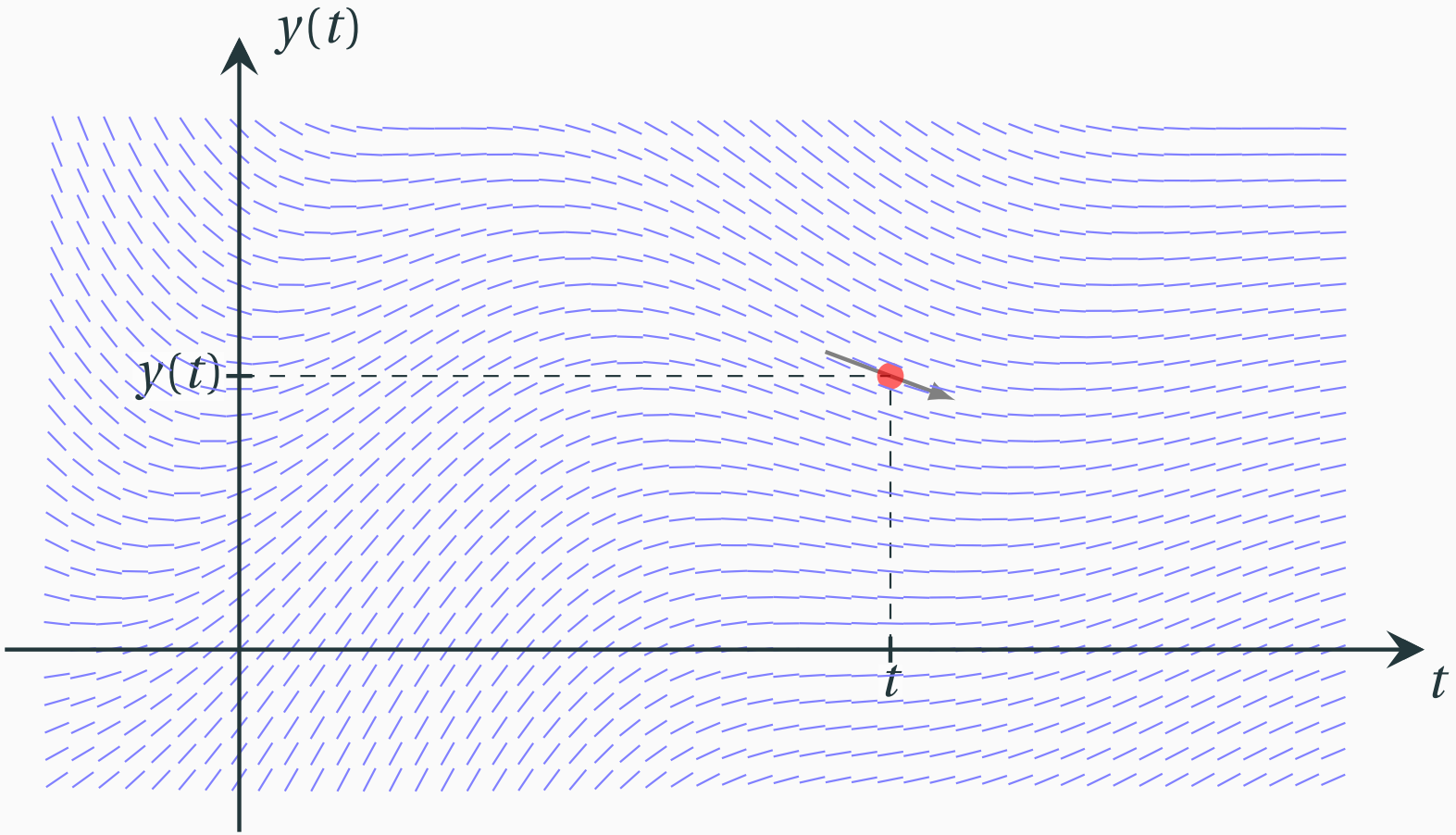
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



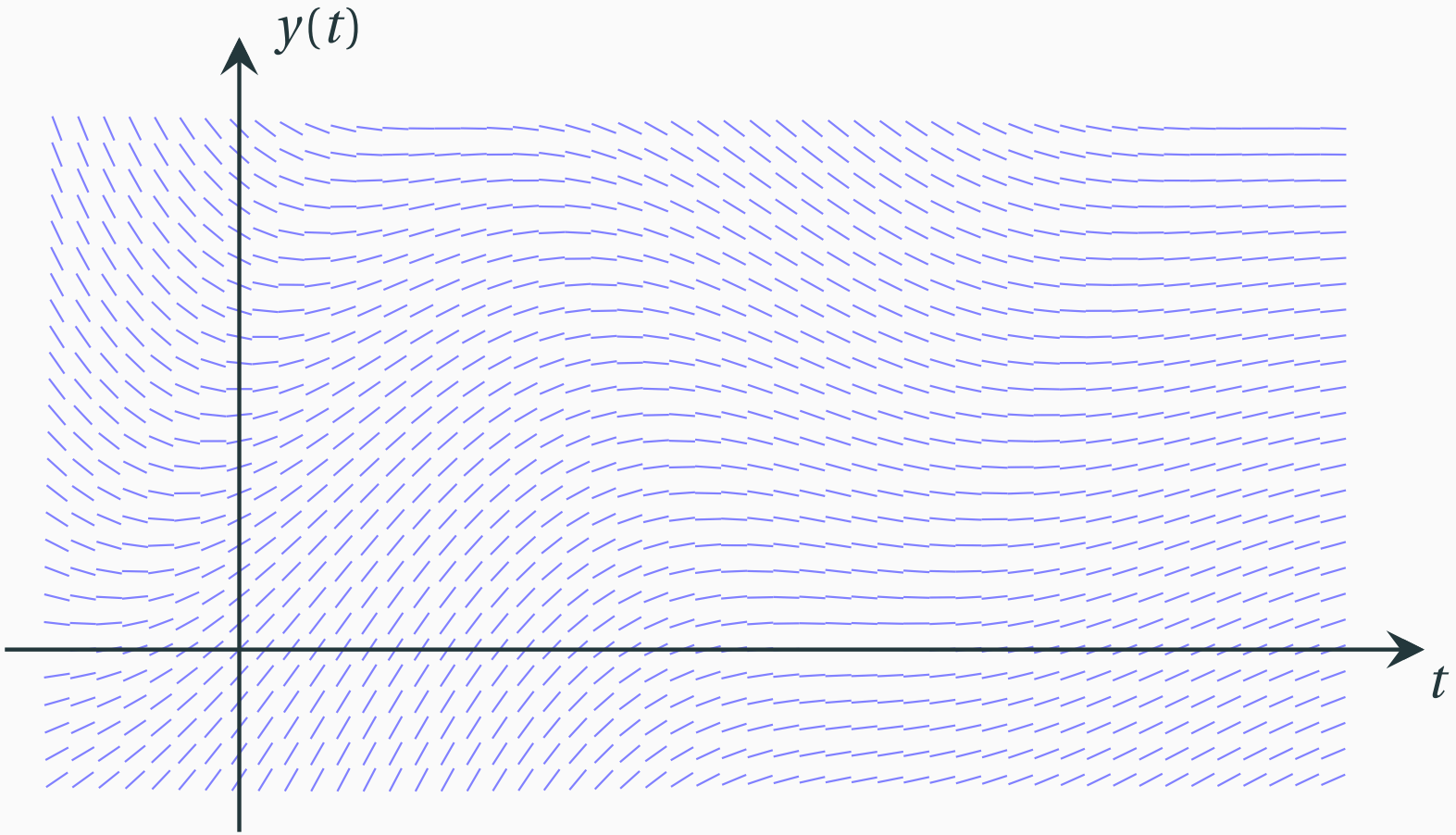
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



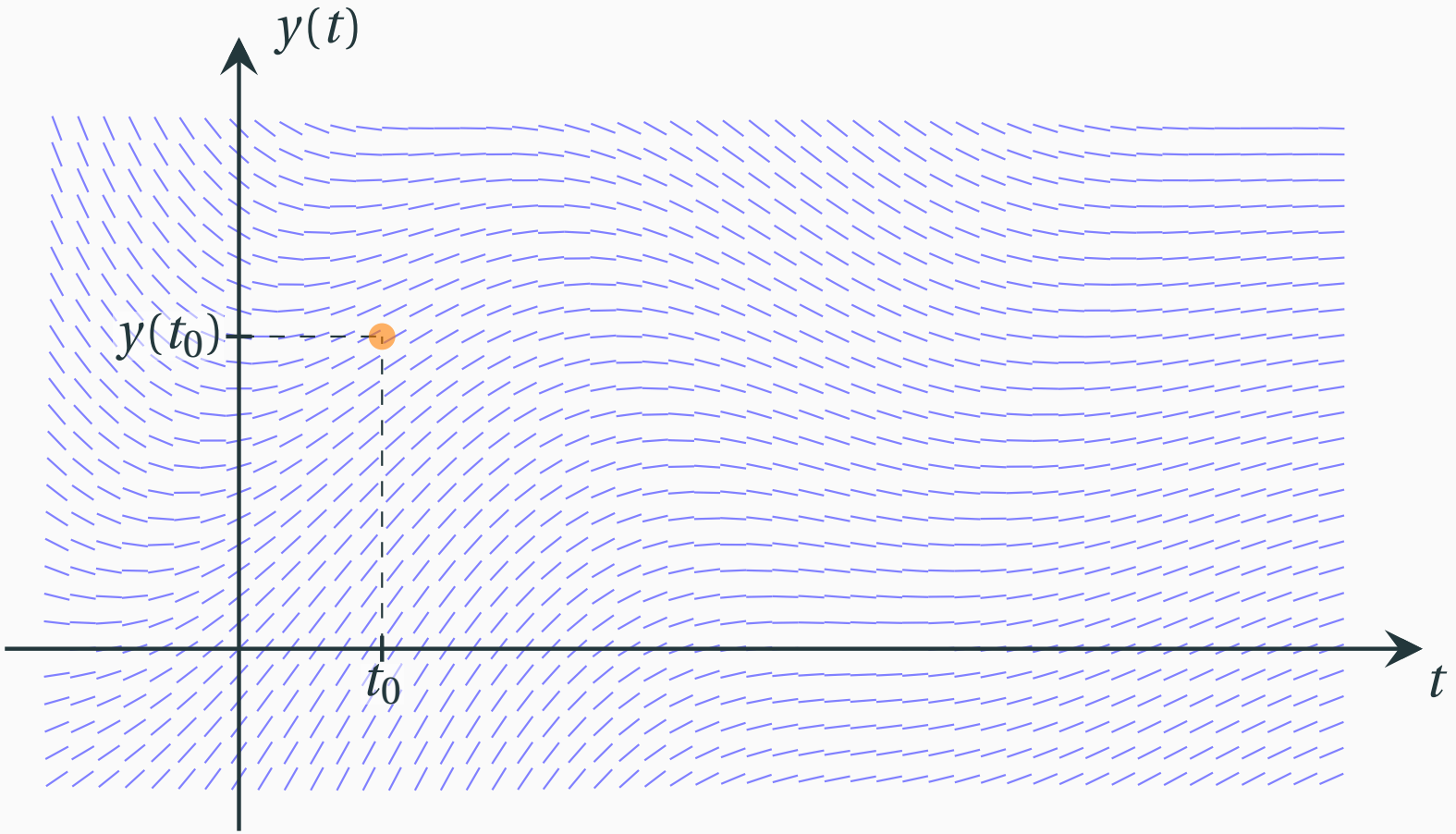
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



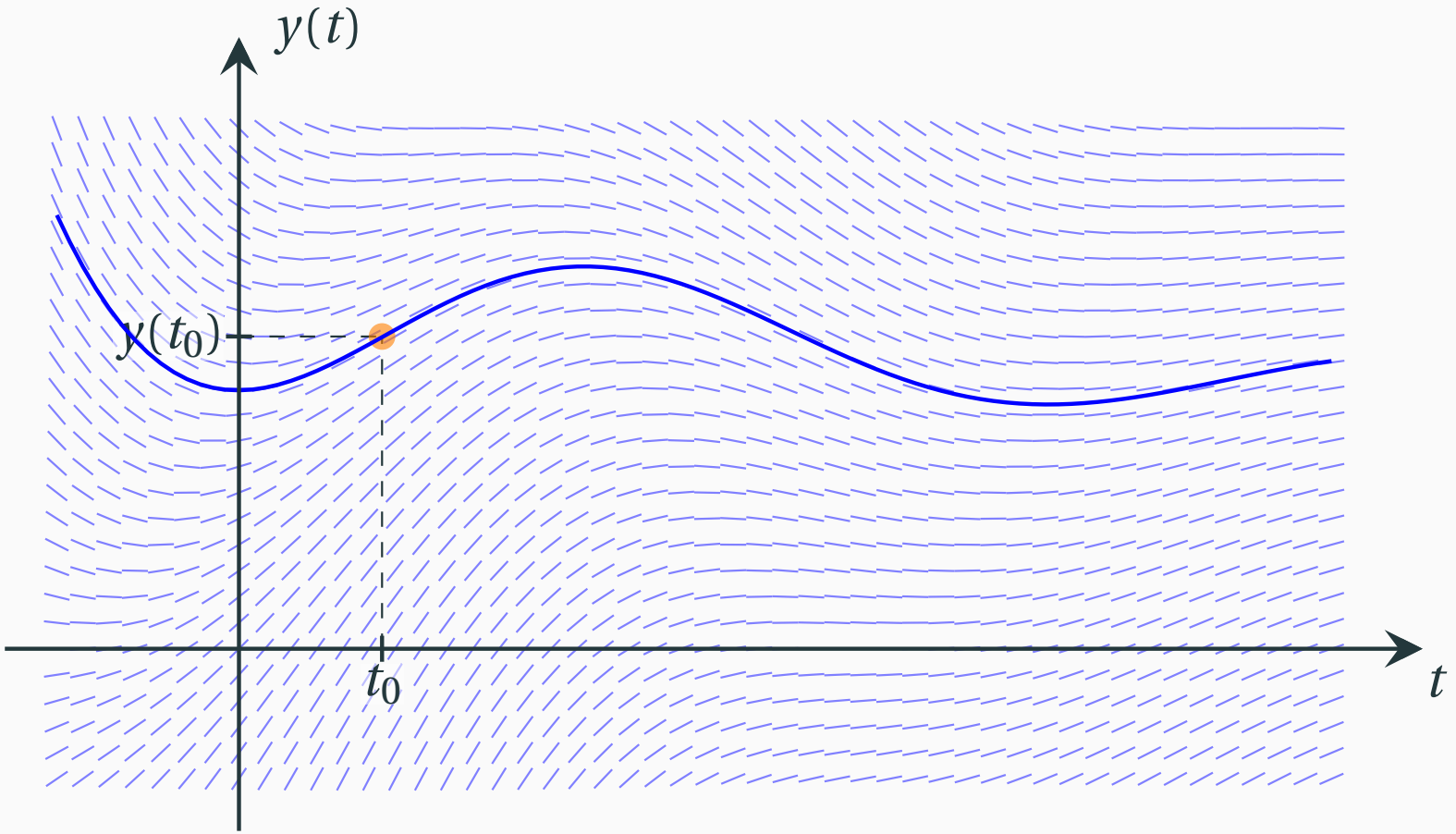
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



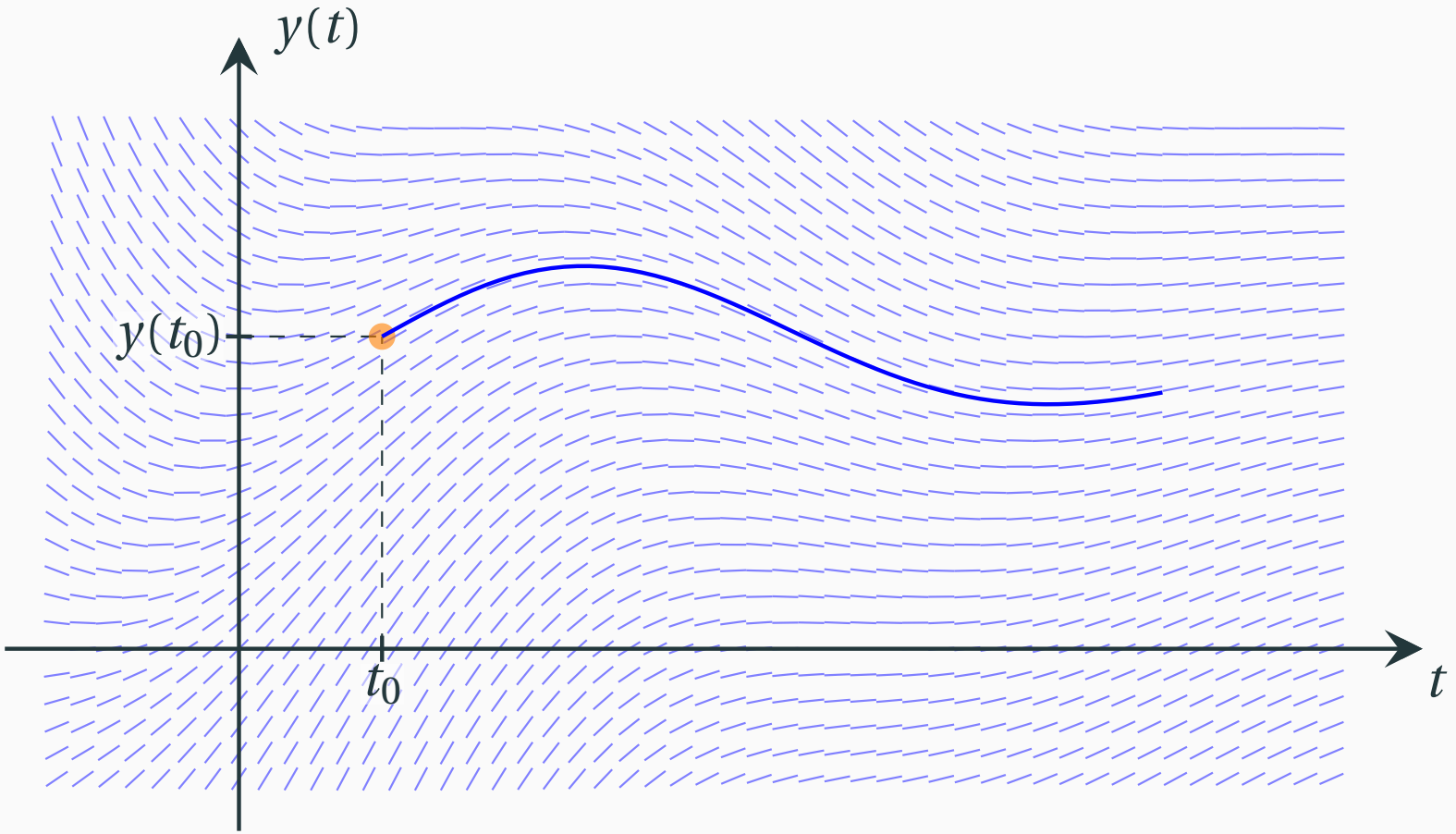
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



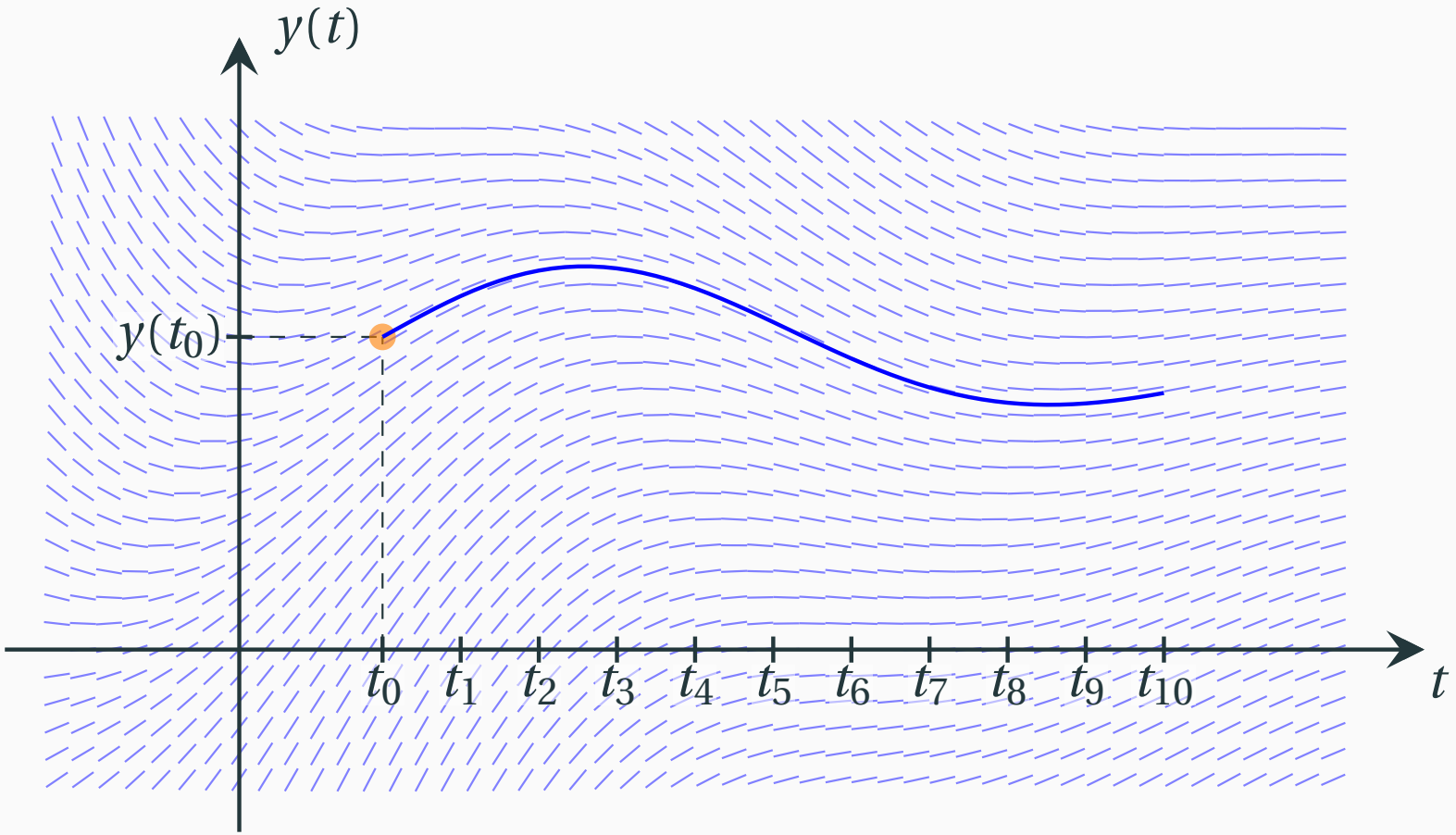
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



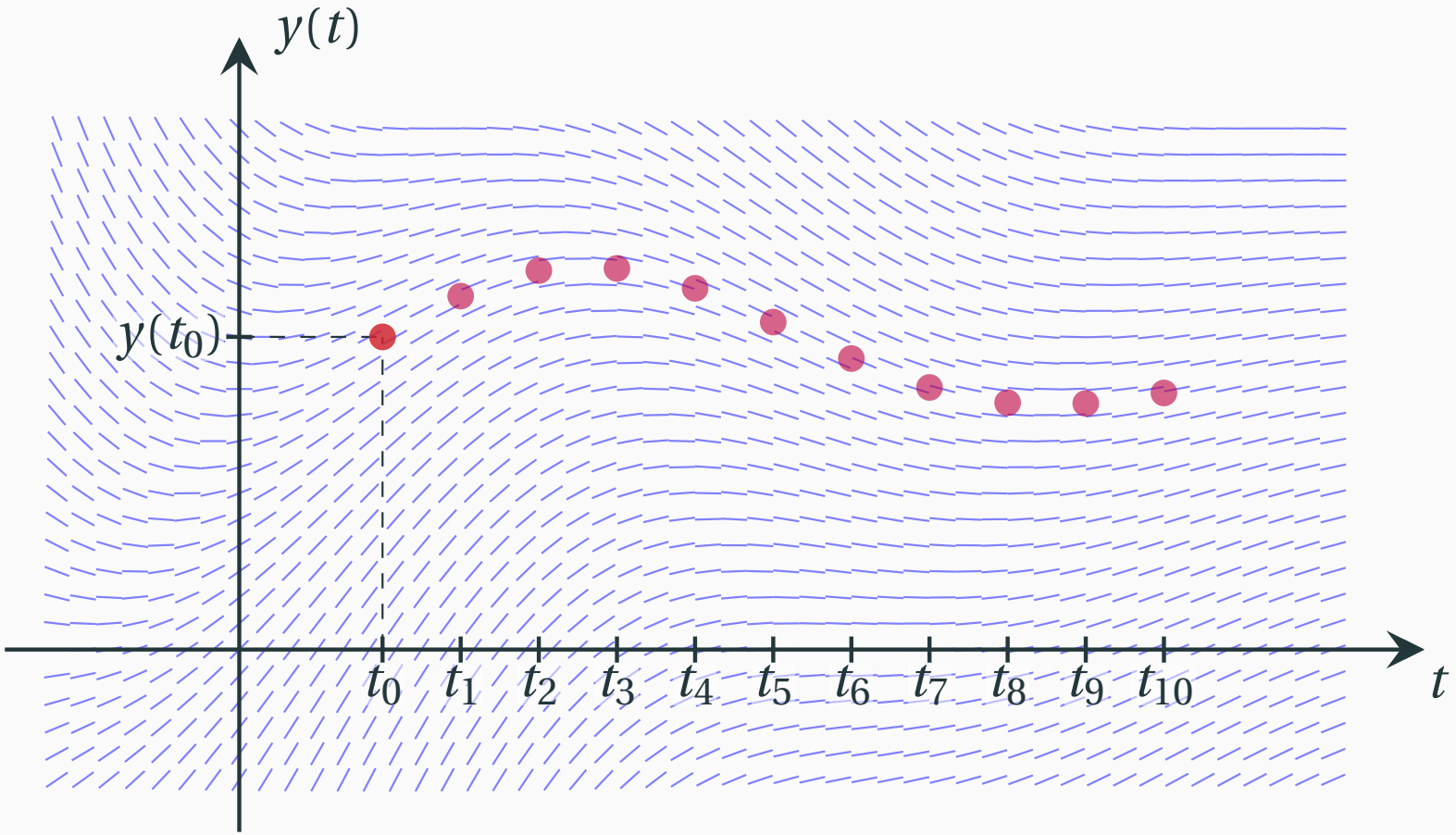
Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



Interprétation géométrique des équations différentielles

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$$



Résolution numérique

Pour une équation différentielle $\dfrac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$

de solution $y(t)$ pour des conditions initiales $\big(t_0, y(t_0)\big)$

Étant donnés des instants t_i, la résolution numérique

vise à déterminer des y_i aussi proches que possible de $y(t_i)$

Schéma d'Euler explicite

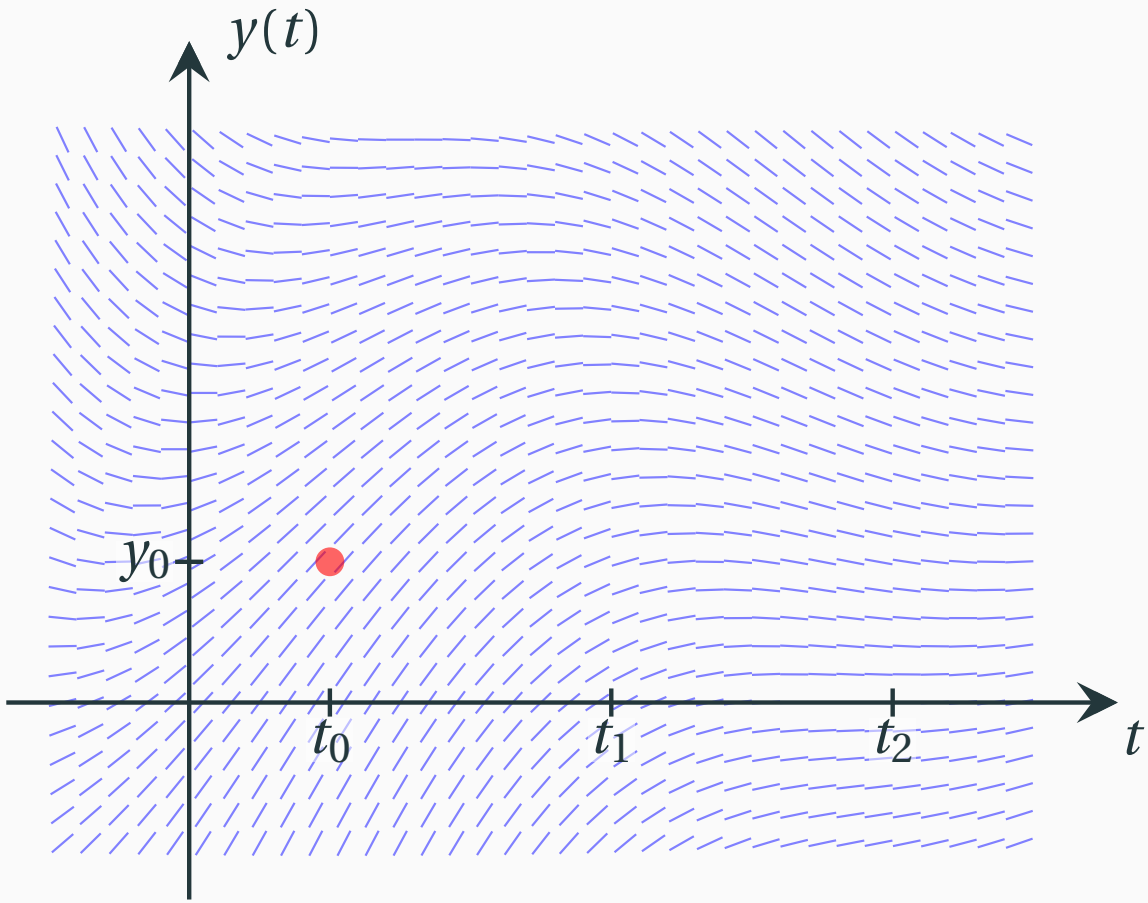


Schéma d'Euler explicite

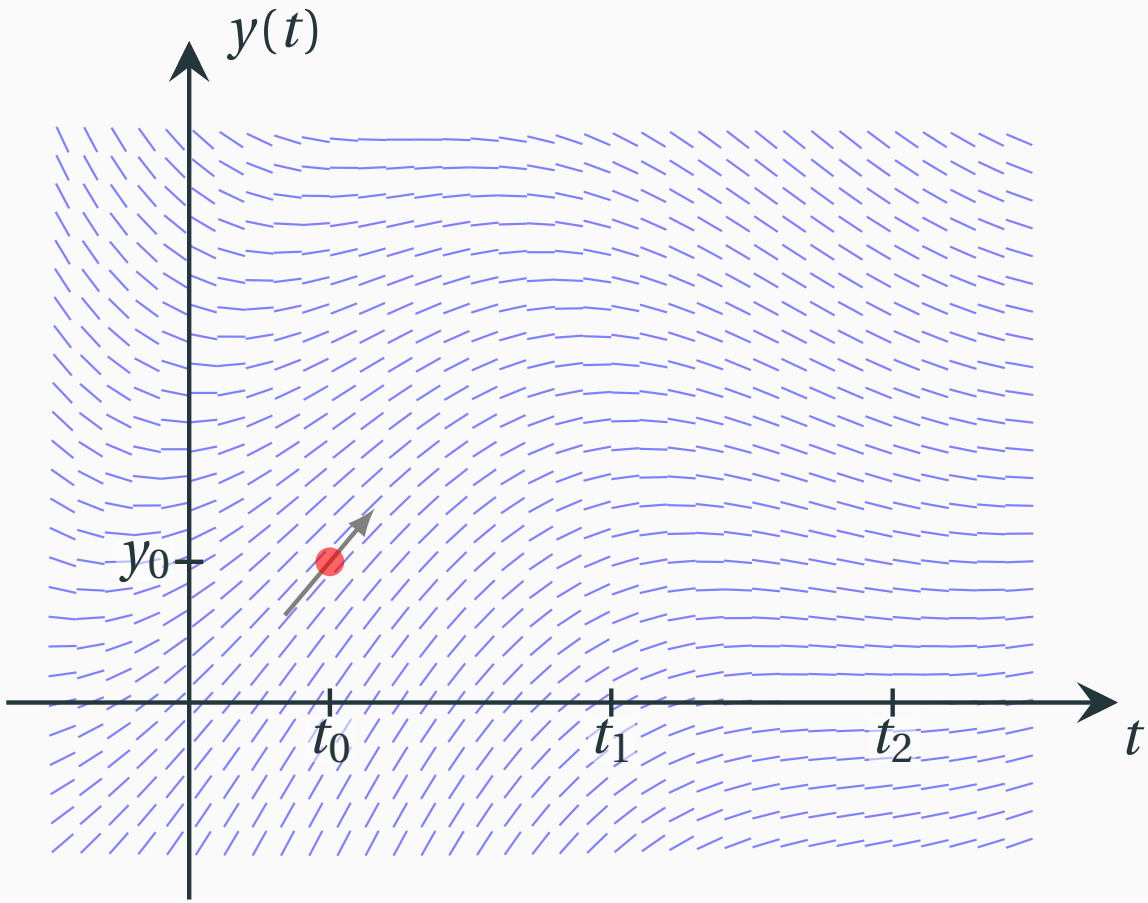


Schéma d'Euler explicite

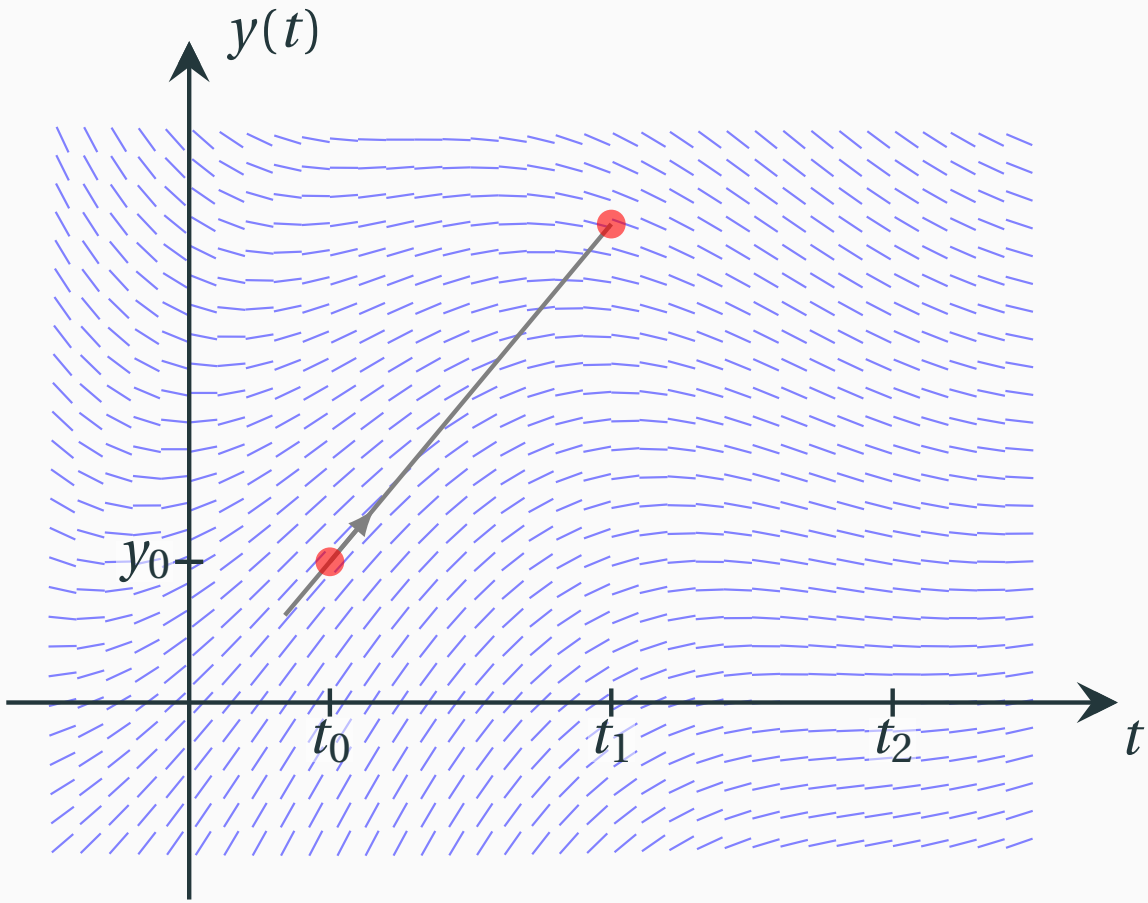


Schéma d'Euler explicite

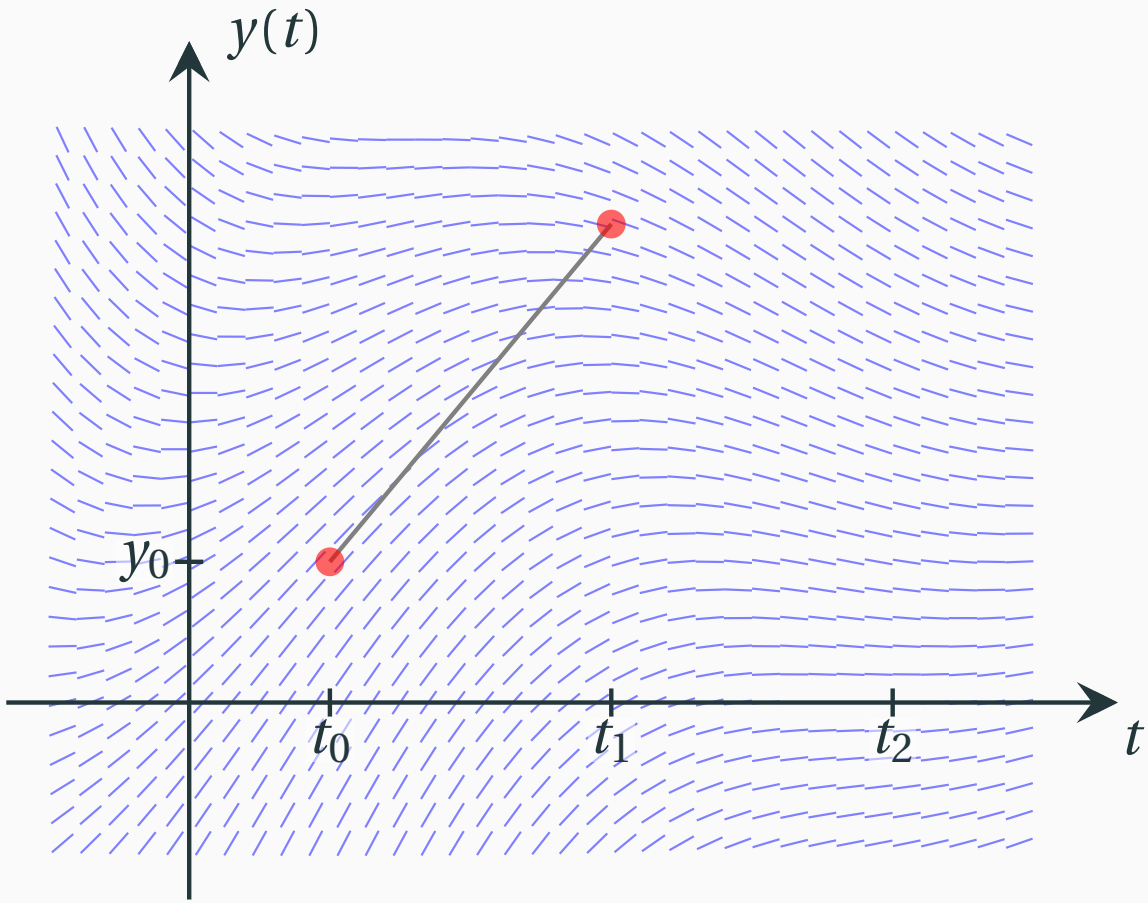


Schéma d'Euler explicite

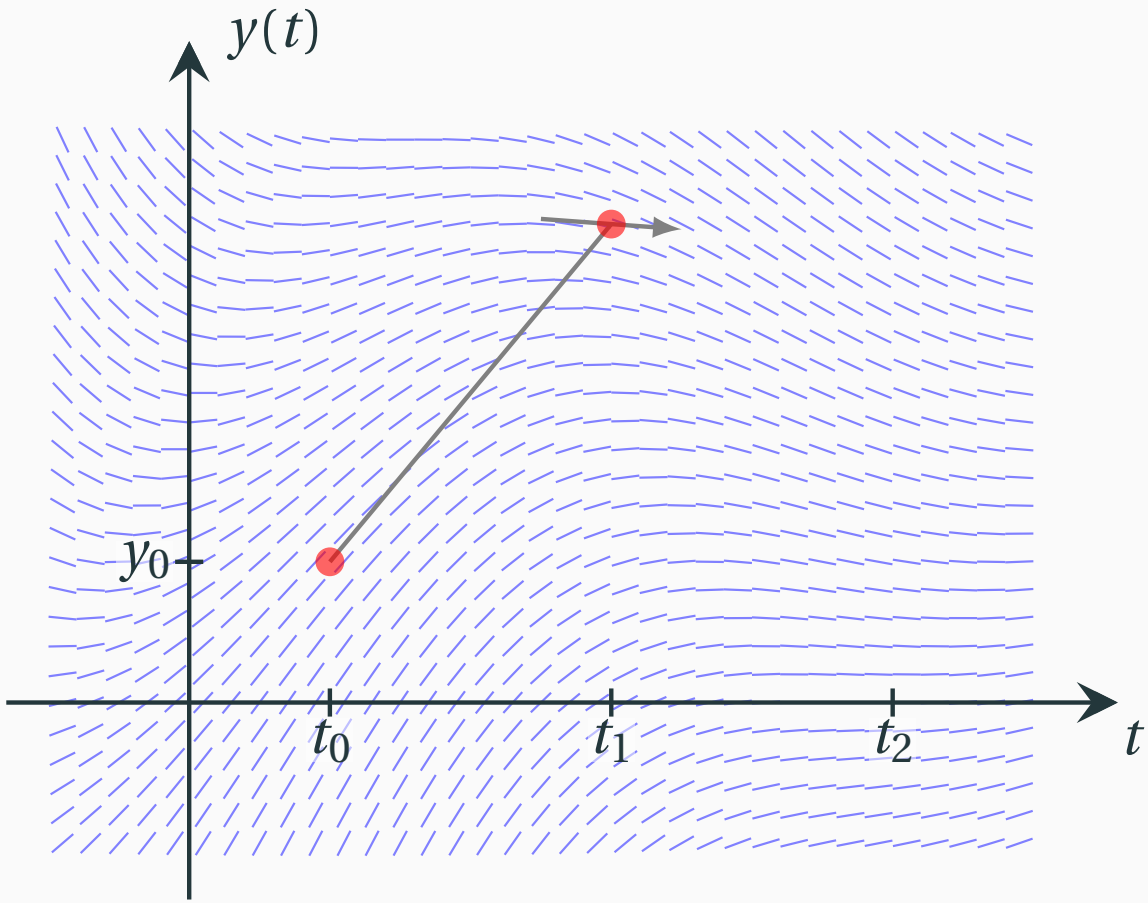


Schéma d'Euler explicite

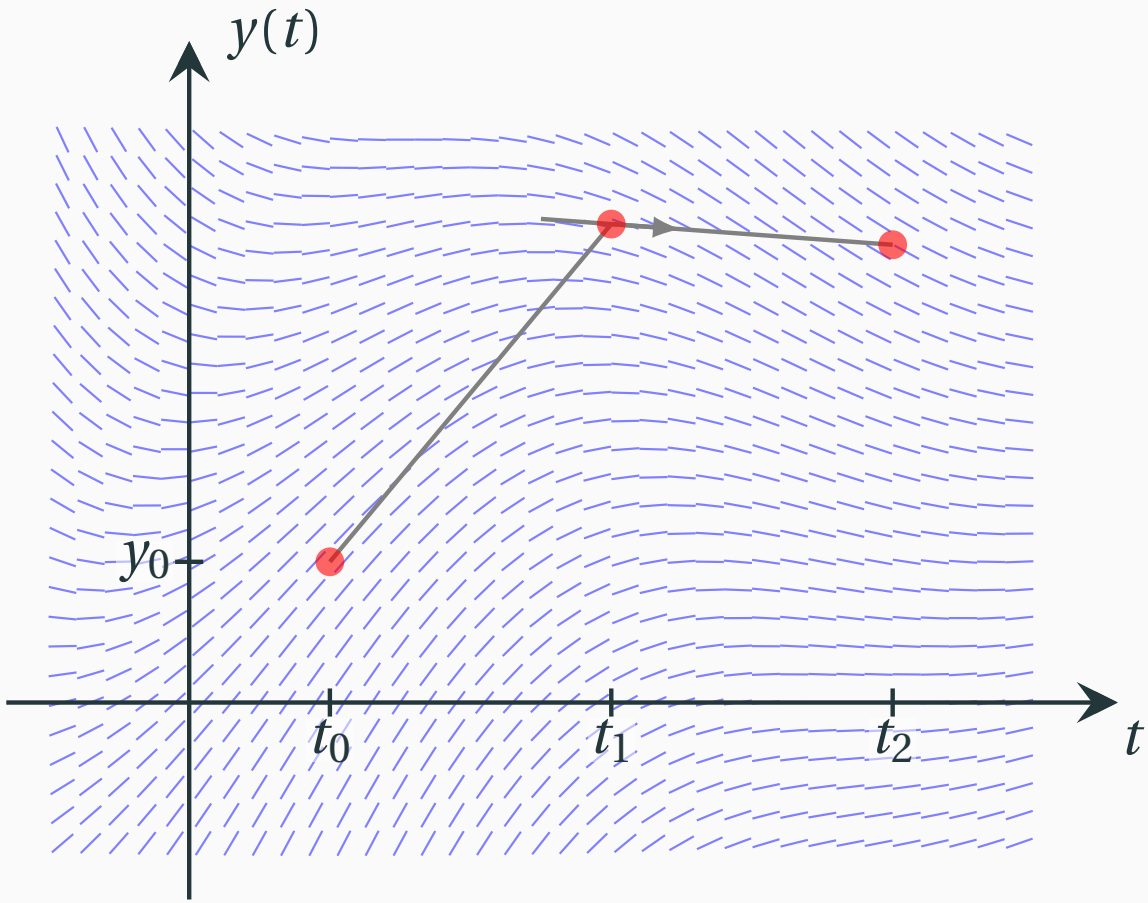


Schéma d'Euler explicite

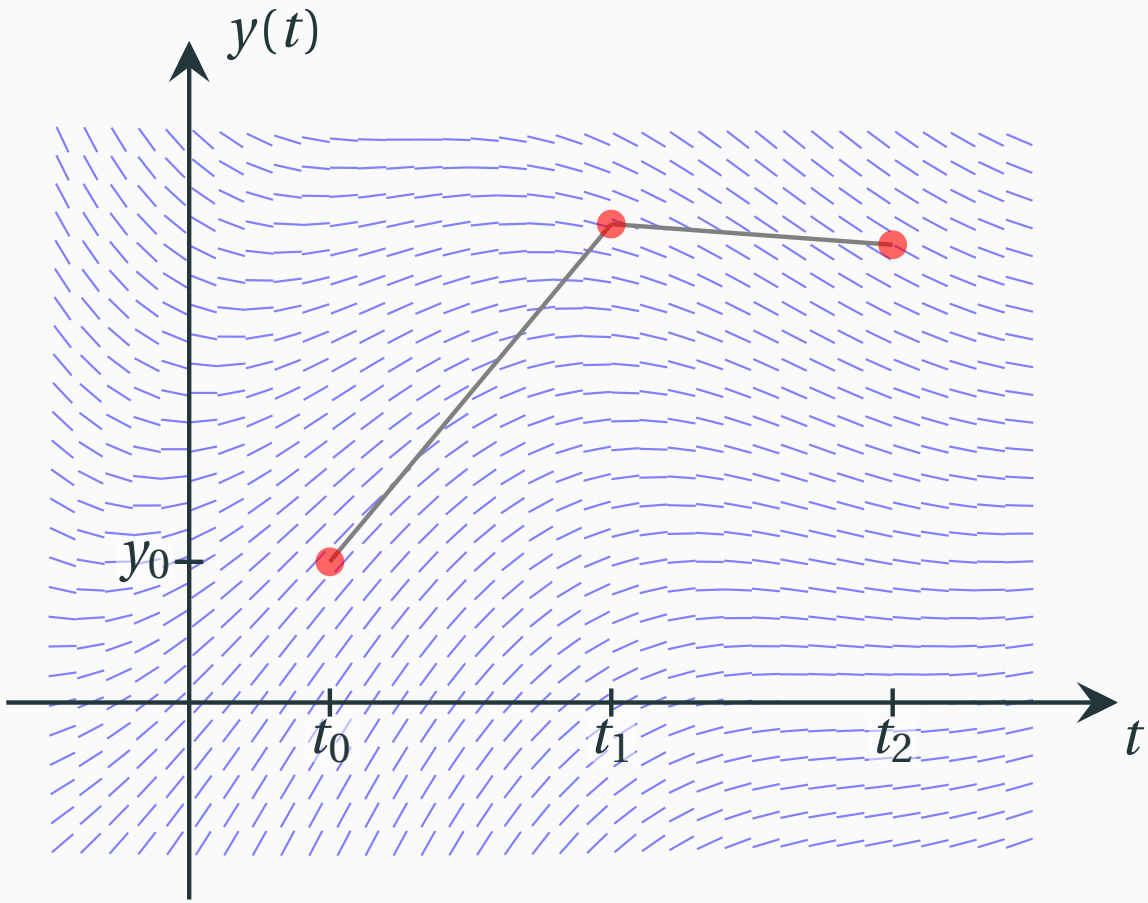


Schéma d'Euler explicite

On détermine y_i à partir de y_{i-1} :

$$y_i = y_{i-1} + f\left(y_{i-1}, t_{i-1}\right) \times (t_i - t_{i-1})$$

$\mathrm{d}t = t_i - t_{i-1}$ est appelé *pas d'intégration*

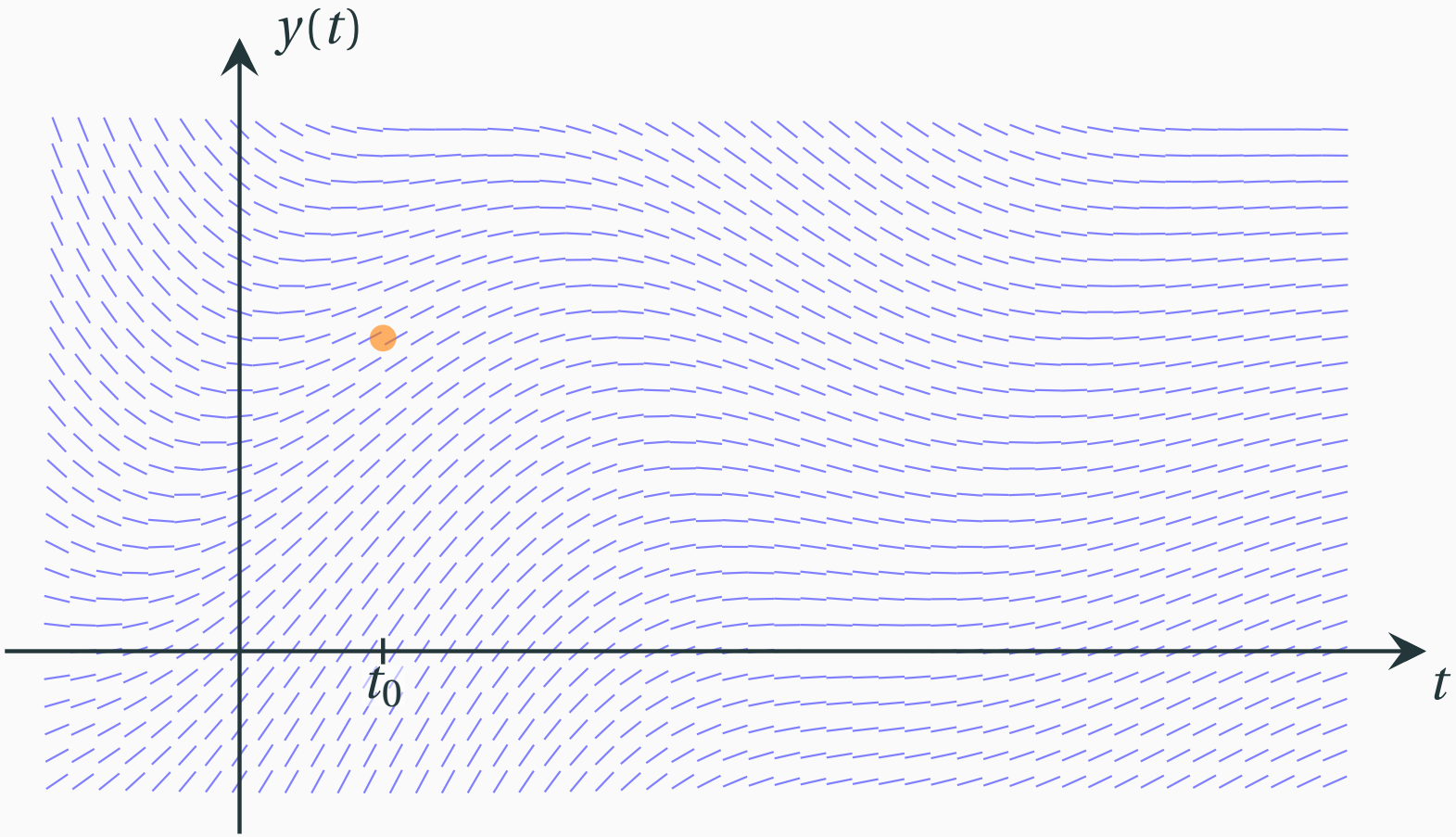
Ainsi, à partir de y_0, on obtient tous les y_i !

Schéma d'Euler explicite

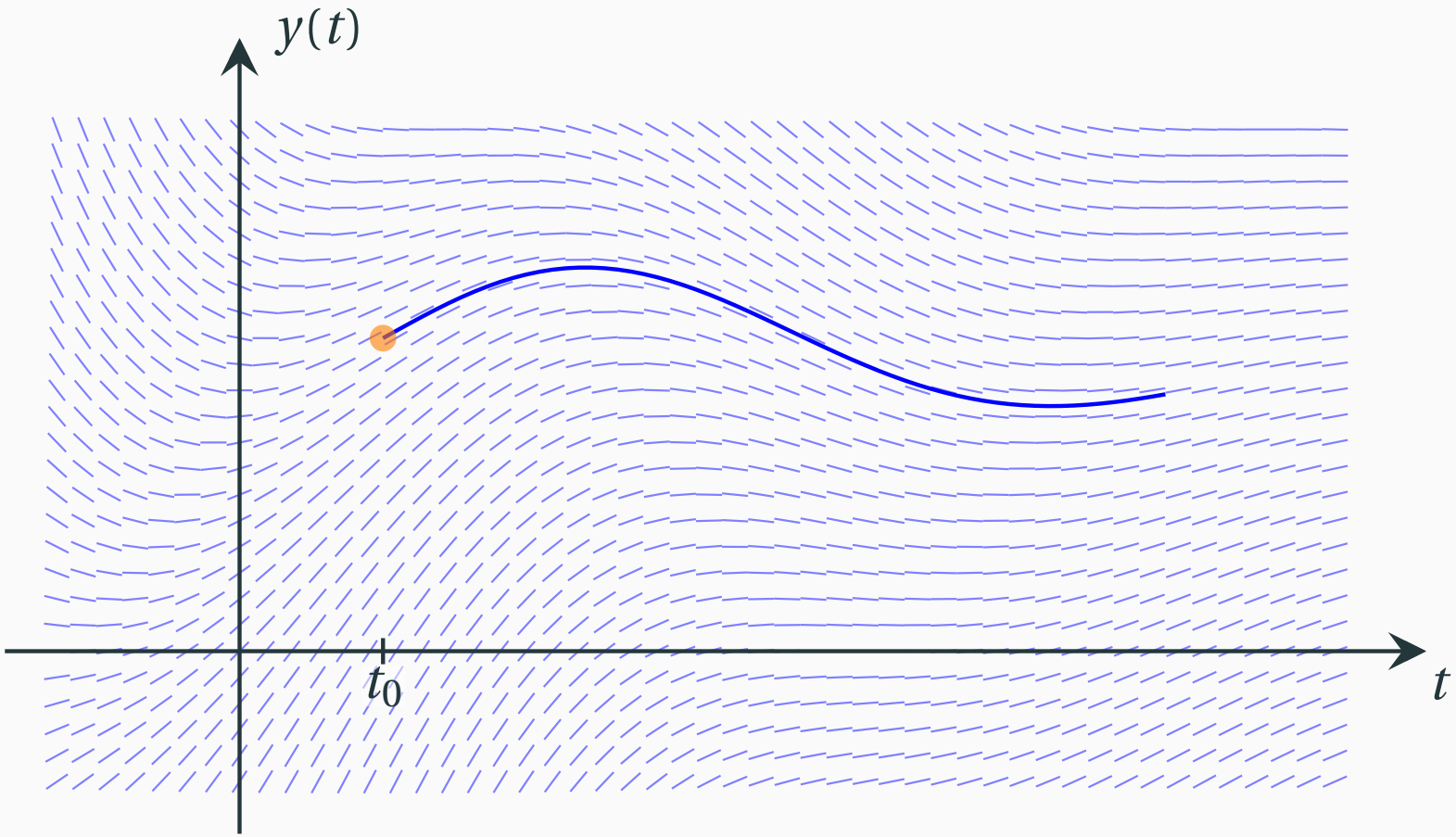
L'implémentation Python ne pose pas de difficulté :

```python
def EulerExplicite(f, y0, T) :  # T = [t_0, t_1, ...]
    Y = [ y0 ]                  # Y va contenir les y_i

    for i in range(1, len(T)) :
        # dt = t_i - t_{i-1}
        dt = T[i] - T[i-1]

        # Calcul de y_i à partir de y_{i-1}
        yi = Y[i-1] + f(Y[i-1], T[i-1]) * dt
        Y.append(yi)

    return Y
```

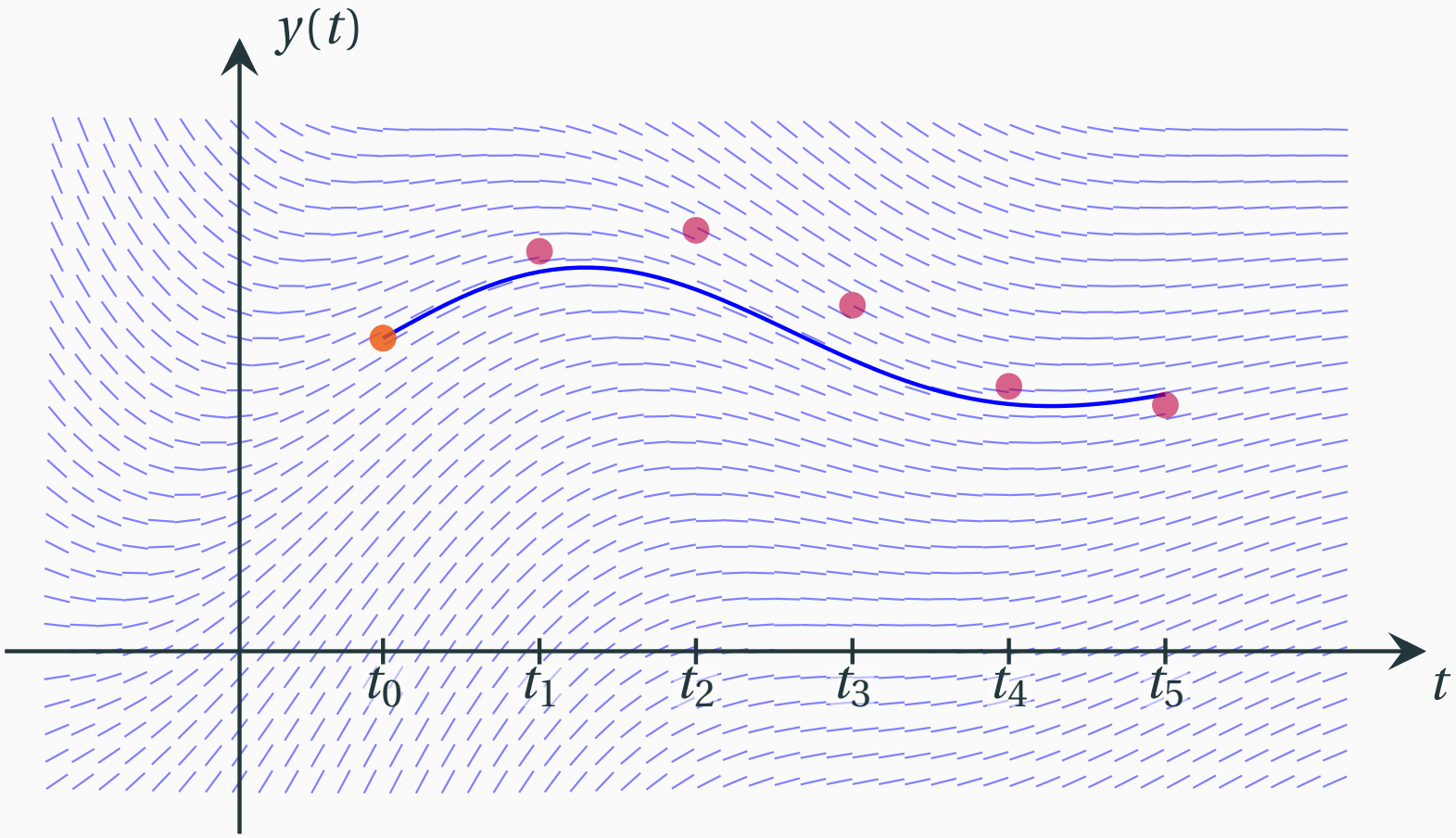
Influence du pas



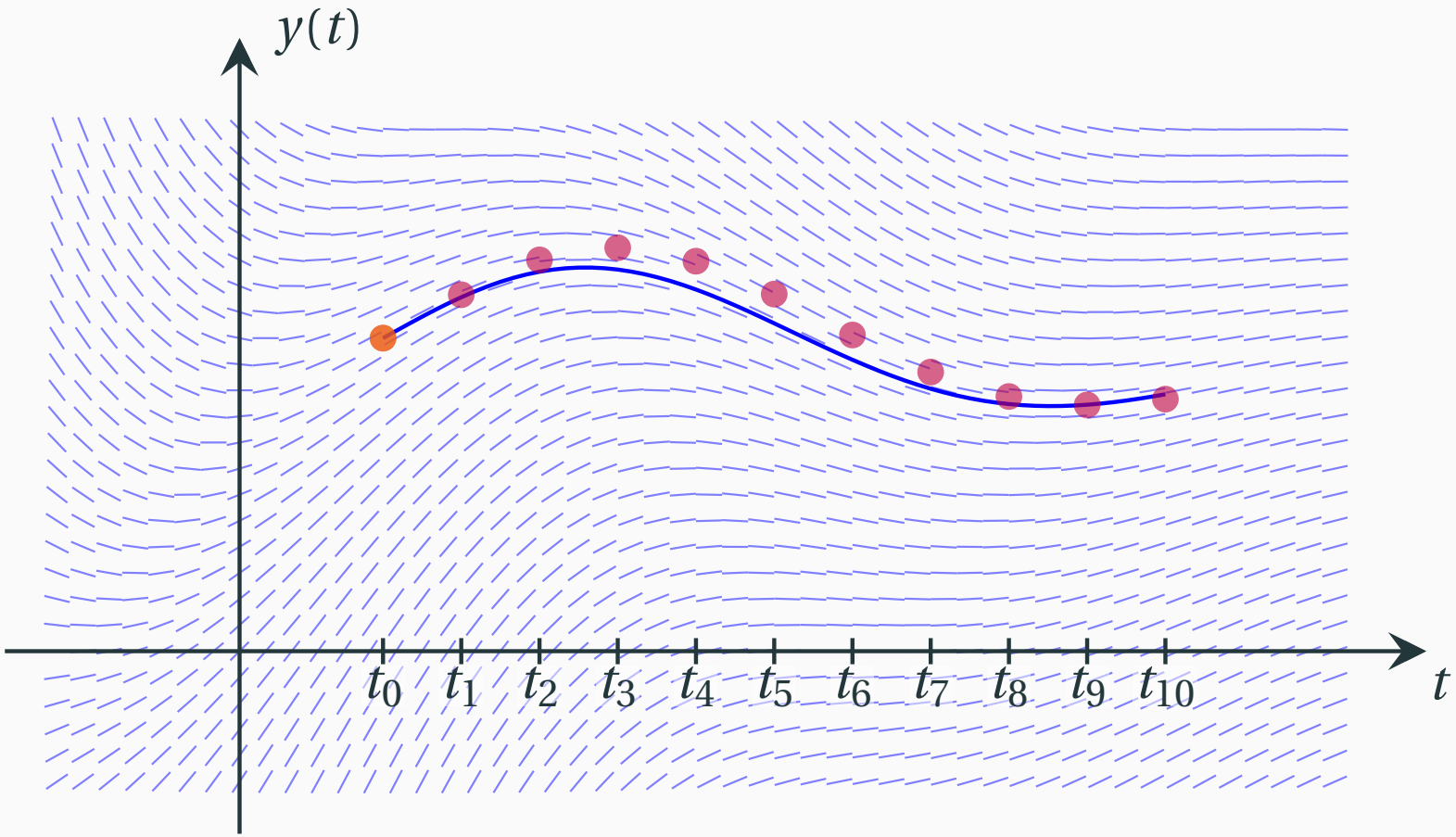
Influence du pas



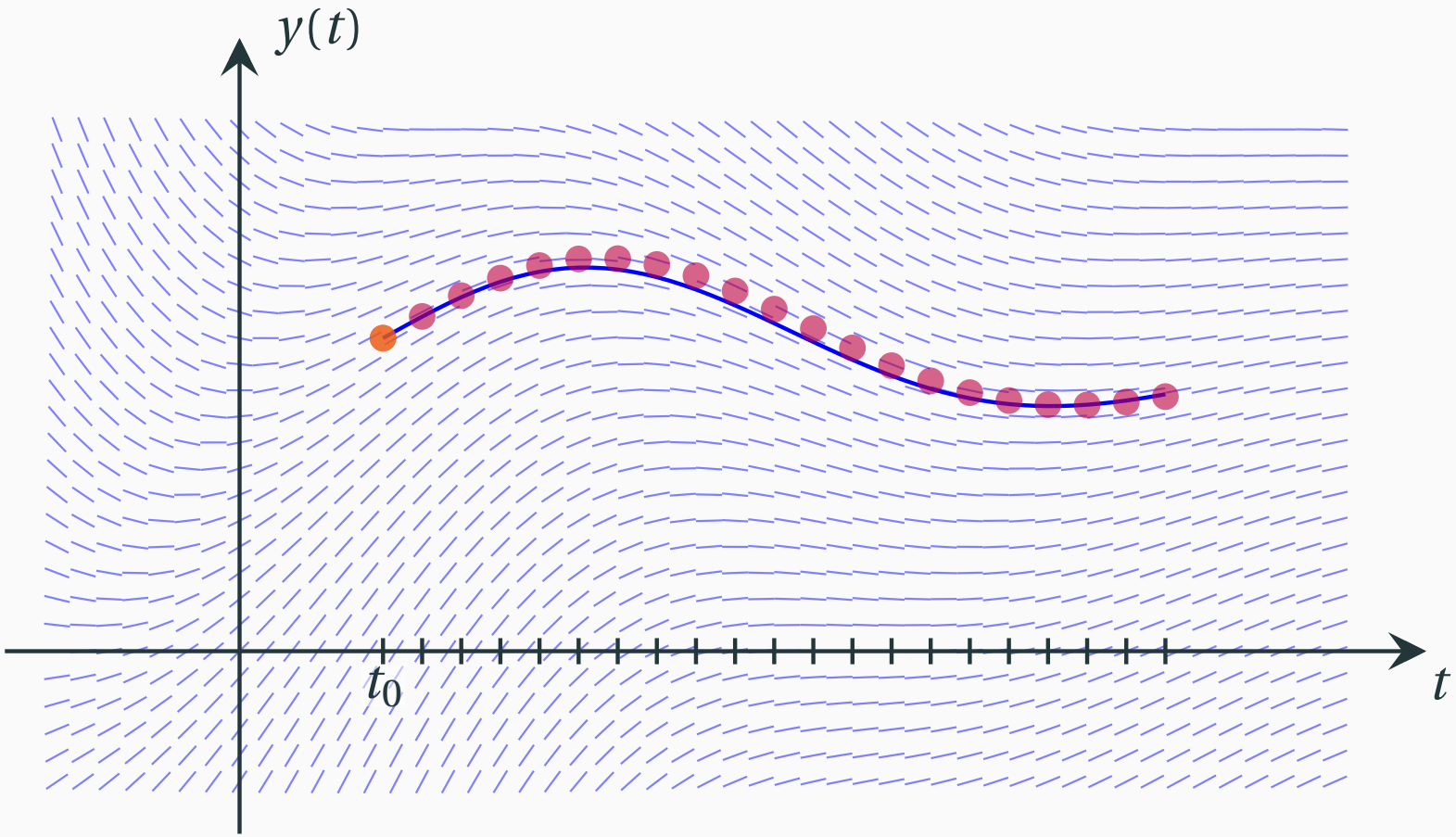
Influence du pas



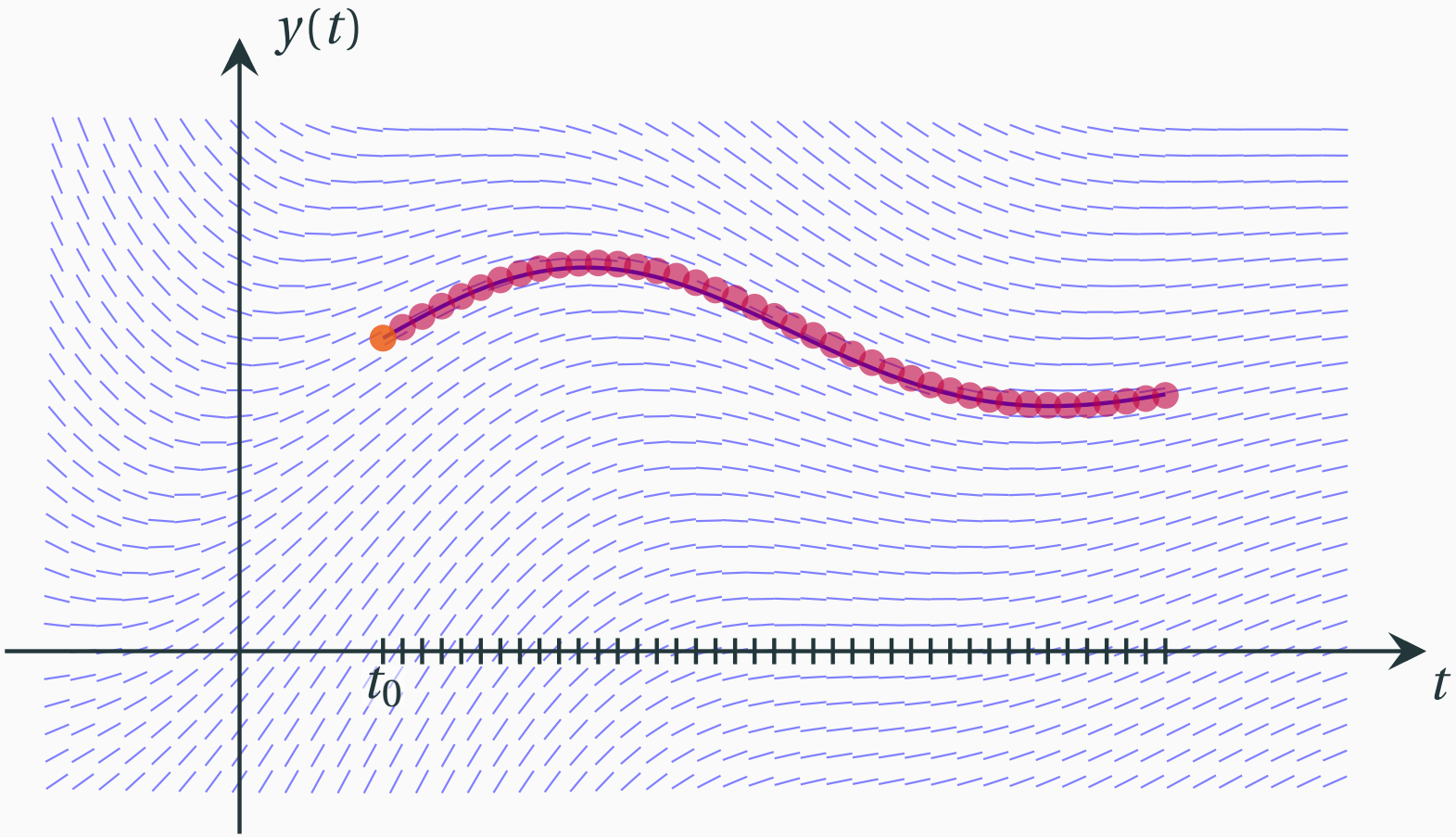
Influence du pas



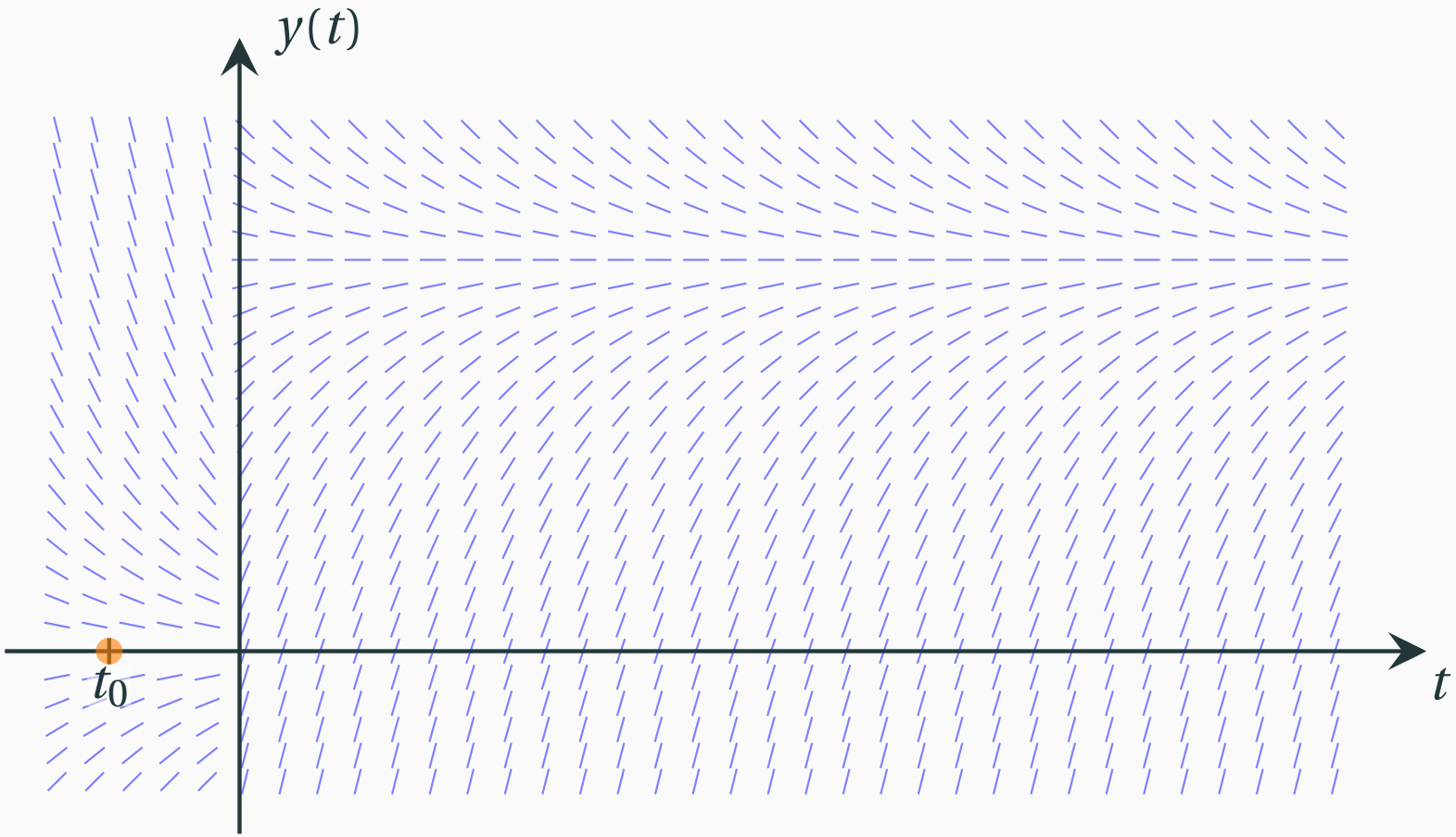
Influence du pas



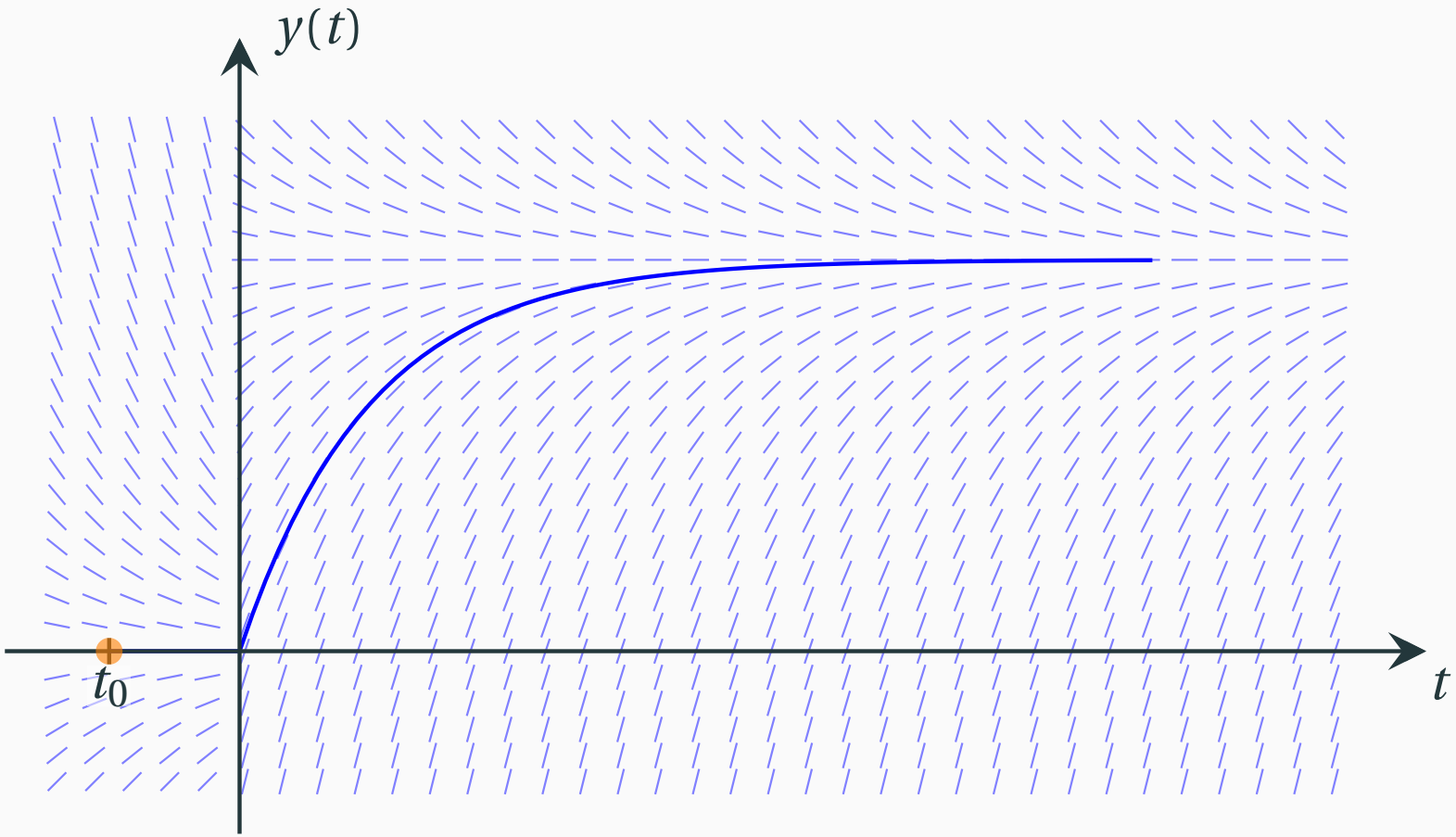
Influence du pas



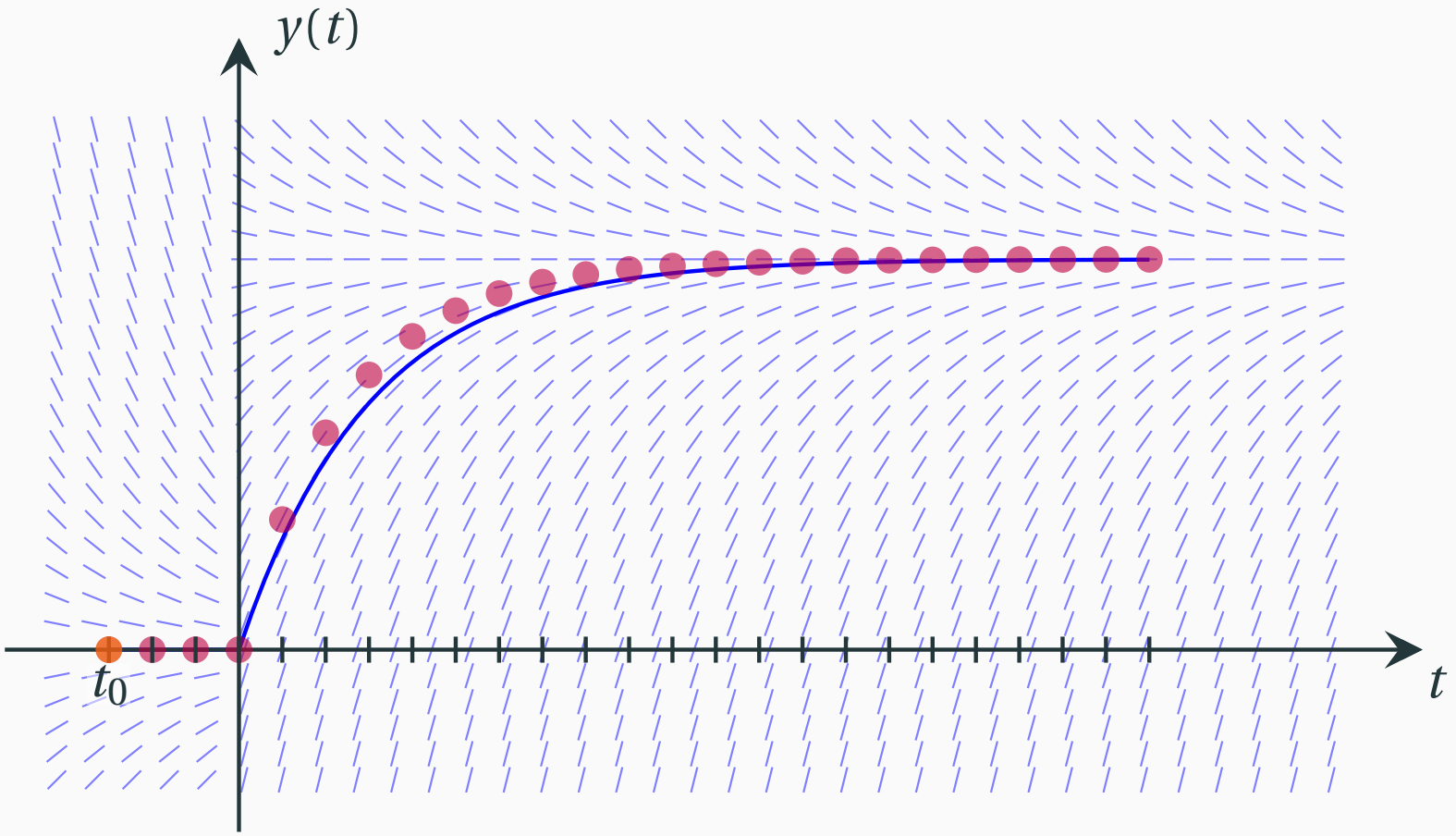
Retour sur l'exemple



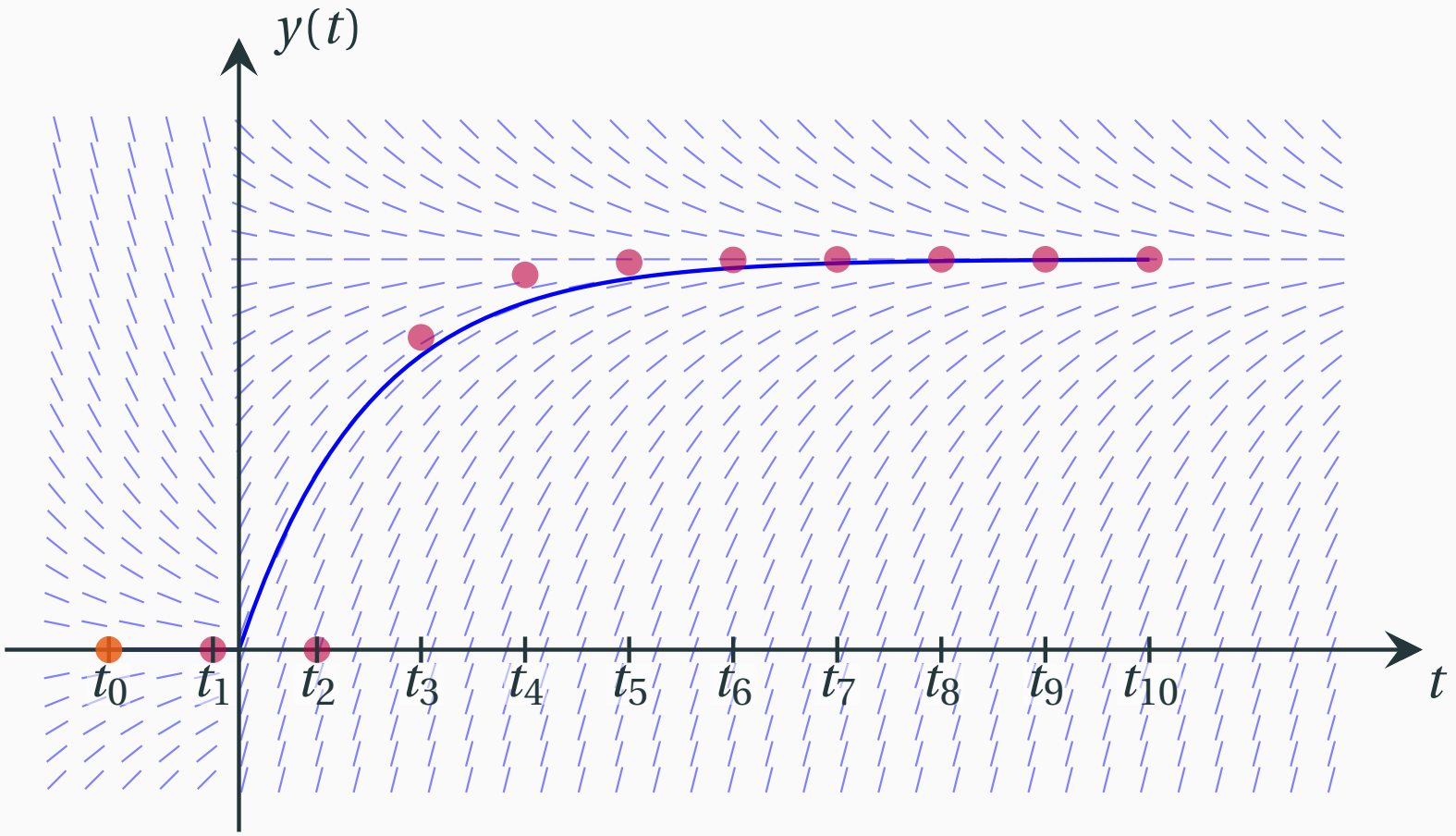
Retour sur l'exemple



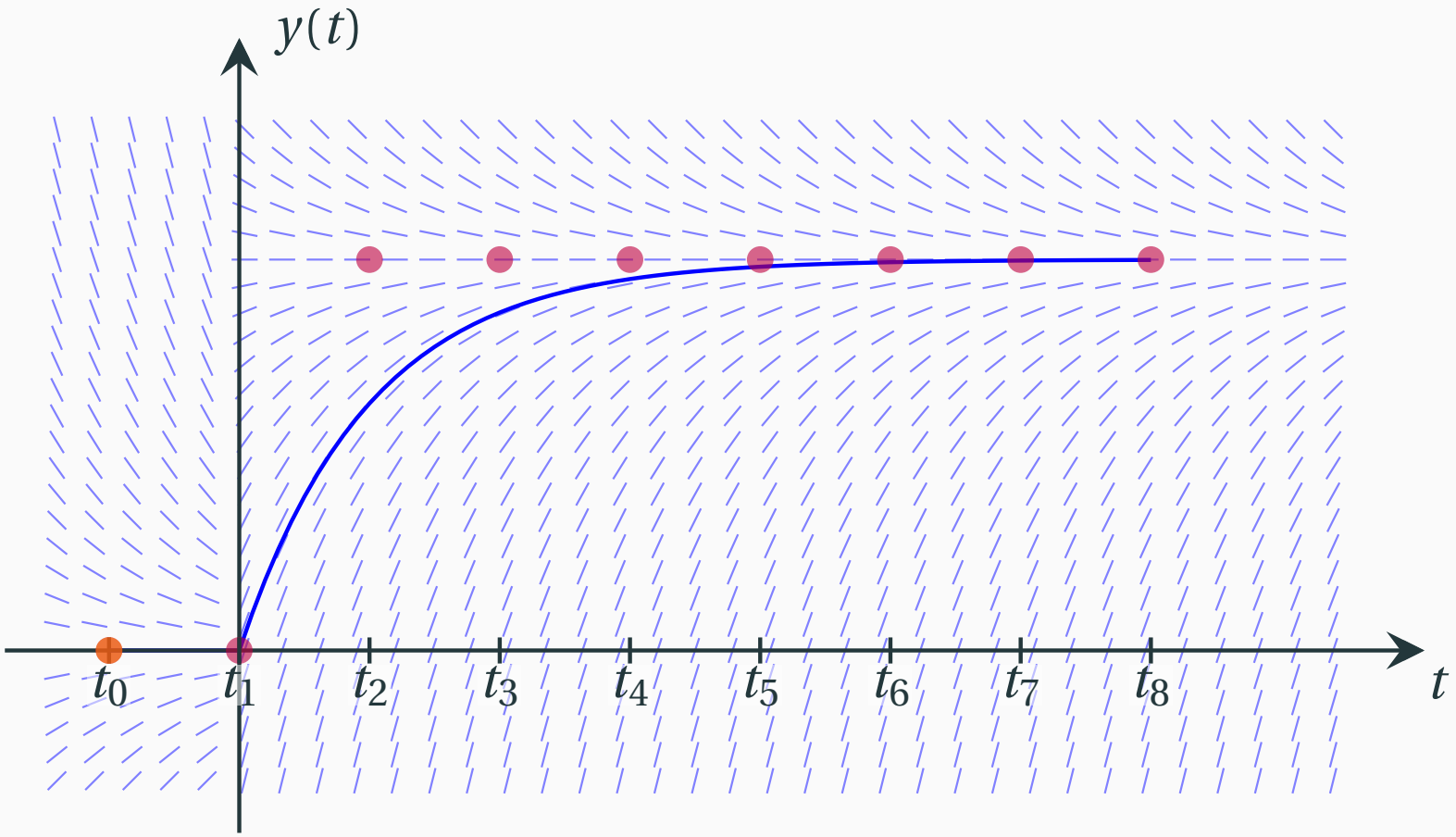
Retour sur l'exemple



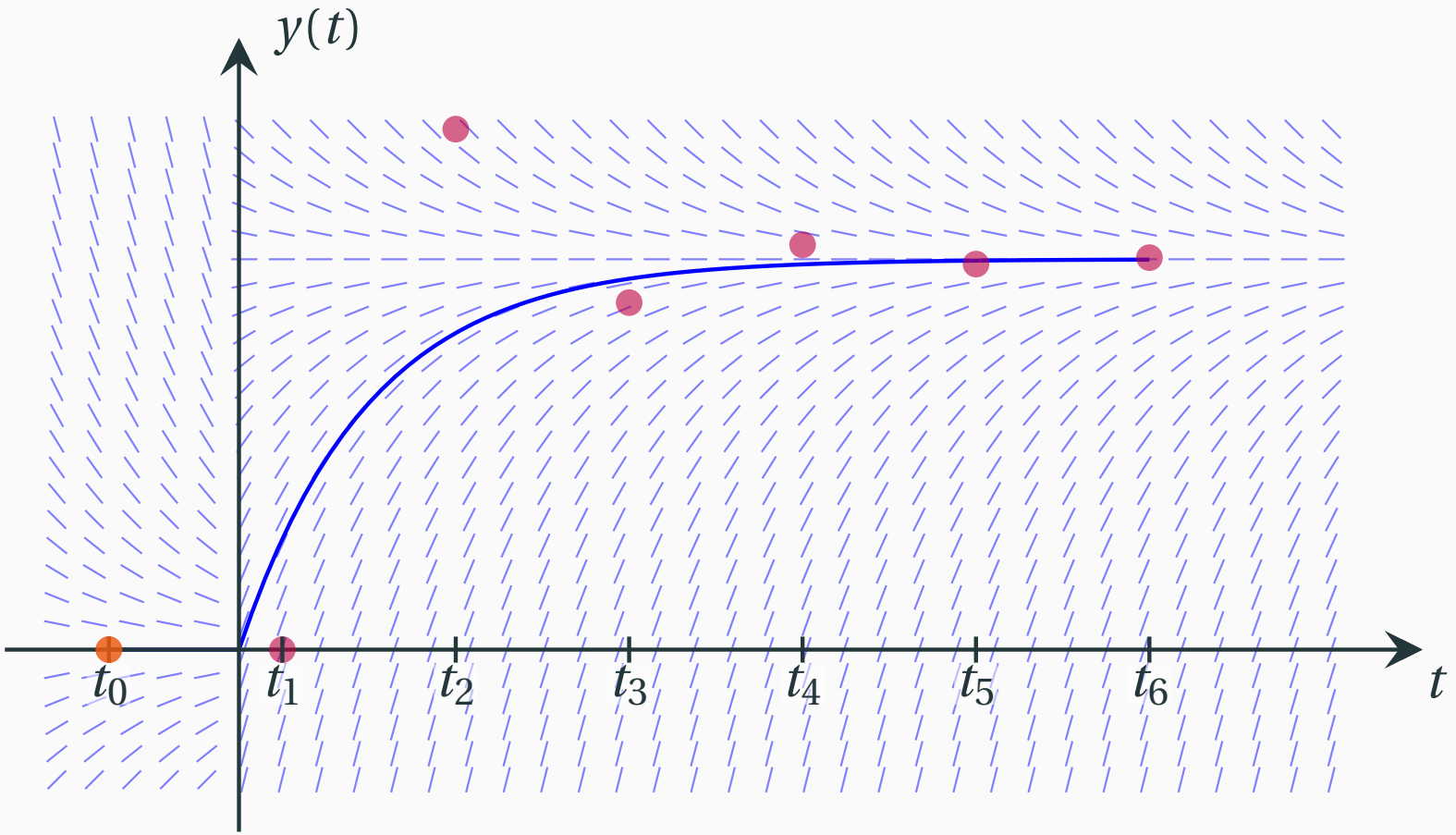
Retour sur l'exemple



Retour sur l'exemple



Retour sur l'exemple



Amélioration des résultats

Réduire suffisamment le pas n'est pas une solution réaliste

Le schéma d'intégration d'Euler n'est jamais utilisé !

Amélioration des résultats

Réduire suffisamment le pas n'est pas une solution réaliste

Le schéma d'intégration d'Euler n'est jamais utilisé !

Pour de meilleurs résultats, on utilise :

- des schémas d'intégration plus efficaces
- une adaptation du pas

Utilisation de Scipy

On ne programme jamais un schéma d'intégration !

Il existe des bibliothèques pour cela...

`scipy.integrate` fournit toutes les fonctions utiles

Par exemple la fonction `odeint`

Utilisation de Scipy

```
In []: from scipy.integrate import odeint

In []: help(odeint)
Help on function odeint in module scipy.integrate.odepack:

odeint(func, y0, t, ...)
    Integrate a system of ordinary differential equations.

    Solve a system of ordinary differential equations using
    lsoda from the FORTRAN library odepack.

    Solves the initial value problem for stiff or non-stiff
    systems of first order ode-s: dy/dt = func(y, t, ...)

    ...
```

Utilisation de Scipy

On cherche à résoudre $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f\big(y(t), t\big)$

avec $y(t) = \mathrm{U_C}(t)$ et $f(\mathrm{U_C}(t), t) = \frac{1}{\mathrm{RC}}(\mathrm{E}(t) - \mathrm{U_C}(t))$

```
def f(y, t) :
    Uct = y    # y désigne ici U_C(t)
    R = 1.0e3  # Ω
    C = 1.0e-6 # F
    if t<0:
        E = 0.0
    else:
        E = 5.0 # V

    return 1.0/(R*C) * (E - Uct)
```

Utilisation de Scipy

On suppose le condensateur déchargé à $t_0 = -1\,\text{ms}$

On veut étudier la charge jusque $t = 5\,\text{ms}$

On construit un tableau de $\mathrm{N} = 61$ instants t_i entre $-1\,\text{ms}$ et $5\,\text{ms}$:

```
T = numpy.linspace(-1.0e-3, 5.0e-3, 61) # s
```

On choisit une condition initiale $\mathrm{U_C}(t_0) = 0$:

```
y0 = 0.0 # V
```

Utilisation de Scipy

On fait enfin appel à odeint :

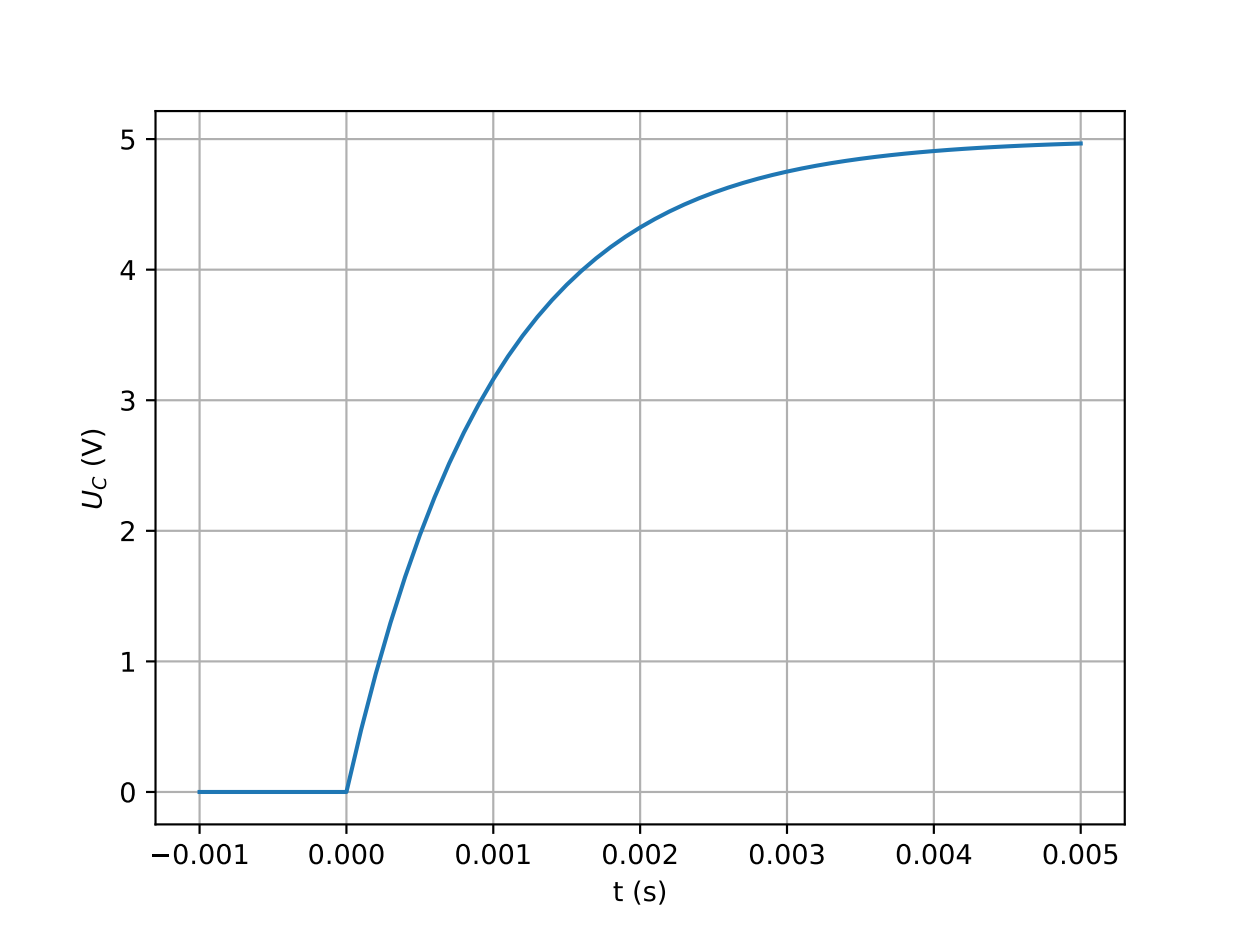
```
res = odeint(f, y0, T)
```

res est alors un numpy.ndarray de taille $\mathrm{N} \times 1$ contenant les y_i !

res[i,0] correspond à y_i, soit l'estimation de $y(t_i)$

Utilisation de Scipy

```
plt.plot(T, res)
```



Modèle de l'atmosphère isotherme

Pour l'atmosphère isotherme, $\frac{\mathrm{dP}}{\mathrm{d}z} + \rho g = 0$ et $\rho = \frac{\mathrm{PM}}{\mathrm{RT}_0}$

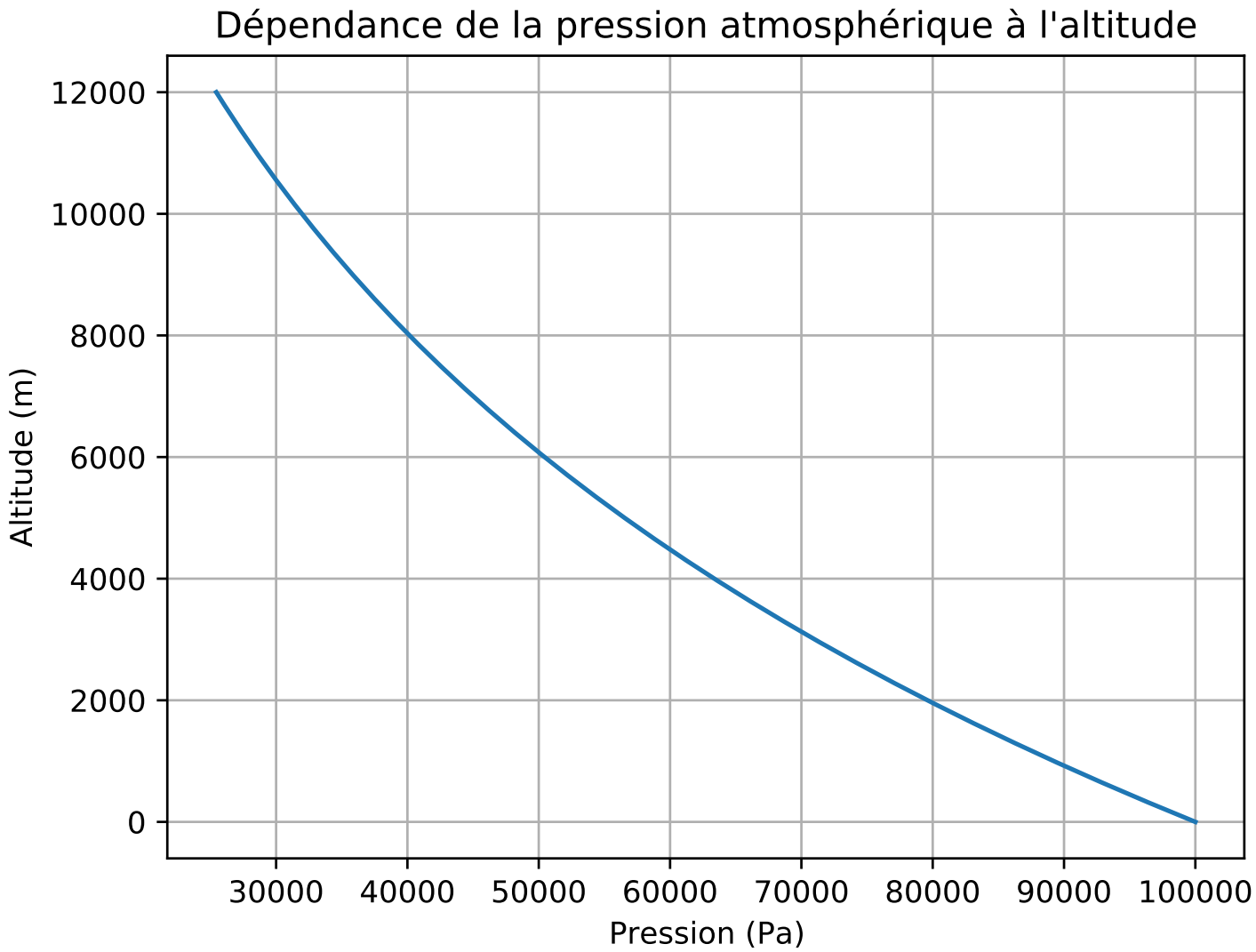
Ce qui donne $\frac{\mathrm{dP}}{\mathrm{d}z} = -\frac{\mathrm{M}g}{\mathrm{RT}_0}\mathrm{P}$

```python
P0, T0 = 1e5, 300.0

def f(p, z):
    M = 29e-3  # kg/mol
    g = 9.81   # m/s^2
    R = 8.314  # SI

    return -(M*g)/(R*T0) * p
```

Modèle de l'atmosphère isotherme

```python
Z = np.linspace(0, 12e3, 1000) # 0 -> 12 km


P = odeint(f, P0, Z)


plt.plot(P, Z)
```

Modèle de l'atmosphère isotherme



Systèmes d'équations du premier ordre

Considérons à présent le système suivant :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = f_u(u(t), v(t), t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = f_v(u(t), v(t), t) \end{cases}$$

Systèmes d'équations du premier ordre

Considérons à présent le système suivant :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = f_u(u(t), v(t), t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = f_v(u(t), v(t), t) \end{cases}$$

On a :

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} u(t) \\ v(t) \end{pmatrix}$$

Systèmes d'équations du premier ordre

Considérons à présent le système suivant :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = f_u(u(t), v(t), t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = f_v(u(t), v(t), t) \end{cases}$$

On a :

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} u(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{d}u}{\mathrm{d}t}(t) \\ \frac{\mathrm{d}v}{\mathrm{d}t}(t) \end{pmatrix}$$

Systèmes d'équations du premier ordre

Considérons à présent le système suivant :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = f_u(u(t), v(t), t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = f_v(u(t), v(t), t) \end{cases}$$

On a :

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} u(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{d}u}{\mathrm{d}t}(t) \\ \frac{\mathrm{d}v}{\mathrm{d}t}(t) \end{pmatrix} = \begin{pmatrix} f_u(u(t), v(t), t) \\ f_v(u(t), v(t), t) \end{pmatrix}$$

Systèmes d'équations du premier ordre

Considérons à présent le système suivant :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = f_u(u(t), v(t), t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = f_v(u(t), v(t), t) \end{cases}$$

On a :

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} u(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{d}u}{\mathrm{d}t}(t) \\ \frac{\mathrm{d}v}{\mathrm{d}t}(t) \end{pmatrix} = \begin{pmatrix} f_u(u(t), v(t), t) \\ f_v(u(t), v(t), t) \end{pmatrix} = f\left(\begin{pmatrix} u(t) \\ v(t) \end{pmatrix}, t\right)$$

Le système différentiel correspond donc à une équation différentielle

$$\frac{\mathrm{dY}}{\mathrm{d}t}(t) = f(\mathrm{Y}(t), t)$$

Systèmes d'équations du premier ordre

Le système différentiel correspond donc à une équation différentielle

$$\frac{\mathrm{dY}}{\mathrm{d}t}(t) = f(\mathrm{Y}(t), t)$$

où $\mathrm{Y}(t) = \begin{pmatrix} u(t) \\ v(t) \end{pmatrix}$ est un *vecteur*

Systèmes d'équations du premier ordre

Le système différentiel correspond donc à une équation différentielle

$$\frac{\mathrm{dY}}{\mathrm{d}t}(t) = f(\mathrm{Y}(t), t)$$

où $\mathrm{Y}(t) = \begin{pmatrix} u(t) \\ v(t) \end{pmatrix}$ est un *vecteur*

et $f(\mathrm{Y}(t), t) = \begin{pmatrix} f_u(u(t), v(t), t) \\ f_v(u(t), v(t), t) \end{pmatrix}$

Systèmes d'équations du premier ordre

Prenons un exemple :

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = 2u(t) + v(t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = -u(t) + 2v(t) \end{cases}$$

On souhaite déterminer $u(t)$ et $v(t)$ sur $[0,1]$

avec les conditions « initiales » $u(0) = 3$ et $v(0) = 4$

Systèmes d'équations du premier ordre

À partir du système

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}(t) = 2u(t) + v(t) \\ \dfrac{\mathrm{d}v}{\mathrm{d}t}(t) = -u(t) + 2v(t) \end{cases}$$

on construit la fonction $f(\mathrm{Y}(t), t)$:

```
def f(Y, t) :
    u, v = Y                    # On extrait u et v de Y

    # On retourne le vecteur (liste) des dérivées
    return [2*u+v, -u+2*v]
```

Systèmes d'équations du premier ordre

Puis on fait appel à odeint :

```
T = numpy.linspace(0.0, 1.0, 50)

Y0 = [3.0, 4.0]  # Conditions initiales

res = odeint(f, Y0, T)
```

res est un numpy array de taille 50×2

res[i] donne (u_i, v_i)

res[i,0] donne u_i et res[i,1] donne v_i

On peut aisément extraire l'ensemble des u_i avec :

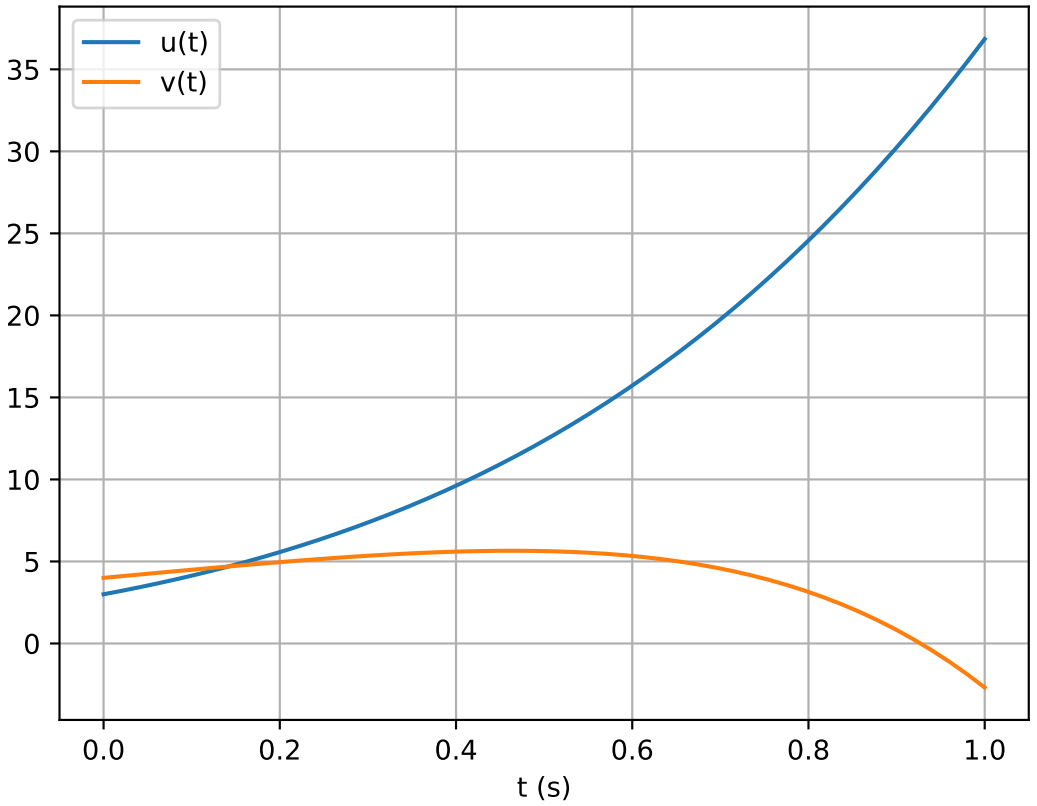
```
U = res[:,0]
```

De même pour les v_i :

```
V = res[:,1]
```

Systèmes d'équations du premier ordre

```
plt.plot(T, U, label=r"u(t)")
plt.plot(T, V, label=r"v(t)") ...
```



Remarque

L'implémentation Python déjà vue reste correcte avec numpy !

```python
def EulerExplicite(f, y0, T) :  # T = [t_0, t_1, ...]
    Y = [ y0 ]                  # Y va contenir les y_i

    for i in range(1, len(T)) :
        # dt = t_i - t_{i-1}
        dt = T[i] - T[i-1]

        # Calcul de y_i à partir de y_{i-1}
        yi = Y[i-1] + f(Y[i-1], T[i-1]) * dt
        Y.append(yi)

    return Y
```

Équations différentielles du second ordre

En mécanique, les équations sont typiquement de la forme $\ddot{z} = f(\dot{z}, z, t)$

on décrit généralement l'état d'un système avec

- sa position $z(t)$
- sa vitesse $v_z(t) = \dot{z}(t) = \frac{\mathrm{d}z}{\mathrm{d}t}(t)$

Le P.F.D. donne l'évolution de cet état

Cas du pendule

L'état est $(\theta, \dot{\theta})$, gouverné par

$$m\mathrm{L}^2 \frac{\mathrm{d}\dot{\theta}}{\mathrm{d}t} = -mg\mathrm{L}\sin(\theta)$$

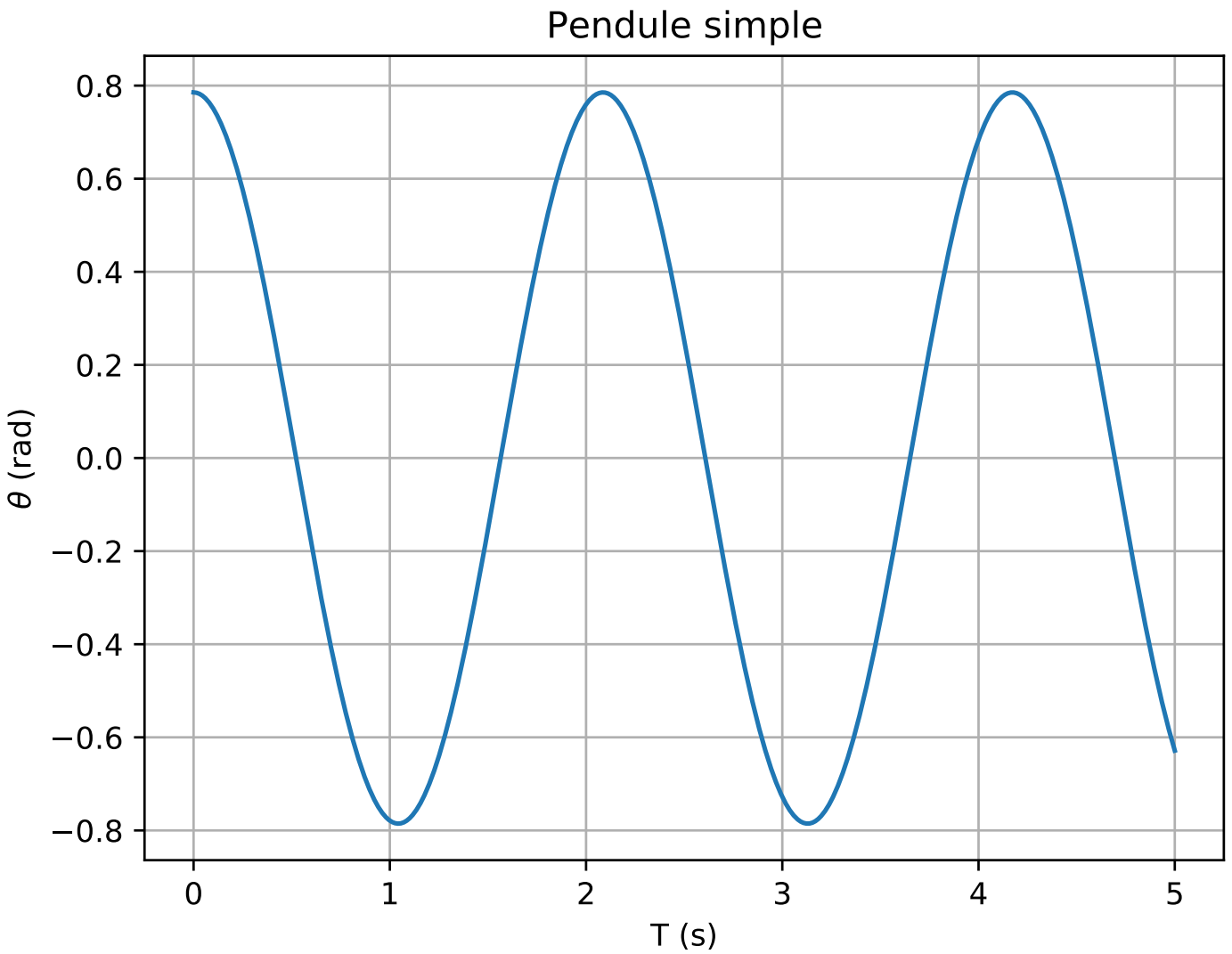
```
def f(Y, t):
    theta, dtheta = Y

    g = 9.81 # m/s^2
    L = 1.0  # m

    d2theta = -g/L * np.sin(theta)

    return [dtheta, d2theta]
```

Cas du pendule

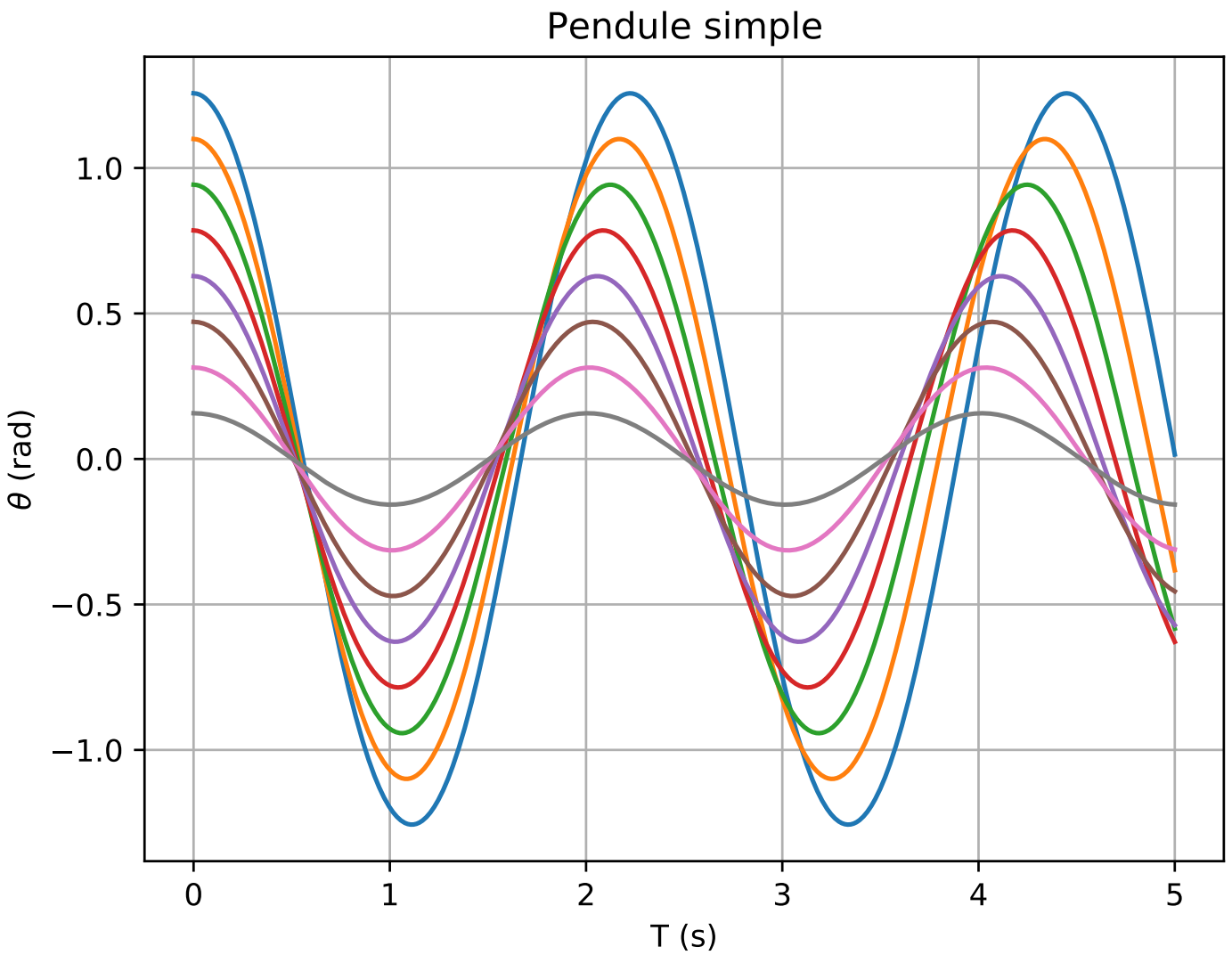
Puis on résout :

```
T = np.linspace(0.0, 5.0, 1000)

res = odeint(f, [np.pi/4, 0.0], T)

Theta = res[:,0]
DTheta = res[:,1]

plt.plot(T, Theta)
```

Cas du pendule



Cas du pendule : perte d'isochronisme



Mouvements à force centrale

Pour un corps céleste soumis à l'attraction solaire :

$$m\vec{a} = -\frac{\mathscr{G}\mathrm{M}_\odot m}{\mathrm{OM}^3}\overrightarrow{\mathrm{OM}}$$

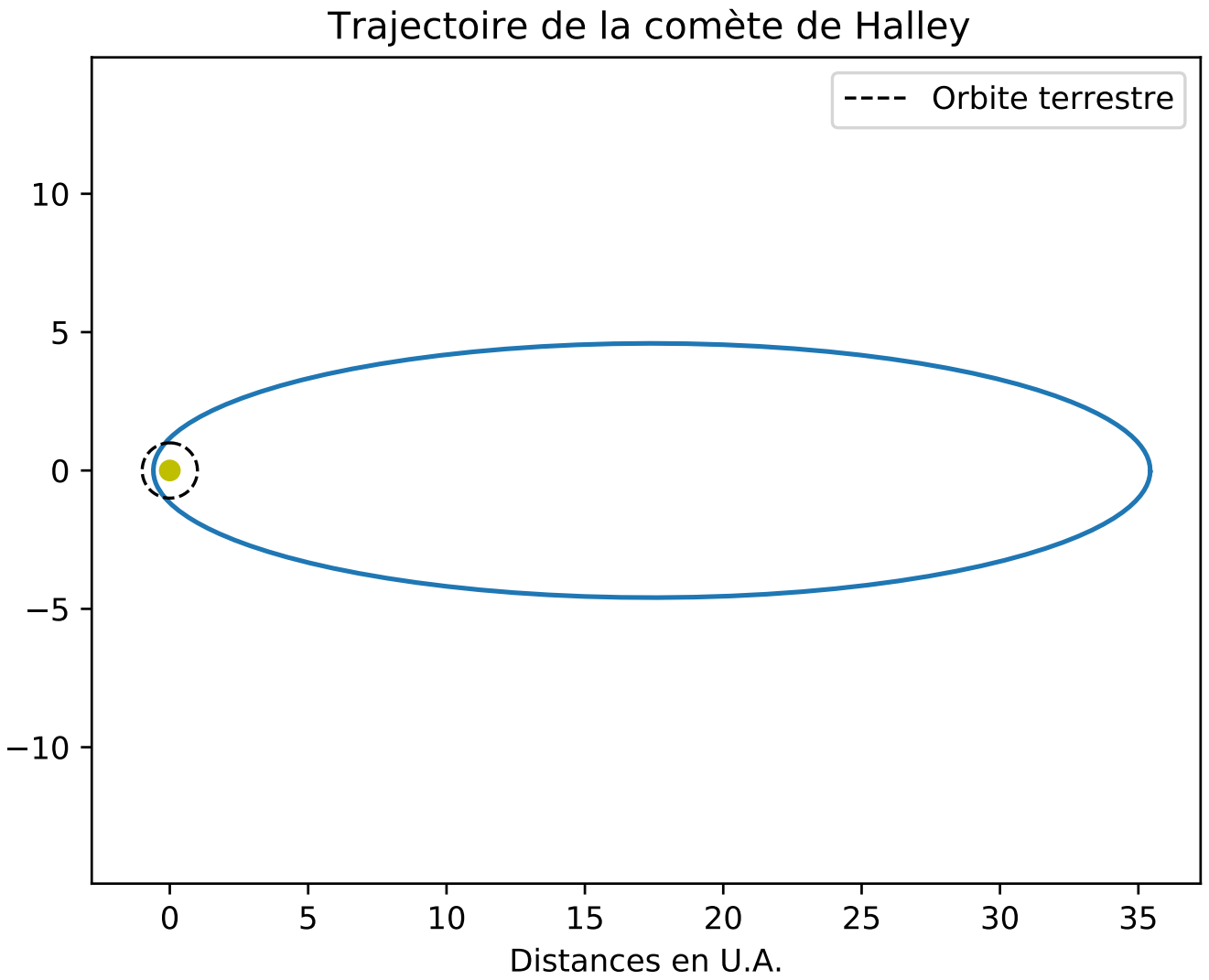
```python
def f(Y, t):
    x, y, vx, vy = Y

    G = 6.67428e-11 # m^3/kg/s^2
    M = 1.9884e30   # kg

    dist2 = x**2.0 + y**2.0
    ax = -G*M / dist2**1.5 * x
    ay = -G*M / dist2**1.5 * y

    return [vx, vy, ax, ay]
```

Mouvements à force centrale

Puis on résoud (comète de Halley, période 76.09 années) :

```
T = np.linspace(0.0, 76.09*365.25*86400, 5000)

res = odeint(f, [5.3e12, 0.0, 0.0, 910.0], T)

X = res[:,0]
Y = res[:,1]

plt.plot(X, Y)
```

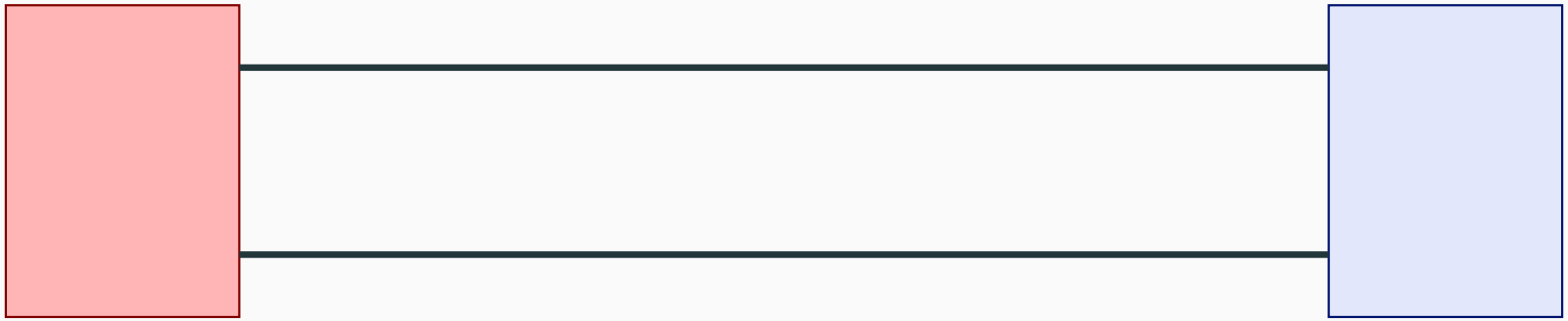
Mouvements à force centrale



Diffusion thermique

On considère une barre de longueur L, section S

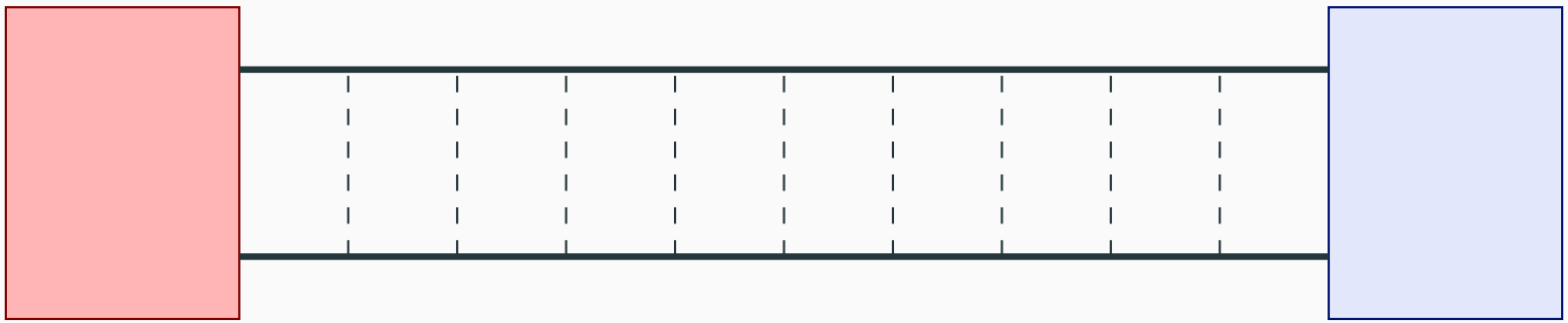
Masse volumique ρ, capacité calorifique massique c_m, conductivité λ



Diffusion thermique

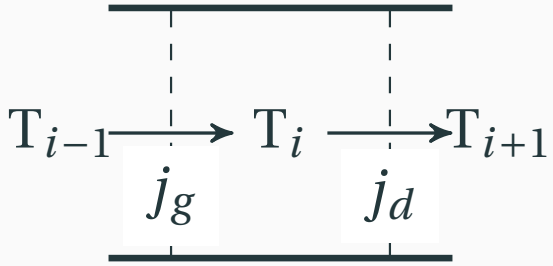
On considère une barre de longueur L, section S

Masse volumique ρ, capacité calorifique massique c_m, conductivité λ



Diffusion thermique

N subdivisions, T uniforme dans chaque « morceau »



$$\frac{dU_i}{dt} = j_g S - j_d S = \lambda \frac{T_{i-1} - T_i}{dx} S + \lambda \frac{T_{i+1} - T_i}{dx} S$$

$$\frac{dT_i}{dt} = \frac{1}{C_i} \frac{dU_i}{dt} = \frac{1}{\rho S\, dx c_m} \frac{dU_i}{dt} = \frac{\lambda}{\rho\, dx c_m} \frac{T_{i+1} + T_{i-1} - 2T_i}{dx}$$

Donc $\frac{dT_i}{dt} = \mathscr{A}(T_{i+1} + T_{i-1} - 2T_i)$ où $\mathscr{A} = \frac{\lambda}{\rho\, dx^2 c_m}$

Diffusion thermique

On déclare les coefficients :

```
L = 1.0       # longueur, m
S = 0.01      # section, m^2

lmbd = 380.0 # conductivité, W/(m.K)
cm = 380.0   # capacité cal. mass., J/(kg.K)
rho = 8.96e3 # masse volumique, kg/m^3
cc = 5.0e3   # coeff. contact, W/(m^2.K)

N = 50        # nombre de subdivisions
```

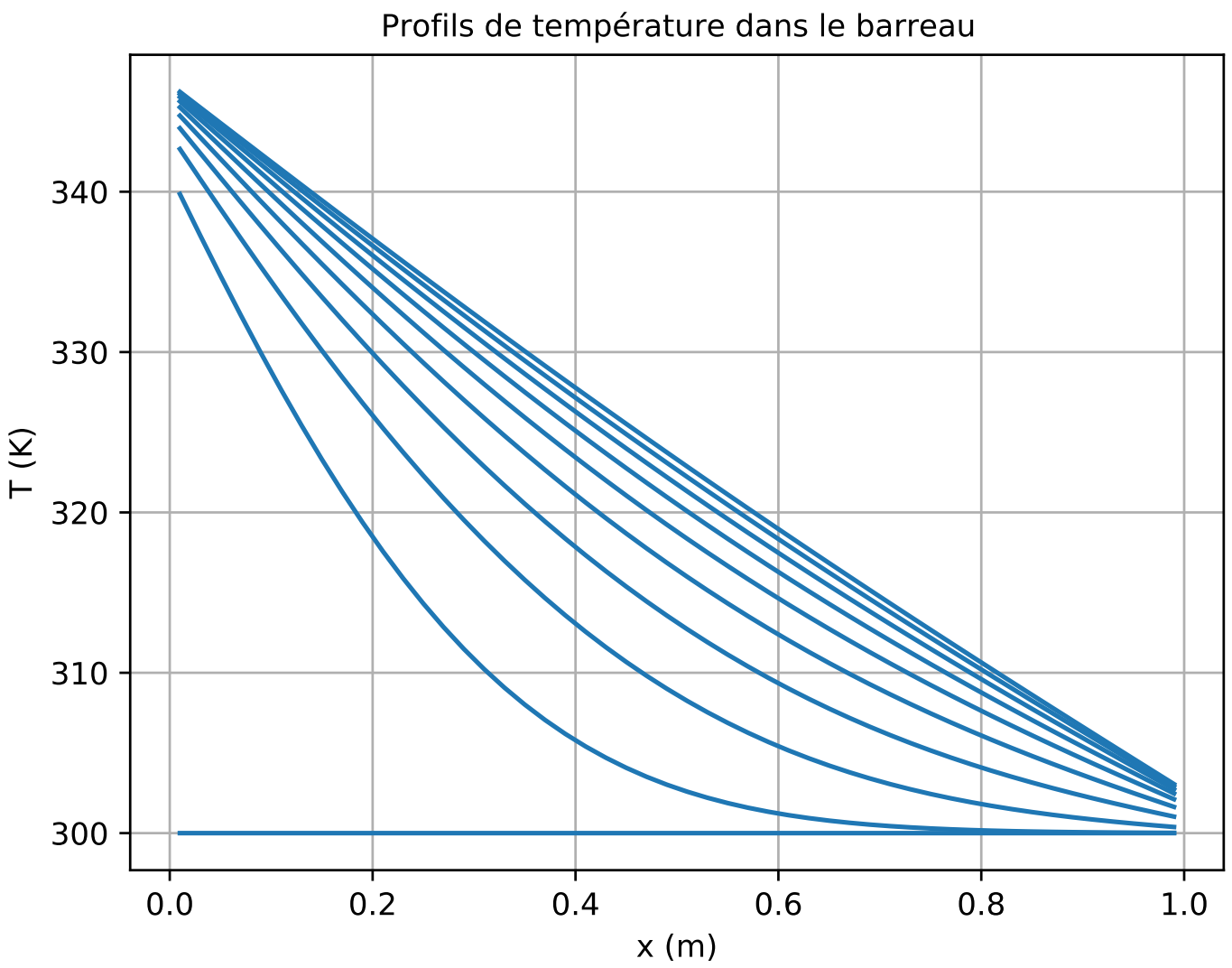
Diffusion thermique

```python
def f(Y, t):
    Temps = Y
    dTemps = np.zeros(Temps.shape)

    dx = L/N
    A = lmbd / (rho * dx**2.0 * Cm)
    B = cc / (rho * dx * cm)

    for i in range(1, len(Temps)-1):
        dTemps[i] = A * (Temps[i+1]+Temps[i-1]-2*Temps[i])

    Tg = 300 if t<0 else 350
    Td = 300

    dTemps[0]   = A*(Temps[1]-Temps[0])     + B*(Tg-Temps[0])
    dTemps[N-1] = A*(Temps[N-2]-Temps[N-1]) + B*(Td-Temps[N-1])

    return dTemps
```

Diffusion thermique

```
T_ini = 300*np.ones(N)

T = np.linspace(0, 3600, 100)

res = itg.odeint(f, T_ini, T)

for i in range(0, 100, 10):
    plt.plot(X, res[i], color="tab:blue")
```

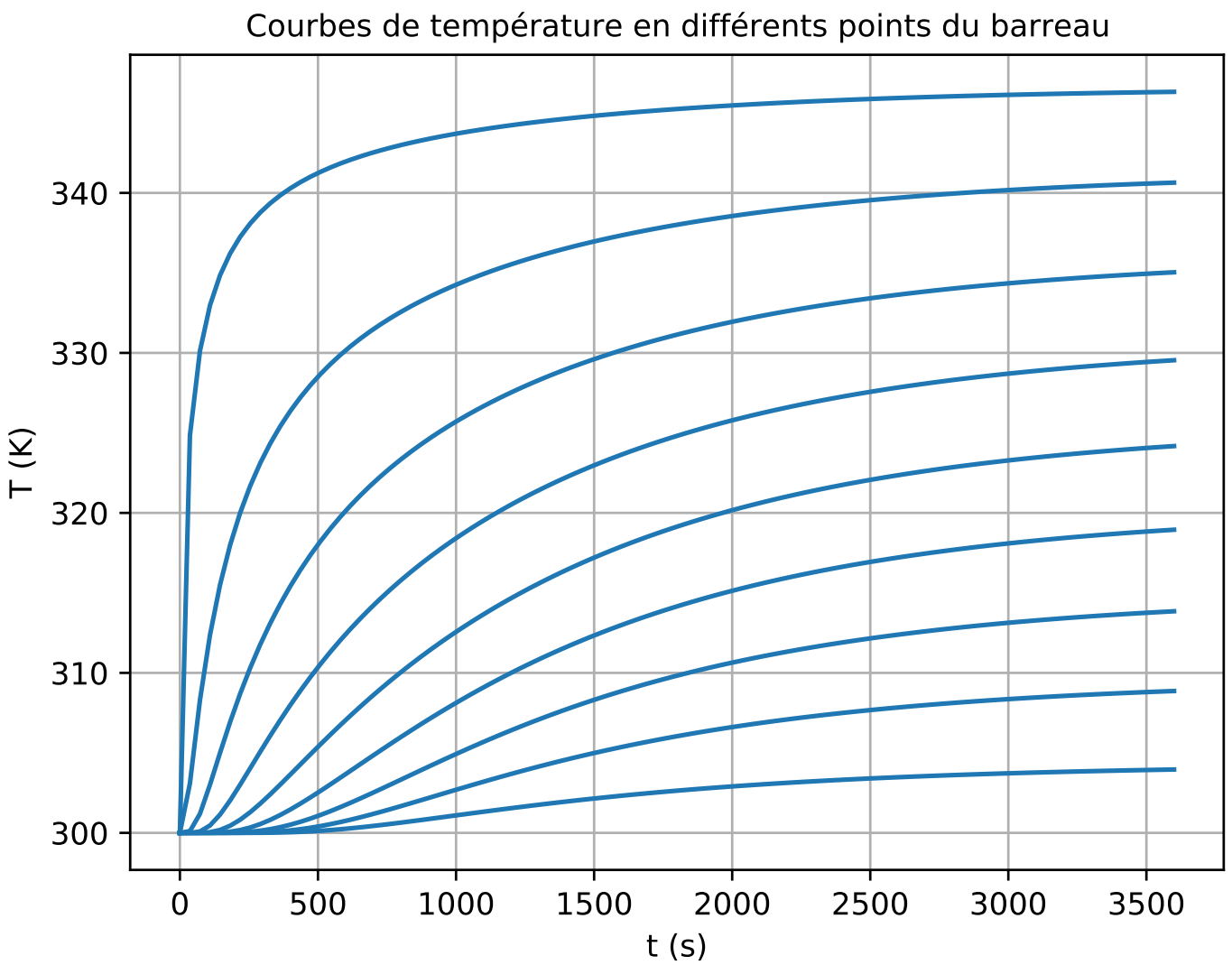
Diffusion thermique



Diffusion thermique

```
...


for i in range(0, N, N//8):
    plt.plot(T, res[:,i], color="tab:blue")
```

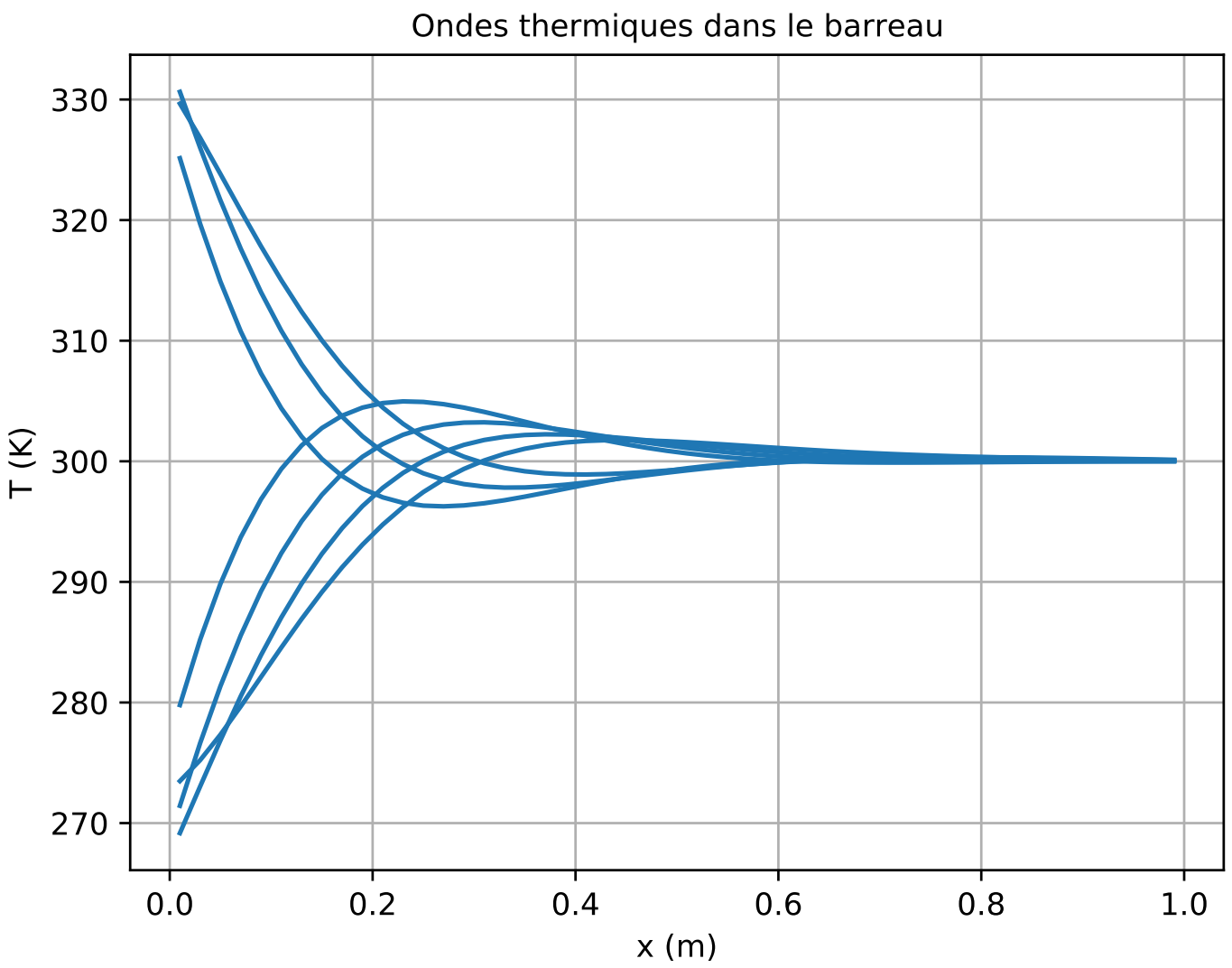
Diffusion thermique



Diffusion thermique

```
def f(Y, t):
    ...
    Tg = 300 + 50*np.sin(t/100)
    ...


...

for i in range(50, 100, 9):
    plt.plot(X, res[i], color="tab:blue")
```

Diffusion thermique



Bilan

Pour résoudre un problème soumis à une équation différentielle

- on met le problème sous la forme $\frac{\mathrm{dY}}{\mathrm{d}t}(t) = \mathrm{F}(\mathrm{Y}(t), t)$
- on définit $\mathrm{F}(\mathrm{Y}(t), t)$ et $\mathrm{Y}(t_0)$
- **on confie le tout à `odeint` ou équivalent**
- ...
- profit !

Déroulement de la présentation

Principe du pivot de Gauss

Appliquer différentes transformations à une matrice :

- transvection
- dilatation
- permutation

But : donner à la matrice une forme particulière

Le programme ne concerne que les matrices carrées inversibles

Rendre la matrice triangulaire supérieure

$$\begin{bmatrix} 0 & 8 & 8 & 7 & 8 \\ 7 & 8 & 7 & 3 & 5 \\ 3 & 8 & 6 & 1 & 5 \\ 5 & 6 & 2 & 2 & 2 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 & 8 & 7 & 3 & 5 \\ 0 & 8 & 8 & 7 & 8 \\ 0 & 0 & 6 & 0.125 & 3 \\ 0 & 0 & 0 & -4.253 & -0.929 \\ 0 & 0 & 0 & 0 & -0.143 \end{bmatrix}$$

Rendre la matrice diagonale

$$\begin{bmatrix} 0 & 8 & 8 & 7 & 8 \\ 7 & 8 & 7 & 3 & 5 \\ 3 & 8 & 6 & 1 & 5 \\ 5 & 6 & 2 & 2 & 2 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & -4.253 & 0 \\ 0 & 0 & 0 & 0 & -0.143 \end{bmatrix}$$

Transformer en matrice identité

$$\begin{bmatrix} 0 & 8 & 8 & 7 & 8 \\ 7 & 8 & 7 & 3 & 5 \\ 3 & 8 & 6 & 1 & 5 \\ 5 & 6 & 2 & 2 & 2 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Dilatation

« Multiplier une ligne par $a \neq 0$ »

$$\mathscr{L}_i \leftarrow a \times \mathscr{L} i$$

```python
def dilatation(M, i, a) :
        M[i] = a*M[i]
```

```python
def dilatation(M, i, a) :
        M[i] *= a
```

Transvection

« Ajouter a fois une ligne à une autre »

$$\mathscr{L}_i \leftarrow \mathscr{L}_i + a \times \mathscr{L} j$$

```
def transvection(M, i, j, a) :
        M[i] = M[i] + a*M[j]
```

```
def transvection(M, i, j, a) :
        M[i] += a*M[j]
```

Permutation

« Échanger deux lignes »

$$\mathscr{L}_i \leftrightarrow \mathscr{L}\, j$$

```python
def permutation(M, i, j) :
        M[i], M[j] = M[j], M[i].copy()
```

Note sur `a, b = expr1, expr2`

En Python, `a, b = expr1, expr2` correspond à

```
tmp1 = expr1 # On évalue les expressions dans l'ordre
tmp2 = expr2 # et on mémorise les résultats

a = tmp1     # On associe les noms aux résultats
b = tmp2     # là encore dans l'ordre
```

Note sur `a, b = expr1, expr2`

Si l'on écrit `M[i], M[j] = M[j], M[i]` avec `M` `numpy.array` :

```python
tmp1 = M[j]
tmp2 = M[i]  # tmp2 est une vue sur M

M[i] = tmp1  # cette mutation affecte tmp2 !
M[j] = tmp2
```

D'où `M[i], M[j] = M[j], M[i].copy()`

Avec des listes de listes

Transvection :

```
def transvection(M, i, j, a) :
    M[i] = [ M[i][k] + a*M[j][k] for k in range(len(M[i])) ]
```

Dilatation :

```
def dilatation(M, i, a) :
    M[i] = [ a*M[i][k] for k in range(len(M[i])) ]
```

Permutation :

```
def permutation(M, i, j) :
    M[i], M[j] = M[j], M[i]
```

Méthode

Conseil : ne PAS suivre le cours de mathématiques, une seule passe

Pour chaque colonne i :

- s'assurer d'un coefficient non-nul sur la ligne i (sinon, utiliser une permutation avec une ligne **inférieure**)
- normaliser le coefficient `M[i,i]` par une dilatation (facultatif)
- rendre nuls les coefficients `M[j,i]` ($j > i$ ou $j \neq i$) avec des transvections

Pour des raisons de stabilité/précision numérique, on ne se satisfait pas d'un coefficient non-nul sur la diagonale, on le veut le plus grand possible (en valeur absolue)

Méthode

Pour trouver l'index de la ligne du plus grand coefficient (en valeur absolue) dans la colonne i **et sous la diagonale** :

```python
def largest(M, i) :
    maxabs, j_maxabs = abs(M[i, i]), i

    for j in range(i+1, len(M)) :
        if abs(M[j, i]) > maxabs :
            maxabs, j_maxabs = abs(M[j, i]), j

    return j_maxabs
```

```python
def largest(M, i) :
    return i+numpy.argmax(numpy.abs(M[i:,i]))
```

Triangulaire supérieure

```python
def pivot(M) :
    h, l = M.shape

    for col in range(l) :
        lgn = largest(M, col)
        if lgn != col :
            permutation(M, lgn, col)

        pivot = M[col,col]

        for lgn in range(col+1, h) :
            if M[lgn, col] != 0 :
                transvection(M, lgn, col, -M[lgn,col]/pivot)
```

Diagonale

```
def pivot(M) :
    h, l = M.shape

    for col in range(l) :
        lgn = largest(M, col)
        if lgn != col :
            permutation(M, lgn, col)

        pivot = M[col,col]

        for lgn in range(h) :
            if lgn != col and M[lgn, col] != 0 :
                transvection(M, lgn, col, -M[lgn,col]/pivot)
```

Identité

```python
def pivot(M) :
    h, l = M.shape

    for col in range(l) :
        lgn = largest(M, col)
        if lgn != col :
            permutation(M, lgn, col)

        dilatation(M, col, 1/M[col,col])

        for lgn in range(h) :
            if lgn != col and M[lgn, col] != 0 :
                transvection(M, lgn, col, -M[lgn,col])
```

Colonne 0

$$\begin{bmatrix} 0 & 8 & 8 & 7 & 8 \\ 7 & 8 & 7 & 3 & 5 \\ 3 & 8 & 6 & 1 & 5 \\ 5 & 6 & 2 & 2 & 2 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1.143 & 1 & 0.429 & 0.714 \\ 0 & 8 & 8 & 7 & 8 \\ 0 & 4.571 & 3 & -0.286 & 2.857 \\ 0 & 0.286 & -3 & -0.143 & -1.571 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix}$$

$$\mathscr{L}_0 \leftrightarrow \mathscr{L}_1$$

$$\mathscr{L}_0 \leftarrow \frac{1}{7}\mathscr{L}_0$$

$$\mathscr{L}_2 \leftarrow \mathscr{L}_2 - 3\mathscr{L}_0$$

$$\mathscr{L}_3 \leftarrow \mathscr{L}_3 - 5\mathscr{L}_0$$

Colonne 1

$$\begin{bmatrix} 1 & 1.143 & 1 & 0.429 & 0.714 \\ 0 & 8 & 8 & 7 & 8 \\ 0 & 4.571 & 3 & -0.286 & 2.857 \\ 0 & 0.286 & -3 & -0.143 & -1.571 \\ 0 & 1 & 7 & 1 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & -0.143 & -0.571 & -0.429 \\ 0 & 1 & 1 & 0.875 & 1 \\ 0 & 0 & -1.571 & -4.286 & -1.714 \\ 0 & 0 & -3.286 & -0.393 & -1.857 \\ 0 & 0 & 6 & 0.125 & 3 \end{bmatrix}$$

$$\mathscr{L}_1 \leftarrow \frac{1}{8}\mathscr{L}_1$$

$$\mathscr{L}_0 \leftarrow \mathscr{L}_0 - 1.143\mathscr{L}_1$$

$$\mathscr{L}_2 \leftarrow \mathscr{L}_2 - 4.571\mathscr{L}_1$$

$$\mathscr{L}_3 \leftarrow \mathscr{L}_3 - 0.286\mathscr{L}_1$$

$$\mathscr{L}_4 \leftarrow \mathscr{L}_4 - 1\mathscr{L}_1$$

Colonne 2

$$\begin{bmatrix} 1 & 0 & -0.143 & -0.571 & -0.429 \\ 0 & 1 & 1 & 0.875 & 1 \\ 0 & 0 & -1.571 & -4.286 & -1.714 \\ 0 & 0 & -3.286 & -0.393 & -1.857 \\ 0 & 0 & 6 & 0.125 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & -0.568 & -0.357 \\ 0 & 1 & 0 & 0.854 & 0.5 \\ 0 & 0 & 1 & 0.021 & 0.5 \\ 0 & 0 & 0 & -0.324 & -0.214 \\ 0 & 0 & 0 & -4.253 & -0.929 \end{bmatrix}$$

$$\mathscr{L}_2 \leftrightarrow \mathscr{L}_4$$

$$\mathscr{L}_2 \leftarrow \frac{1}{6}\mathscr{L}_2$$

$$\mathscr{L}_0 \leftarrow \mathscr{L}_0 - (-0.143)\mathscr{L}_2$$

$$\mathscr{L}_1 \leftarrow \mathscr{L}_1 - 1\mathscr{L}_2$$

$$\mathscr{L}_3 \leftarrow \mathscr{L}_3 - (-3.286)\mathscr{L}_2$$

$$\mathscr{L}_4 \leftarrow \mathscr{L}_4 - (-1.571)\mathscr{L}_2$$

Colonne 3

$$\begin{bmatrix} 1 & 0 & 0 & -0.568 & -0.357 \\ 0 & 1 & 0 & 0.854 & 0.5 \\ 0 & 0 & 1 & 0.021 & 0.5 \\ 0 & 0 & 0 & -0.324 & -0.214 \\ 0 & 0 & 0 & -4.253 & -0.929 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & -0.233 \\ 0 & 1 & 0 & 0 & 0.314 \\ 0 & 0 & 1 & 0 & 0.495 \\ 0 & 0 & 0 & 1 & 0.218 \\ 0 & 0 & 0 & 0 & -0.143 \end{bmatrix}$$

$$\mathscr{L}_3 \leftrightarrow \mathscr{L}_4$$

$$\mathscr{L}_3 \leftarrow \frac{1}{-4.253}\mathscr{L}_3$$

$$\mathscr{L}_0 \leftarrow \mathscr{L}_0 - (-0.568)\mathscr{L}_3$$

$$\mathscr{L}_1 \leftarrow \mathscr{L}_1 - 0.854\mathscr{L}_3$$

$$\mathscr{L}_3 \leftarrow \mathscr{L}_3 - 0.021\mathscr{L}_3$$

$$\mathscr{L}_4 \leftarrow \mathscr{L}_4 - (-0.324)\mathscr{L}_3$$

Colonne 4

$$\begin{bmatrix} 1 & 0 & 0 & 0 & -0.233 \\ 0 & 1 & 0 & 0 & 0.314 \\ 0 & 0 & 1 & 0 & 0.495 \\ 0 & 0 & 0 & 1 & 0.218 \\ 0 & 0 & 0 & 0 & -0.143 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathscr{L}_4 \leftarrow \frac{1}{-0.143}\mathscr{L}_4$$

$$\mathscr{L}_0 \leftarrow \mathscr{L}_0 - (-0.233)\mathscr{L}_4$$

$$\mathscr{L}_1 \leftarrow \mathscr{L}_1 - 0.314\mathscr{L}_4$$

$$\mathscr{L}_2 \leftarrow \mathscr{L}_2 - 0.495\mathscr{L}_4$$

$$\mathscr{L}_3 \leftarrow \mathscr{L}_3 - (-0.218)\mathscr{L}_4$$

Dans quel but ?

- Résoudre $\mathrm{AX} = \mathrm{B}$

On transforme A en $\mathbb{1}$.

On effectue les mêmes manipulations sur B que sur A.

B correspond alors à la solution X.

- Calculer A^{-1}

On transforme A en $\mathbb{1}$.

On effectue les mêmes manipulations sur $\mathbb{1}$ que sur A.

La matrice obtenue est A^{-1}.

- Calculer $\mathrm{Det(A)}$

On transforme A en triangulaire supérieure.

On n'utilise que des transvections avec $i \neq j$ qui ne changent pas le déterminant et des permutations qui le multiplient par -1.

Complexité

Toutes les opérations élémentaires sont $\mathrm{O}(n)$

Pour *chaque* colonne :

- possiblement 1 recherche en $\mathrm{O}(n)$ et 1 permutation en $\mathrm{O}(n)$
- éventuellement 1 normalisation en $\mathrm{O}(n)$
- jusqu'à $n-1$ transvections en $\mathrm{O}(n)$

Donc $\mathrm{O}(n^2)$ pour chaque colonne.

Soit $n \times \mathrm{O}(n^2) = \mathrm{O}(n^3)$ pour l'ensemble de la matrice.

Plus $\mathrm{O}(n^2)$ pour les opérations sur B ou $\mathrm{O}(n^3)$ sur l'identité, donc cela reste $\mathrm{O}(n^3)$ pour la résolution d'un système linéaire ou le calcul d'une inverse.