Equations différentielles ordinaires

Dans ce chapitre

Algorithmes

- Méthode d'Euler
- Méthode de Runge-Kutta classique

Généralités

- Application aux équations de mouvement des systèmes en mécanique classique

Equations différentielles ordinaires

Une équation différentielle ordinaire du premier ordre est une équation dont l'inconnue est une fonction d'un seul paramètre $x(t)$ et qui implique la dérivée de cette fonction $\dot{x}(t)$.

On s'intéressera ici aux problemes de Cauchy (ou "problèmes aux valeurs initiales") où on donne une EDO et une condition initiale $x(0) = x_0$.

Théorème de Cauchy-Lipschitz : On donne l'EDO

$$\dot{x}(t) = f(x(t), t)$$

avec f une fonction suffisamment régulière (par exemple, dérivable par rapport à son premier argument). Alors, pour tout $x_0 \in \mathbb{R}$, il existe un voisinage U de $t = 0$ et une fonction unique x sur U qui vérifie l'EDO ainsi que la condition initiale $x(0) = x_0$.

Notre objectif sera de trouver une approximation numérique de cette fonction solution = une liste de valeurs de fonction approximatives $x(t_1)$, $x(t_2)$, ..., $x(t_n)$.

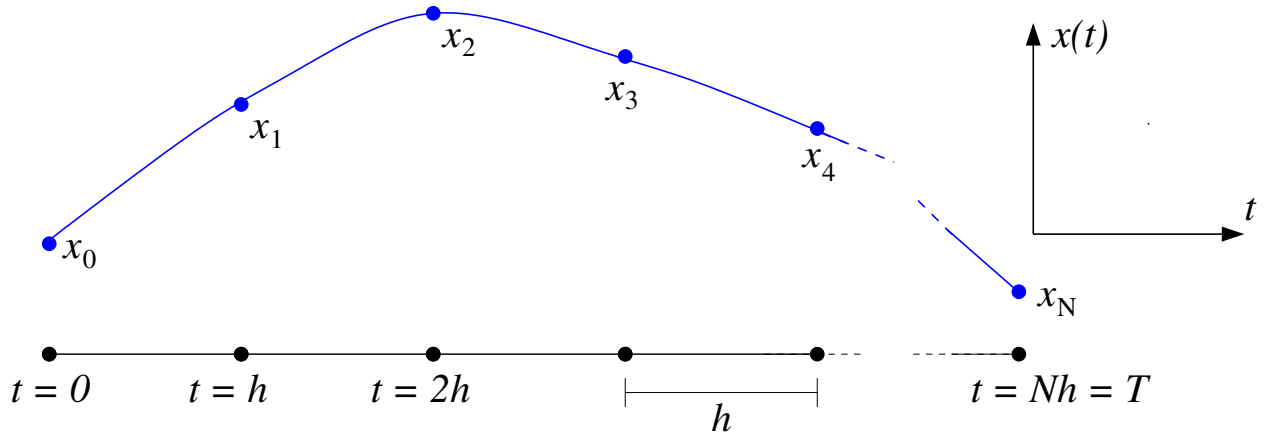
Equations différentielles ordinaires

Motivation physique importante : Les équations de mouvement $\vec{F} = m\vec{a}$ en mécanique peuvent s'écrire comme un système d'EDO du premier ordre. Les fonctions inconnues sont les positions et les vitesses (ou plus généralement les coordonnées généralisées et leurs moments conjugués).

Avec les positions et vitesses donnés à $t = 0$, on pourra les trouver pour tout t.

Méthode d'Euler

Pour l'EDO $\dot{x}(t) = f(x(t), t)$ avec la condition initiale $x(0) = x_0$, on souhaite obtenir la solution sur tout un intervalle $[0, T]$, c.-à-d. une liste de N valeurs de fonction $x_1 = x(h), x_2 = x(2h), x_3 = x(3h), \ldots, x_N = x(Nh)$ avec $Nh = T$.



Méthode d'Euler : Développement limité pour h suffisamment petit,

$$x(h) = x(0) + \dot{x}(0)\, h + \frac{1}{2}\ddot{x}(0)\, h^2 + \frac{1}{3!}\dddot{x}(0)\, h^3 + \ldots$$

On néglige les termes $\mathcal{O}(h^2)$ et supérieurs ; on substitue l'EDO $\dot{x}(0) = f(x(0), 0)$ et la condition initiale $x(0) = x_0$:

$$x_1 = x_0 + h\, f(x_0, 0)\,.$$

De même : Une fois x_n connu, on obtient x_{n+1} par développement limité et substitution,

$$x_{n+1} = x_n + \dot{x}_n\, h = x_n + h\, f(x_n, nh)\,.$$

Méthode d'Euler

On obtient ainsi successivement les x_1, x_2, x_3,... $x_N = x(T)$:

$$\boxed{x_{n+1} = x_n + h\, f(x_n, nh)}$$

```
# Résoudre l'EDO dx/dt = f(x(t), t) avec la méthode d'Euler
#
# Arguments:
# f = fonction à deux arguments = membre de droite de l'EDO
# x0 = x(0) condition initiale
# h = pas d'incrément en t
# N = nombre de points à calculer
#
# Renvoie une liste de N+1 valeurs [x(0),...,x(Nh)]
def euler(f, x0, h, N):
    x = [x0] # une liste qui contient initialement x0
    xn = x0
    for n in range(N):
        xn += h * f(xn, h*n)
        x += [xn] # ajouter xn à la liste des x
    return x
```

Méthode d'Euler : Un simple exemple

Pour p.ex. $f(x(t), t) = x(t) + t$ et $x_0 = 0$:

$$\dot{x}(t) = x(t) + t\,, \qquad x(0) = 0\,.$$

Solution analytique :

$$x(t) = e^t - t - 1\,.$$

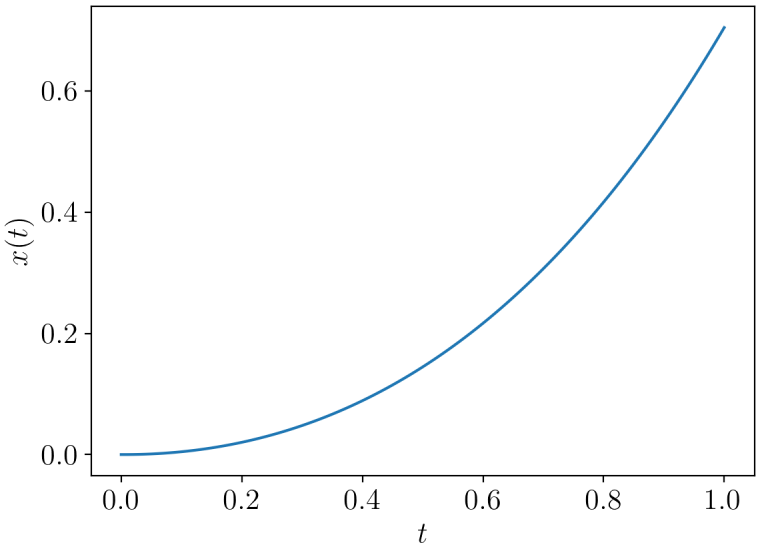
Solution numérique sur $[0,\, 1]$ avec incrément $h = 10^{-2}$:

```
from euler import euler  # la fct. euler du fichier euler.py

def f(x, t):
    return x + t

solution = euler(f, 0.0, 1.E-2, 100)
```

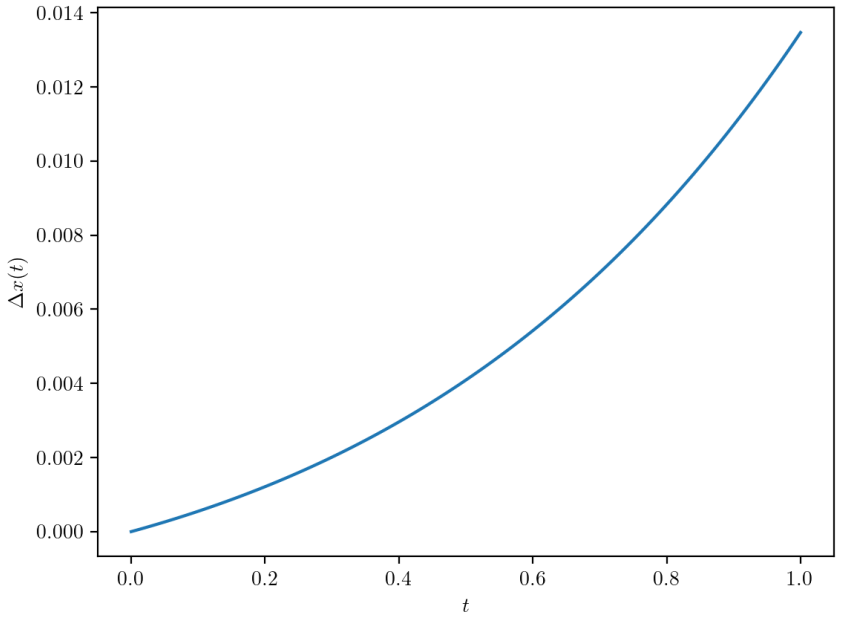
Affichage :

```
import matplotlib.pyplot as plt

plt.plot(np.linspace(0, 1, 101), solution)
plt.show()
```

Méthode d'Euler : Un simple exemple



Solution numérique $x(t)$



$\Delta x(t) = |x_{\text{numérique}}(t) - x_{\text{analytique}}(t)|$

- On s'aperçoit que l'erreur numérique Δx à $t = T = 1$ est ≈ 0.014 et alors du même ordre que $h = 0.01$. Explication : à chaque pas on néglige des termes $\mathcal{O}(h^2)$ dans le développement limité ; il y a $N = 1/h$ pas pour arriver à $t = 1$.
- La méthode d'Euler est une méthode du premier ordre : l'erreur numérique globale est de l'ordre h.
- Pour améliorer la précision numérique par un facteur 2, il faudrait calculer $\sim 2\times$ plus de points.

Exemple physique

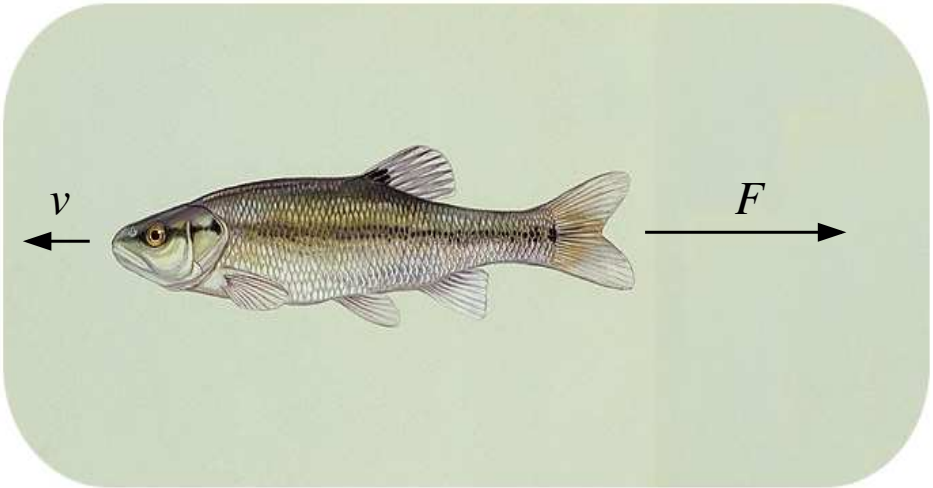
On regarde un objet qui se déplace dans un milieu fluide et qui est ralenti par une force de traînée dépendant de la vitesse, $F(v)$,

$$F(v) = f_1\, v + f_2\, v^2\,, \qquad\qquad f_i = \text{ctes.}$$

L'équation de mouvement est alors $F = ma = m\dot{v} = f_1\, v + f_2\, v^2$, ou

$$\boxed{\dot{v} = -\alpha\, v - \beta\, v^2}$$

$\alpha = -\dfrac{f_1}{m}$, $\beta = -\dfrac{f_2}{m}$ positives (traînée opposée au mouvement).



Exemple physique

$$\boxed{\dot{v} = -\alpha\, v - \beta\, v^2}$$

- Petites vitesses, écoulement laminaire : $\dot{v} = -\alpha\, v$ (loi de Stokes) avec solution $v(t) = v_0\, e^{-\alpha\, t}$.
- Grandes vitesses, écoulement turbulent : traînée $\propto v^2$, $\dot{v} = -\beta v^2$ donc $v(t) = \frac{v_0}{1+\beta\, v_0\, t}$.
- Cas général : exemple d'une équation différentielle de Bernoulli,

$$v(t) = \frac{\alpha v_0}{e^{\alpha t}(\alpha + \beta v_0) - \beta v_0}\,.$$

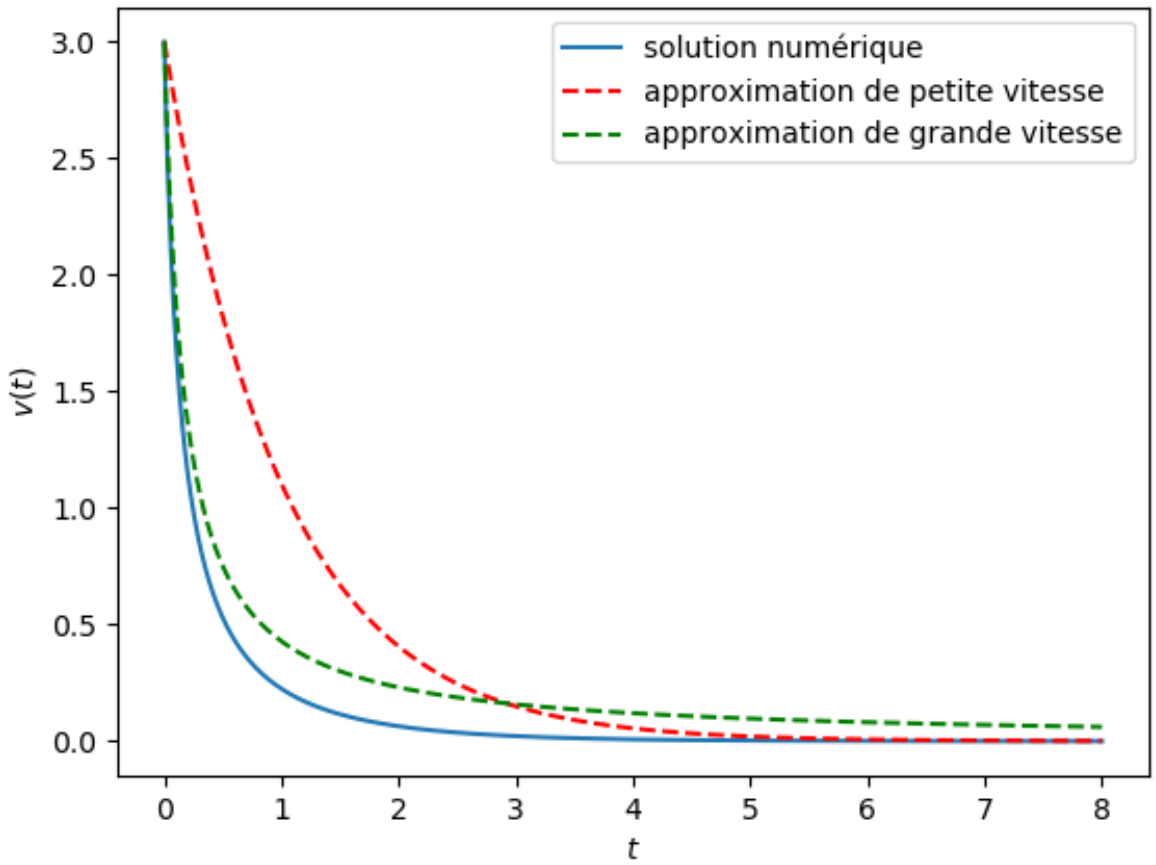
Calculons une solution numérique pour comparer avec la solution exacte.

Exemple physique

```
# Constantes physiques:
alpha = 1.
beta = .5
v0 = 2.
# Constantes numériques:
T = 5.      # intervalle de temps
N = 1000    # nombre de pas à calculer
h = T / N   # pas d'incrément

def f(v, t):
    return -alpha * v - beta * v**2

from euler import euler
solution = euler(f, v0, h, N)

import numpy as np
import matplotlib.pyplot as plt
plt.plot(np.linspace(0, T, N+1), solution)
plt.show()
```

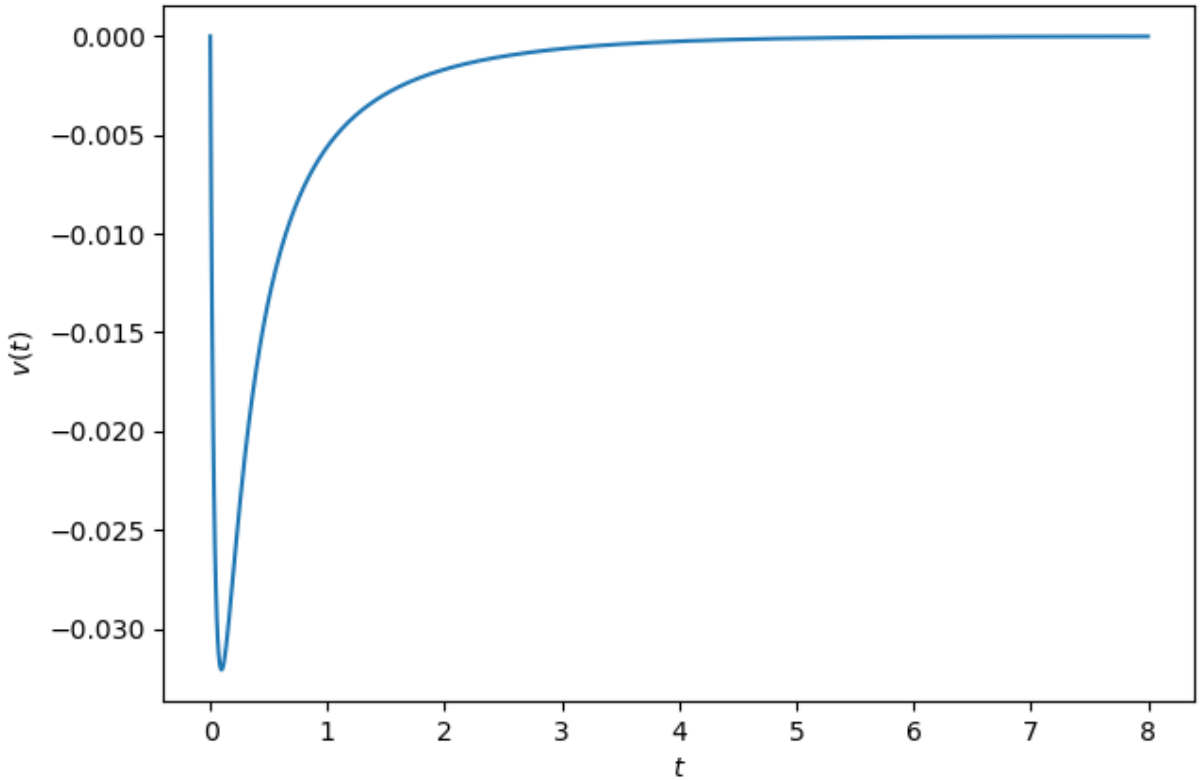
Exemple physique

Solution numérique pour $\alpha = 1$, $\beta = 2$, $v_0 = 3$ calculée avec $N = 1000$ points :



Exemple physique

Différence entre la solution numérique pour $\alpha = 1$, $\beta = 2$, $v_0 = 3$ avec $N = 1000$ points et la solution exacte $v(t) = \frac{\alpha v_0}{e^{\alpha t}(\alpha+\beta v_0)-\beta v_0}$



Systèmes d'EDO

Un système de n équations différentielles du premier ordre avec n fonctions inconnues :

$$\begin{aligned}
\dot{x}(t) &= f_x(x(t),\, y(t),\, z(t), \ldots,\, t) \\
\dot{y}(t) &= f_y(x(t),\, y(t),\, z(t), \ldots,\, t) \\
\dot{z}(t) &= f_z(x(t),\, y(t),\, z(t), \ldots,\, t) \\
&\vdots
\end{aligned}$$

Il faut n conditions initiales $x(0) = x_0,\, y(0) = y_0,\, z(0) = z_0, \ldots$ pour un problème de Cauchy bien posé.

Pour le résoudre, on se sert de la même méthode qu'avant, en regroupant les n fonctions inconnues $x,\, y,\, z, \ldots$ et les n fonctions des membres de droite $f_x,\, f_y,\, f_z, \ldots$ dans des fonctions vectorielles $\vec{u}$ et $\vec{f}$:

$$\dot{\vec{u}}(t) = \vec{f}(\vec{u}(t),\, t)$$

Après développement limité et substitution :

$$\boxed{\vec{u}(t+h) = \vec{u}(t) + h\,\vec{f}(\vec{u}(t),\, t) + \mathcal{O}(h^2)}$$

Méthode d'Euler n-dimensionnelle.

EDO du second ordre

Application importante : équations du second ordre qui peuvent toujours se transformer en deux équations du premier ordre. Par exemple :

$$\ddot{x} = \frac{1}{m} F(x, \dot{x}, t)$$

Introduire une fonction inconnue auxiliaire $v(t)$ par

$$v = \dot{x}$$

On obtient un système de deux EDO du premier ordre :

$$\dot{x} = v$$

$$\dot{v} = \frac{1}{m} F(x, v, t)$$

Plus généralement : en d dimensions,

$$\ddot{\vec{x}} = \frac{1}{m} \vec{F}(\vec{x}, \dot{\vec{x}}, t)$$

équivalent aux $2d$ équations du premier ordre

$$\dot{\vec{x}} = \vec{v}$$

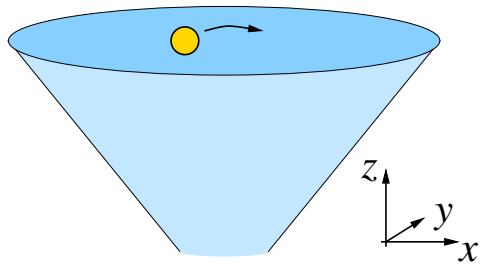
$$\dot{\vec{v}} = \frac{1}{m} \vec{F}(\vec{x}, \vec{v}, t)$$

Exemple

Une bille de masse m se déplace sans friction sur la surface d'un étonnoir conique qui est décrite par l'équation (en coordonnées cartésiennes)

$$x^2 + y^2 = \frac{z^2}{\alpha^2}$$

avec $\alpha > 0$ une constante. La gravité agît en direction négative des z. On souhaite résoudre les équations de mouvement.



Le lagrangien en coordonnées cylindriques (r, ϕ, z) est

$$L = \frac{1}{2}m\left(\dot{r}^2 + \dot{z}^2 + r^2\dot{\phi}^2\right) - mgz\,.$$

En remplaçant $z = \alpha r$ on obtient

$$L = \frac{1}{2}m\left((1+\alpha^2)\dot{r}^2 + r^2\dot{\phi}^2\right) - mg\alpha\, r\,.$$

Exemple

L'équation de mouvement pour r est donnée par l'équation d'Euler-Lagrange

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{r}} - \frac{\partial L}{\partial r} = 0$$

ce qui donne

$$\ddot{r} = \frac{\ell^2}{m^2(1+\alpha^2)}\frac{1}{r^3} - \frac{g\alpha}{1+\alpha^2}\,.$$

Ici le moment cinétique $\ell = mr^2\dot{\phi}$ est une constante du mouvement car $\frac{\partial L}{\partial \phi} = 0$ (invariance par rotations autour de l'axe des z).

Avec $v_r = \dot{r}$ on obtient la forme suivante des équations de mouvement :

$$\boxed{\begin{aligned} \dot{r} &= v_r \\ \dot{v}_r &= \frac{\ell^2}{m^2(1+\alpha^2)}\frac{1}{r^3} - \frac{g\alpha}{1+\alpha^2} \\ \dot{\phi} &= \frac{\ell}{mr^2} \end{aligned}}$$

Exemple

Importer la bibliothèque numpy, initialiser les constantes :

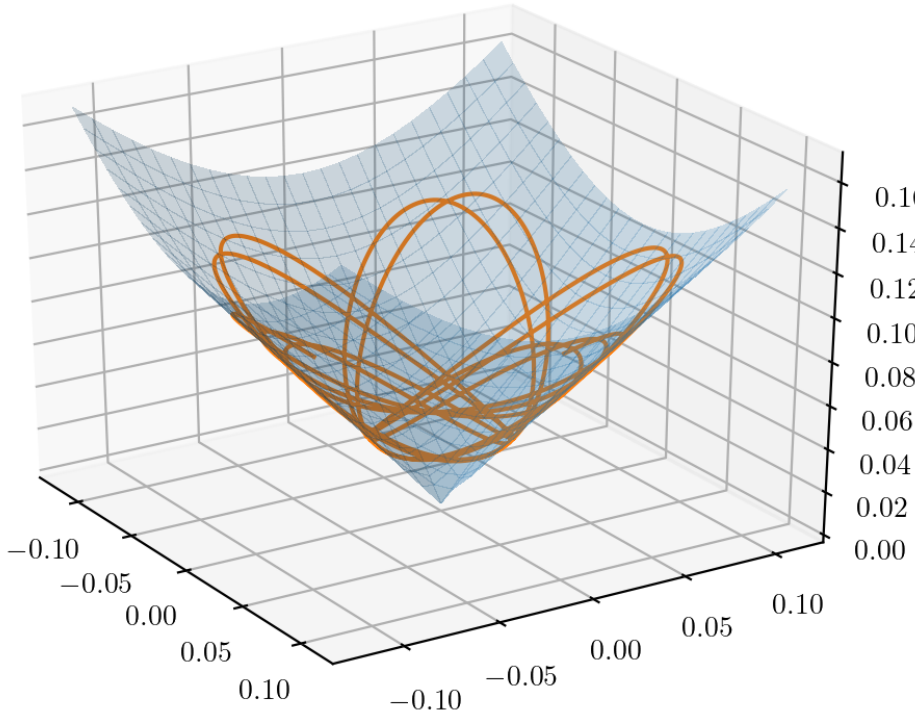
```
import numpy as np

# Constantes physiques:
g = 9.81     # accélération gravitationnelle en m/s^2
alpha = 1.   # pente du cône
m = 1.E-3    # masse de la bille en kg
# Constantes numériques:
h = 1.E-4    # pas d'incrément
N = 50000    # nombre de pas à calculer
# Conditions initiales:
r0 = .1      # rayon
v0 = .2      # vitesse radiale
phi0 = 0.0   # angle
omega0 = 4. # vitesse angulaire
# Moment cinétique (constant pour ce problème):
l = m * r0**2 * omega0
```

Exemple

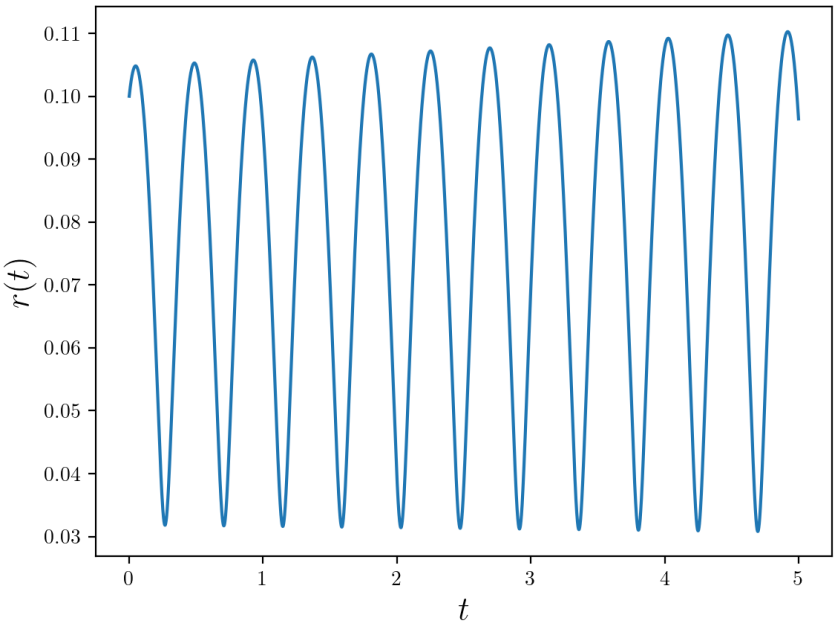
Résoudre les équations de mouvement par la méthode d'Euler :

```
def euler(f, u0, h, N):
    u = np.empty([N+1, 3])  # variables dynamiques
    u[0] = u0               # u[i] = ligne i du tableau u
    for n in range(N):
        u[n+1] = u[n] + h * f(u[n])
    return u

# La fonction vectorielle des membres de droite des e.d.m.
# u = un vecteur à 3 composantes (r, v et phi au temps t)
def f(u):
    r, v, phi = u
    fr = v
    fv = l**2 / (m**2 * (1 + alpha**2) * r**3) \
         - g * alpha / (1 + alpha**2)
    fphi = l / (m * r**2)
    return np.array([fr, fv, fphi])

u0 = np.array([r0, v0, phi0])
solution = euler(f, u0, h, N)
```

Exemple



Trajectoire pendant 5 s



$r(t)$

Visiblement l'amplitude en r de la solution numérique est croissante malgré la conservation d'énergie (artéfact numérique). Pour l'éviter on peut soit réduire h et augmenter N soit utiliser une méthode numérique plus puissante.

Méthode de Runge-Kutta classique

Problème : Résoudre le problème de Cauchy $\dot{x} = f(x(t), t)\,, \quad x(0) = x_0\,.$

Euler : Étant donné $x(t)$, pour obtenir $x(t+h)$: développement limité autour de $x(t)$, ne retenir que le premier terme,

$$x(t+h) = x(t) + h\,\dot{x}(t) + \mathcal{O}(h^2) = x(t) + h\,f(x(t), t) + \mathcal{O}(h^2)\,.$$

Runge-Kutta : Étant donné $x(t)$, pour obtenir $x(t+h)$: combinaisons linéaires de plusieurs développements limités autour des valeurs intermédiaires $x(t+\tau_i)$ avec $0 \leq \tau_i \leq h$. Pour p_i, ξ_i, τ_i bien choisis, l'erreur local devient $\mathcal{O}(h^N)$ avec $N > 1$:

$$x(t+h) = x(t) + h \sum_{i=1}^{r} p_i\,k_i + \mathcal{O}(h^N)\,, \qquad k_i = f(x + \xi_i, t + \tau_i)\,, \quad 0 < p_i < 1\,.$$

Variante la plus importante : Méthode de R-K du 4ème ordre, "RK4", "RK classique"

$$\begin{aligned}
x(t+h) &= x(t) + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right) + \mathcal{O}(h^5) \\
k_1 &= f(x(t),\ t)\,, \qquad k_2 = f\left(x(t) + \frac{h}{2}k_1,\ t + \frac{h}{2}\right)\,, \\
k_3 &= f\left(x(t) + \frac{h}{2}k_2,\ t + \frac{h}{2}\right)\,, \qquad k_4 = f\left(x(t) + h\,k_3,\ t + h\right)\,.
\end{aligned}$$

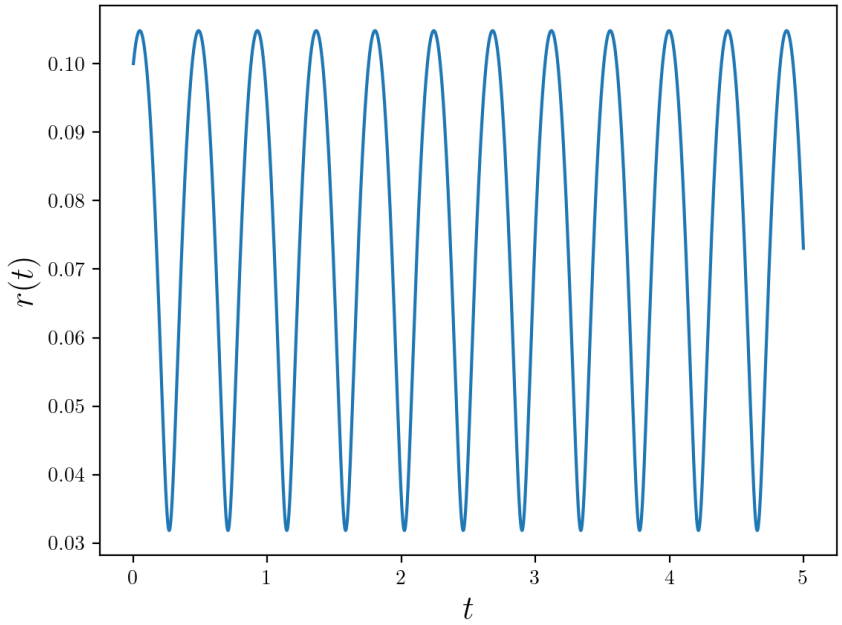
Méthode de Runge-Kutta classique

- La méthode RK4 est beaucoup plus précise que celle d'Euler.
- Elle nécessite plus d'évaluations de la fonction `f` (4 par pas, pour Euler c'est une seule), mais en pratique cela est plus que compensée par le fait qu'il faut beaucoup moins de pas pour atteindre la même précision.
- Elle est presque aussi facile à implémenter que la méthode d'Euler (4 lignes de plus).
- Elle aussi peut être appliquée aux systèmes de plusieurs EDO.

Exemple : Pour l'exemple de la bille dans l'étonnoir, remplacer la fonction `euler` par une fonction `rk4` :

```
def rk4(f, u0, h, N):
    u = np.empty([N+1, 3]) # variables dynamiques
    u[0] = u0              # u[i] = ligne i du tableau u
    for n in range(N):
        k1 = f(u[n])
        k2 = f(u[n] + h/2 * k1)
        k3 = f(u[n] + h/2 * k2)
        k4 = f(u[n] + h * k3)
        u[n+1] = u[n] + h * (k1/6 + k2/3 + k3/3 + k4/6)
    return u
```

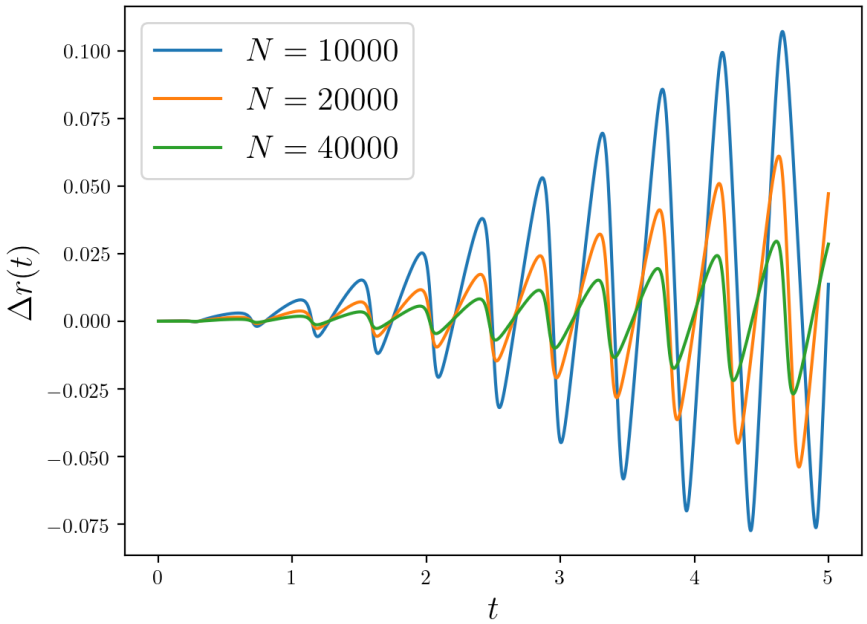
Méthode de Runge-Kutta classique

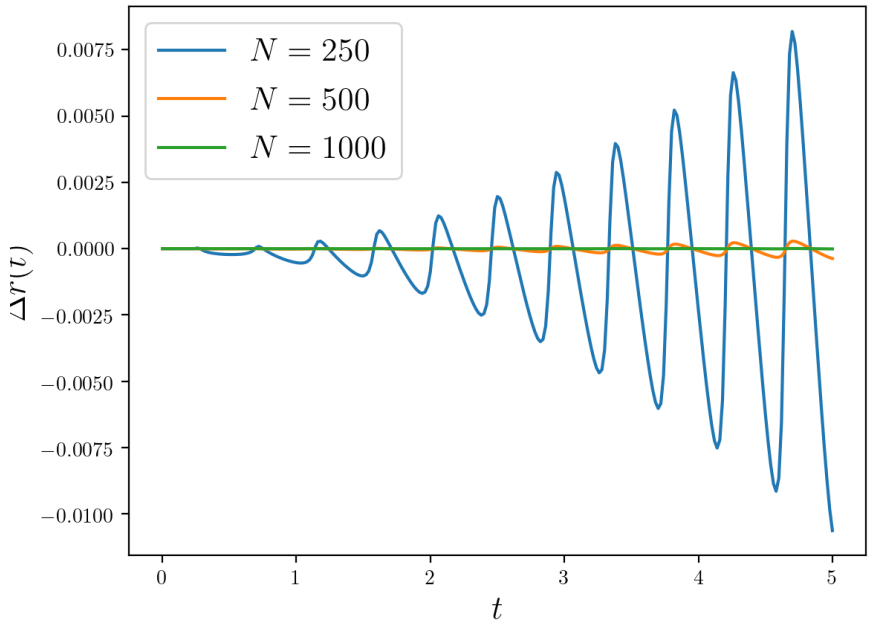


Plus de croissance artificielle visible du maximum de $r(t)$ grace à une meilleure précision de l'algorithme.

Précision numérique, Euler vs. RK4

- Avec la méthode d'**Euler** on néglige des termes $\mathcal{O}(h^2)$ à chaque pas. Après $N = \mathcal{O}(1/h)$ pas, l'erreur accumulée est alors $\mathcal{O}(h)$.
- Redoubler le nombre de pas $\Rightarrow$ l'erreur est réduite par la moitié
- Avec la méthode **RK4** les termes négligés sont $\mathcal{O}(h^5)$, l'erreur accumulée est $\mathcal{O}(h^4)$ ("methode du 4ème ordre")
- Redoubler le nombre de pas $\Rightarrow$ l'erreur est réduite par un facteur $2^4 = 16$.





Erreur en fonction de N : Euler RK4

Résumé : EDO, problèmes de Cauchy

Pour numériquement résoudre un problème aux valeurs initiales en mécanique :

- Choisir un système de coordonnées (en prenant en compte la symétrie du système, les contraintes éventuelles...)
- Trouver les équations de mouvement par le formalisme de Newton ($\vec{F} = m\vec{a}$) ou de Lagrange ($\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0$) ou de Hamilton ($\dot{p}_i = -\frac{\partial H}{\partial q_i}$, $\dot{q}_i = \frac{\partial H}{\partial p_i}$).
- Avec Newton et Lagrange, les e.d.m. sont du second ordre $\Rightarrow$ transformer en système d'équations du premier ordre.
- Résoudre les e.d.m. avec la ~~méthode d'Euler~~ méthode RK4 (plus puissante).
- Résultat : un tableau de coordonnées et vitesses/quantités de mouvement à différents t.

Algèbre linéaire numérique

Dans ce chapitre

Généralités

- Systèmes d'équations linéaires
- Diagonalisation des matrices

Algorithmes

- La méthode de Gauss
- La décomposition LU
- La décomposition QR et l'algorithme QR

Résoudre un système d'équations linéaires

On cherche une solution des n équations linéaires à n inconnues x_n

$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + \ldots + a_{1n}\,x_n &= b_1 \\ a_{21}\,x_1 + a_{22}\,x_2 + \ldots + a_{2n}\,x_n &= b_2 \\ \ldots & \\ a_{n1}\,x_1 + a_{n2}\,x_2 + \ldots + a_{nn}\,x_n &= b_n \end{aligned}$$

L'algorithme de Gauss

L'algorithme de Gauss est une méthode pour résoudre ces systèmes d'équations linéaires. En notation matricielle :

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Regardons la matrice $n \times (n+1)$ suivante :

$$M = \begin{pmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{pmatrix}$$

Observation : La solution $(x_1, \ldots, x_n)$ du système est inchangée par les transformations élémentaires :

- échange de deux lignes de M
- multiplication d'une ligne de M par un nombre $\neq 0$
- ajout d'une ligne de M à une autre

L'algorithme de Gauss

Par une suite de transformations élémentaires, on transforme M en forme triangulaire supérieure :

$$M' = \begin{pmatrix} a'_{11} & a'_{12} & \cdots & \cdots & a'_{1,n-1} & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & \cdots & a'_{2,n-1} & a'_{2n} & b'_2 \\ 0 & 0 & \ddots & \cdots & a'_{3,n-1} & a'_{3n} & b'_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & a'_{n-1,n-1} & a'_{n-1,n} & b'_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & a'_{nn} & b'_n \end{pmatrix}$$

Ici $a'_{ij} = 0$ si $i > j$.

L'algorithme de Gauss

Pour transformer M en matrice triangulaire supérieure :

1. Si la matrice ne contient qu'une seule ligne, rien à faire : terminer.
2. S'il y a au moins un coefficient non nul dans la première colonne :
   - Si $a_{11} = 0$, échanger la première ligne avec une autre dont le premier coefficient est $\neq 0$. On appellera a_{11} le coefficient pivot, il est désormais $\neq 0$.
   - Éliminer tous les coefficients a_{k1} de la première colonne au-dessous du pivot : Ajouter $(-a_{k1}/a_{11})$ fois la première ligne à la k-ème ligne.
3. Répéter à partir de 1. avec la sous-matrice de M que l'on obtient en supprimant la première ligne et la première colonne.

En pratique, on choisit souvent comme pivot le plus grand coefficient de chaque colonne, pour minimiser les erreurs d'arrondi. Pour nous il suffira de choisir un élément quelconque (non nul).

L'algorithme de Gauss

On a maintenant transformé M en forme triangulaire supérieure,

$$M' = \begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1,n-1} & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2,n-1} & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{n-1,n-1} & a'_{n-1,n} & b'_{n-1} \\ 0 & 0 & \cdots & 0 & a'_{nn} & b'_n \end{pmatrix}$$

sans avoir changé la solution $(x_1, \ldots, x_n)$. Ensuite on calcule successivement les x_i :

- $x_n = b'_n / a'_{nn}$
- $x_{n-1} = (b'_{n-1} - a'_{n-1,n} x_n)/a'_{n-1,n-1}$, avec x_n déjà connu
- ...
- $x_i = \left(b'_i - \sum_{k>i} a'_{ik} x_k\right) / a'_{ii}$, avec les x_k pour $k > i$ déjà connus
- ...
- $x_1 = \left(b'_1 - \sum_{k>1} a'_{1k} x_k\right) / a'_{11}$, avec les x_k pour $k > 1$ déjà connus

Si un des a'_{ii} est zéro, il n'y a pas de solution, sauf si le numérateur $b'_i - \sum_{k>i} a'_{ik} x_k$ est aussi zéro (dans ce cas la solution pour x_i n'est pas unique).

L'algorithme de Gauss

Exemple : On cherche la solution $\vec{x}$ du système d'équations $A\vec{x} = \vec{b}$ avec

$$A = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 1 & 2 & 3 & 4 \\ -1 & 1 & 2 & 2 \\ 0 & 3 & -2 & 0 \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} 0 \\ -3 \\ 2 \\ 0 \end{pmatrix}$$

- Au début :

$$M = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 1 & 2 & 3 & 4 & -3 \\ -1 & 1 & 2 & 2 & 2 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

- Première colonne : le pivot est 2. On ajoute donc $(-1/2)$-fois la première ligne à la deuxième et $(1/2)$-fois la première ligne à la troisième. Pour la quatrième ligne il n'y a rien à faire.

L'algorithme de Gauss

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

- Deuxième colonne : on ne peut pas prendre 0 comme pivot. Alors d'abord on échange la deuxième ligne ave la troisième :

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

  Puis, rien à faire pour la troisième ligne. Ajouter (-1)-fois la deuxième à la quatrième pour éliminer M'_{42}.

L'algorithme de Gauss

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 0 & -3 & -\frac{5}{2} & -2 \end{pmatrix}$$

- Troisième colonne : le pivot est 4. Ajouter $(3/4)$ de la troisième ligne à la quatrième :

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 0 & 0 & \frac{1}{8} & -\frac{17}{4} \end{pmatrix}$$

Maintenant M' est triangulaire supérieure et on peut calculer la solution $\vec{x}$:

- $\frac{1}{8}\, x_4 = -\frac{17}{4} \qquad \Rightarrow \; x_4 = -34$
- $4\, x_3 + \frac{7}{2}\, x_4 = 4\, x_3 - 119 = -3 \qquad \Rightarrow \; x_3 = 29$
- $3\, x_2 + x_3 + \frac{5}{2}\, x_4 = 3\, x_2 + 29 - 85 = 2 \qquad \Rightarrow \; x_2 = \frac{58}{3}$
- $2\, x_1 + 4\, x_2 - 2\, x_3 + x_4 = 2\, x_1 + \frac{232}{3} - 58 - 34 = 0 \qquad \Rightarrow \; x_1 = \frac{22}{3}$

L'algorithme de Gauss

Une procédure auxiliaire :

```
import numpy as np

# Transforme une matrice n*(n+1) en forme triang. supérieure
def triangulariser(M):
  n = M.shape[0]       # le nombre de lignes
  for i in range(n):  # boucle sur les premières n colonnes
    for k in range(i, n): # chercher pivot sous la diagonale
      if M[k, i] != 0:          # pivot trouvé dans ligne k ?
        M[[i, k], :] = M[[k, i], :] # échanger lignes i et k
        pivot = M[i, i]                # mémoriser pivot
        break                          # quitter boucle sur k
    else:  # tous les éléments sous la diagonale sont zéro?
      continue    # alors rien à faire pour cette colonne
    for k in range(i+1, n): # éliminer tout sous la diag.:
      facteur = -M[k, i]/pivot       # ajouter (facteur) ...
      M[k, :] += facteur * M[i, :]   #  ...*(ligne du pivot)
                                     #   à la ligne k.
```

L'algorithme de Gauss

```
def gauss(A, b): # trouver la solution x de Ax = b
  n = A.shape[0]           # le nombre de lignes
  M = np.empty((n, n+1)) # la matrice M
  M[:, :n] = np.copy(A) # copier A dans les 1ères n colonnes
  M[:, n] = np.copy(b)  # copier b dans la dernière colonne
  triangulariser(M)   # après, M est triangulaire supérieure

  x = np.empty(n)     # on mettra la solution ici
  for i in range(n-1, -1, -1): # parcourir lignes en arrière
    sigma = 0.        # la somme des x que l'on connaît déjà
    for k in range(i+1, n):  # ... * les M_ik correspondants
      sigma += M[i, k] * x[k]
    if M[i, i] == 0: # Matrice singulière?
      if M[i, n] - sigma == 0: # faut résoudre 0*x[i] = 0 ?
        print("Attention, solution pas unique!")
        x[i] = 42
      else:  # faut résoudre 0*x[i] = (non nul)?
        print("Erreur, pas de solution")
        return
    else: # sinon on peut diviser par M[i, i]
      x[i] = (M[i, n] - sigma) / M[i, i]
  return x
```

La décomposition LU

Les opérations élémentaires sur une matrice carrée A peuvent se représenter par des multiplications matricielles.

Ajouter r fois la j-ème à l'i-ème ligne $(i \neq j)$

$$A \rightarrow T_{(i,j;r)}A$$

$T_{(i,j;r)}$ = la matrice identité avec le coefficient (i,j) remplacé par r :

$$T_{(i,j;r)} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & r & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}$$

La décomposition LU

Les opérations élémentaires sur une matrice carrée A peuvent se représenter par des multiplications matricielles.

Échange de l'i-ème et la j-ème ligne :

$$A \rightarrow P_{(i,j)}A$$

$P_{(i,j)}$ = la matrice identité avec les lignes i et j échangées :

$$P_{(i,j)} = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & & 1 & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & 1 & & & & 0 & \\ & & & & & & & & 1 \end{pmatrix}$$

La décomposition LU

La décomposition LU d'une matrice $n \times n$ A est la décomposition

$$PA = LU$$

- U est une matrice $n \times n$ triangulaire supérieure
  (zéro au-dessous de la diagonale principale)
- L est une matrice $n \times n$ triangulaire inférieure
  (zéro au-dessus de la diagonale principale)
- P est une matrice de permutation $n \times n$
  (un produit des matrices du type $P_{(i,j)}$).

La matrice U est le résultat de l'application de l'algorithme de Gauss à la matrice A. En prenant note des transformations effectuées pendant le déroulement de l'algorithme, on peut aussi déterminer L et P.

La décomposition LU

Rappel : pour éliminer la j-ème colonne, l'algorithme de Gauss procède en deux pas,

- (Potentiellement) un échange des lignes $P_{(k,j)}$:
  - Si $a_{jj} = 0$, on cherche une autre ligne a_{k*} avec $a_{kj} \neq 0$ et on l'échange avec la j-ème ligne. Opération réprésenté par $P_{(k,j)}$ avec $k > j$.
  - Si déjà $a_{jj} \neq 0$, pas besoin de changer le pivot. Opération "nulle" réprésentée par $P_{(j,j)} = \mathbb{1}$.
- Ensuite, $(n - j - 1)$ opérations du type $T_{(i,j;r_i)}$ (où $i = j + 1 \dots n$) :
  Ajout de r_i-fois la ligne j du pivot à la ligne i (où $r_i = -a_{ij}/a_{jj}$)

Ces dernières peuvent être regroupées dans une matrice $F_{(j;r_{j+1}\dots r_n)}$ donnée par $F_{(j;r_{j+1}\dots r_n)} = T_{(j+1,j;r_{j+1})} T_{(j+2,j;r_{j+2})} \cdots T_{(n,j:r_n)}$, ou explicitement

$$F_{(j;r_{j+1}\dots r_n)} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & r_{j+1} & 1 & & \\ & & & r_{j+2} & & \ddots & \\ & & & \vdots & & & \\ & & & r_n & & & 1 \end{pmatrix}$$

La décomposition LU

On peut alors représenter les transformations de l'algorithme de Gauss schématiquement comme suit :

$$\boxed{F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,1)}\, A = U}$$

- A est la matrice à transformer
- U est le résultat de l'algorithme de Gauss, une matrice triangulaire supérieure
- Les $P_{(i,j)}$ correspondent aux changements de pivot ($P_{(i,j)} = \mathbb{1}$ si pas de changement nécessaire)
- Les $F_{(j;r_{j+1}\ldots r_n)}$ correspondent à l'élimination de la j-ème colonne

On insère $\mathbb{1}$ dans l'équation ci-dessus, en utilisant $P_{(i,j)}P_{(i,j)} = \mathbb{1}$:

$$\boxed{\begin{aligned} &F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,2)}\, P_{(*,3)} \cdots P_{(*,n-1)} \cdot \\ &\quad \cdot P_{(*,n-1)} \cdots P_{(*,3)}\, P_{(*,2)}\, P_{(*,1)}\, A = U \end{aligned}}$$

On peut montrer : La première ligne de cette expression,

$$\hat{L} = F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,2)}\, P_{(*,3)} \cdots P_{(*,n-1)}\,,$$

est triangulaire inférieure.

La décomposition LU

On est arrivé à la représentation suivante de la triangularisation de la matrice A :

$$\boxed{\hat{L} P_{(*,n-1)} \cdots P_{(*,3)} P_{(*,2)} P_{(*,1)} A = U}$$

où

$$\hat{L} = F_{(n-1;*)} P_{(*,n-1)} F_{(n-2;*)} \cdots P_{(*,2)} F_{(1;*)} P_{(*,2)} P_{(*,3)} \cdots P_{(*,n-1)}$$

est une matrice triangulaire inférieure.

L'inverse d'une matrice triangulaire inférieure est également triangulaire inférieure. On définit

$$\begin{aligned} L &\equiv \hat{L}^{-1} \\ P &\equiv P_{(*,n-1)} \cdots P_{(*,2)} P_{(*,1)} \end{aligned}$$

et on multiplie par L à la gauche aux deux cotés :

$$L\hat{L}PA = LU$$

alors

$$\boxed{PA = LU}$$

.

La décomposition LU

Pour calculer la décomposition LU, $PA = LU$:

- $U =$ le résultat de l'algorithme de Gauss appliqué à A
- $P =$ le produit de tous les $P_{(i,j)}$ (échanges de lignes effectués dans la procédure).
  - Démarrer l'algorithme de Gauss avec $P = \mathbb{1}$.
  - Quand on échange les lignes `i` et `j`, multiplier P à gauche par $P_{(i,j)}$.
- $L = \hat{L}^{-1} = P\, P_{(*,1)}\, F^{-1}_{(1;*)}\, P_{(*,2)}\, F^{-1}_{(2;*)} \cdots P_{(*,n-1)}\, F^{-1}_{(n-1;*)}$
  avec les $F_{(i;*)}$ correspondant aux éliminations de colonnes.
  - Démarrer avec $L = \mathbb{1}$.
  - Quand on échange les lignes `i` et `j`, multiplier L à droite par $P_{(i,j)}$.
  - Lors de l'élimination de la colonne `j`, multiplier L à droite par le $F^{-1}_{(j;*)}$ correspondant.
    Facile à calculer car $\left(F_{(j;r_{j+1},r_{j+2},\dots r_{n-1},r_n)}\right)^{-1} = F_{(j;-r_{j+1},-r_{j+2},\dots,-r_{n-1},-r_n)}$.
  - Enfin multiplier L à gauche par P.

La décomposition LU

Exemple : On cherche la décomposition LU de la matrice

$$A = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 1 & 2 & 3 & 4 \\ -1 & 1 & 2 & 2 \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

- Au début : $U = A$, $L = \mathbb{1}$, $P = \mathbb{1}$

La décomposition LU

- Première colonne :
  pas de changement de pivot, $P_{(i,j)} = P_{(1,1)} = \mathbb{1}$, $P \leftarrow P$, $L \leftarrow L$
  ajouter $(-1/2)$(première ligne) à (deuxième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}$$

  ajouter $(1/2)$(première ligne) à (troisième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ -\frac{1}{2} & & 1 & \\ & & & 1 \end{pmatrix}$$

  rien à faire pour quatrième ligne : $L \leftarrow L$.
  Maintenant

$$L = \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 1 & & \\ -\frac{1}{2} & & 1 & \\ & & & 1 \end{pmatrix}, \qquad P = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}.$$

La décomposition LU

- Deuxième colonne : Maintenant

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

$U_{22} = 0$: il faut changer le pivot. On échange donc la deuxième et la troisième ligne :

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

$$P_{(i,j)} = P_{(2,3)} = \begin{pmatrix} 1 & & & \\ & 0 & 1 & \\ & 1 & 0 & \\ & & & 1 \end{pmatrix}, \quad P \leftarrow P_{(2,3)}P, \quad L \leftarrow LP_{(2,3)}.$$

La décomposition LU

Maintenant

$$L = \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 0 & 1 & \\ -\frac{1}{2} & 1 & 0 & \\ & & & 1 \end{pmatrix}, \qquad P = \begin{pmatrix} 1 & & & \\ & 0 & 1 & \\ & 1 & 0 & \\ & & & 1 \end{pmatrix}.$$

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

Rien à faire pour la troisième ligne : $L \leftarrow L$.
Ajouter (-1) fois (deuxième ligne) à (quatrième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & 1 & & 1 \end{pmatrix} \qquad \text{donc} \qquad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

La décomposition LU

- Troisième colonne : Maintenant

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 0 & -3 & -\frac{5}{2} \end{pmatrix}$$

pas de changement de pivot : $P_{(i,j)} = P_{(3,3)} = \mathbb{1}$, $P \leftarrow P$, $L \leftarrow L$
ajouter $(3/4)$(troisième ligne) à (quatrième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{3}{4} & 1 \end{pmatrix} \qquad \text{donc} \qquad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 1 & -\frac{3}{4} & 1 \end{pmatrix}$$

- Finalement : $L \leftarrow PL$.

La décomposition LU

Enfin :

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 0 & 0 & \frac{1}{8} \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ 0 & 1 & -\frac{3}{4} & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La décomposition LU : Applications

Une fois calculée la décomposition LU d'une matrice carrée A, on peut facilement resoudre tous systèmes d'équations de la forme $A\vec{x} = \vec{b}$ pour un $\vec{b}$ quelconque :

$$A\vec{x} = \vec{b} \Leftrightarrow LU\vec{x} = P\vec{b} \Leftrightarrow L\vec{y} = \vec{b'} \quad \text{avec } \vec{y} = U\vec{x},\ \vec{b'} = P\vec{b}$$

- Déterminer $\vec{y}$ à partir de $L\vec{y} = \vec{b'}$:

$$\begin{aligned} y_1 &= b'_1/L_{11} \\ y_2 &= (b'_2 - L_{21}y_1)/L_{22} \\ &\vdots \\ y_n &= (b'_n - \sum_{k<n} L_{nk}y_k)/L_{nn} \end{aligned}$$

- Calculer $\vec{x}$ à partir de $U\vec{x} = \vec{y}$ (voir dernière étape de la méthode de Gauss) :

$$\begin{aligned} x_n &= y_n/U_{nn} \\ x_{n-1} &= (y_{n-1} - U_{n-1,n}x_n)/U_{n-1,n-1} \\ &\vdots \\ x_1 &= (y_1 - \sum_{k>1} U_{1k}x_k)/U_{11} \end{aligned}$$

La décomposition LU : Applications

```
def resoudre(L, U, P, b): # trouve x dans LUx = Pb
    n = U.shape[0] # les matrices sont n fois n
    bprime = P @ b       # le vecteur b' = Pb
    y = np.zeros(n)      # le vecteur y
    for i in range(n):   # initialiser y
        sigma = 0
        for k in range(i): # k entre 0 et i-1
            sigma += L[i, k] * y[k]
        y[i] = (bprime[i] - sigma) / L[i, i]
    x = np.zeros(n)      # le vecteur x
    for i in range(n-1, -1, -1): # i entre 0 and n-1,
        sigma = 0                #    en arrière
        for k in range(i+1, n):    # k entre i+1 et n-1
            sigma += U[i, k] * x[k]
        x[i] = (y[i] - sigma)/U[i, i]
    return x
```

La décomposition LU : Applications

Calcul de l'inverse : La décomposition LU permet de calculer le vecteur $\vec{x}$ dans le système d'équations $A\vec{x} = \vec{b}$. En appliquant cette méthode n fois avec $\vec{b} =$ les n vecteurs de colonne d'une matrice B, on obtient n vecteurs de solution : les vecteurs de colonne d'une matrice X qui vérifient l'équation matricielle

$$AX = B\,.$$

Si $B = \mathbb{1}$, alors $X = A^{-1}$ l'inverse de la matrice A.

```
def inverse(A): # trouve l'inverse de A
    L, U, P = LU.decomp_LU(A)
    n = A.shape[0]
    Ainv = np.empty((n, n)) # l'inverse de A, vide au début
    iden = np.identity(n)   # l'identité (i-ème ligne = e_i)
    for i in range(n):
        Ainv[:, i] = resoudre(L, U, P, iden[i])
    return Ainv
```

L'efficacité de la méthode de Gauss et la décomposition LU

- Le temps de calcul pour faire tourner la méthode de Gauss ou sa variante la décomposition LU est proportionnel à n^3 pour une matrice $n \times n$.

  Si la taille de la matrice augmente par un facteur 10 $\Rightarrow$ le programme prend 1000$\times$ plus de temps ! Pour $n = 100$ cela prend une fraction d'une seconde, pour $n = 10\,000$ c'est déjà infaisable en pratique.

- Pareil pour calculer l'inverse et le déterminant avec la décomposition LU.

- D'autres algorithmes, souvent plus efficaces, existent pour les matrices de forme spéciale (matrices creuses. . .).

Remarques concernant notre implémentation :

- Pour minimiser le besoin de mémoire et du temps de calcul, il convient de ne pas paramétrer les permutations par une matrice P mais avec un vecteur $\vec{p}$ qui contient les mêmes informations.

- Pour minimiser le temps du calcul, on ne construira pas L par une séquence de multiplications de matrices (qui coûtent chères, $\sim n^3$ opérations) mais plus directement.

- D'autres optimisations sont possibles pour les applications en pratique.

Les valeurs propres et les vecteurs propres d'une matrice

Soit A une matrice réelle $n \times n$ qui est symétrique : $A^T = A$.

Theorème : Il existe une matrice orthogonale S telle que $S^T A S \equiv D$ est une matrice diagonale.

(Rappel : S orthogonale $\overset{\text{déf.}}{\Leftrightarrow} S^T S = \mathbb{1}$; D diagonale $\overset{\text{déf.}}{\Leftrightarrow} D$ zéro hors la diagonale principale .)

Dans ce cas les coefficients de la diagonale principale de D sont les valeurs propres de A, et les colonnes de S sont les vecteurs propres de A.

(Rappel : $\lambda \in \mathbb{R}$ valeur propre de $A \overset{\text{déf.}}{\Leftrightarrow} \exists\ \vec{v} \in \mathbb{R}^n$ non nul, le *vecteur propre*, tel que $A\vec{v} = \lambda\vec{v}$.)

Problème :

Soit A une matrice symétrique donnée. On cherche ses valeurs propres et ses vecteurs propres correspondants. Équivalent : on cherche S et D.

L'algorithme QR

Un algorithme classique pour trouver les valeurs propres et les vecteurs propres d'une matrice est l'algorithme QR. Ici on va discuter une version simple. Des algorithmes plus modernes sont généralement plus efficaces et plus stables, mais aussi plus compliqués.

L'algorithme QR repose sur la décomposition QR d'une matrice carrée A,

$$A = QR$$

où Q est une matrice orthogonale et R est une matrice triangulaire supérieure. On définit la suite A_k par

$$A_0 = A\,, \qquad A_{n+1} = R_n Q_n$$

avec $A_n = Q_n R_n$ la décomposition QR de A_n. On peut montrer que cette suite converge, sous certains conditions, vers une matrice triangulaire dont les coefficients diagonaux sont les valeurs propres de A. La matrice S est le produit de toutes les matrices Q_n.

L'algorithme QR

Algorithme :

1. Démarrer avec $S = \mathbb{1}$.
2. Si A est triangulaire supérieure, terminer. Valeurs propres = coefficients sur la diagonale principale de A, vecteurs propres = colonnes de S.
3. Trouver la décomposition QR de A, $A = QR$.
4. Répéter à partir de (2) avec $A \leftarrow RQ$ et $S \leftarrow SQ$.

À vous de l'implémenter ($\rightarrow$ exercices) !

La décomposition QR

Pour implémenter l'algorithme QR, il faut savoir calculer la décomposition QR d'une matrice A. Il y a plusieurs méthodes ; un simple algorithme est la méthode modifiée de Gram-Schmidt :

- L'idée est de orthonormaliser les colonnes de A par une suite de transformations qui peuvent être representées par des matrices triangulaires supérieures. Si alors $\vec{v}_i$ et $\vec{v}_j$ sont deux colonnes différentes, on souhaite que $\vec{v}_i \cdot \vec{v}_j = 0$ et $\vec{v}_i \cdot \vec{v}_i = 1$.
- On définit la projection de $\vec{w}$ sur $\vec{v}$:

$$\text{pr}_{\vec{v}}\vec{w} \equiv \frac{\vec{v} \cdot \vec{w}}{||\vec{v}||^2}\vec{v}$$

- Pour tout vecteur de colonne $\vec{v}_i$:
  - Normaliser, $\vec{v}_i \leftarrow \vec{v}_i/||\vec{v}_i||$ (si $\vec{v}_i$ n'est pas nul)
  - Soustraire de chaque colonne derrière $\vec{v}_i$ sa projection sur $\vec{v}_i$, $\vec{v}_j \leftarrow \vec{v}_j - \text{pr}_{\vec{v}_i}\vec{v}_j \quad \forall\, j > i$
  - Maintenant $\vec{v}_i$ est orthogonal à tous les $\vec{v}_j$ avec $j > i$ (et à toutes leurs combinaisons linéaires)
- Ainsi on obtient les vecteurs de colonne de Q. Poser $R = Q^{-1}A = Q^T A$.

La décomposition QR

Exemple (Méthode modifiée de Gram-Schmidt) :

$$A = Q = \begin{pmatrix} 2 & 0 & 1 \\ 2 & 4 & -1 \\ -1 & 2 & 3 \end{pmatrix}$$

Normaliser la première colonne : $\sqrt{2^2 + 2^2 + (-1)^2} = 3$, alors

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & 0 & 1 \\ \frac{2}{3} & 4 & -1 \\ -\frac{1}{3} & 2 & 3 \end{pmatrix}$$

Le produit scalaire entre la 1ère et la 2ème colonne est $\frac{2}{3} \times 0 + \frac{2}{3} \times 4 + (-\frac{1}{3}) \times 2 = 2$. Il faut donc soustraire $2\times$(1ère colonne) de la 2ème colonne.

Le produit scalaire entre la 1ère et la 3ème colonne est $\frac{2}{3} \times 1 + \frac{2}{3} \times (-1) + (-\frac{1}{3}) \times 3 = -1$, on soustrait alors $(-1)\times$(1ère colonne) de la 3ème.

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{4}{3} & \frac{5}{3} \\ \frac{2}{3} & \frac{8}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{8}{3} & \frac{8}{3} \end{pmatrix}.$$

Maintenant la première colonne est normalisée et orthogonale aux deux autres.

La décomposition QR

Exemple (Méthode modifiée de Gram-Schmidt) :

Normaliser la deuxième colonne : $\sqrt{(-4/3)^2 + (8/3)^2 + (8/3)^2} = 4$, alors

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{8}{3} \end{pmatrix}$$

La produit scalaire entre la 2ème et la 3ème colonne est 1. Il faut donc soustraire la 2ème de la 3ème colonne.

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & 2 \\ \frac{2}{3} & \frac{2}{3} & -1 \\ -\frac{1}{3} & \frac{2}{3} & 2 \end{pmatrix}$$

Maintenant les premières deux colonnes sont normalisées, orthogonales l'une à l'autre et orthogonales à la troisième.

Il ne reste qu'à normaliser cette dernière :

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \end{pmatrix}$$

La décomposition QR

Exemple (calcul de R) :

On a

$$Q^{-1} = Q^T = \begin{pmatrix} \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \end{pmatrix}$$

et alors

$$R = Q^{-1}A = \begin{pmatrix} \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \end{pmatrix} \begin{pmatrix} 2 & 0 & 1 \\ 2 & 4 & -1 \\ -1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 2 & -1 \\ 0 & 4 & 1 \\ 0 & 0 & 3 \end{pmatrix}$$

La décomposition QR

Deux fonctions auxiliaires :

```
import numpy as np

# pour calculer la norme de v:
def norme(v):
    return np.sqrt(np.sum(v**2))

# pour calculer la projection de w sur v:
def projeter(v, w, epsilon=1.0E-10):
    vnorme = norme(v)
    if vnorme < epsilon:  # proj. sur vect. nul = vect. nul
        return np.zeros(v.shape[0])
    resultat = np.copy(v)
    resultat *= v @ w
    resultat /= vnorme**2
    return resultat
```

La décomposition QR

Orthogonalisation de Gram-Schmidt :

```
def gram_schmidt(A, epsilon=1.0E-10):
  n = A.shape[0]
  resultat = np.copy(A)  # on y mettra le résultat
  for i in range(n):     # Gram-Schmidt modifiée:
    v = resultat[:, i]  # pour tout vecteur de colonne v:
    vnorme = norme(v)
    if vnorme >= epsilon: #  normaliser (sauf si nul)
      v /= vnorme
    for j in range(i+1, n): # de toute colonne après v:
                            # soustraire sa projection sur v
      resultat[:, j] -= projeter(v, resultat[:, j], epsilon)
  return resultat
```

Décomposition QR :

```
def decomp_QR(A, epsilon=1.0E-10):
    Q = gram_schmidt(A, epsilon)
    R = Q.T @ A
    return Q, R
```

Algèbre linéaire avec SciPy

La bibliothèque SciPy contient des méthodes pour l'algorithme de Gauss, la décomposition LU, la décomposition QR et de nombreuses autres : décomposition en valeurs singulières, décomposition de Cholesky, fonctions matricielles, méthodes optimisées pour des matrices spéciales...

Pour les vraies applications dans la physique numérique, il est préférable de se servir de ces méthodes optimisées au lieu de nos implémentations du cours « faits maison ».

On les trouve dans la sous-bibliothèque `scipy.linalg` :

```
import numpy as np
import scipy.linalg as la
A = np.array([[5, 3, -1], [3, 2, -4], [-1, -4, 0]])
Ainv = la.inv(A)            # inverse
d = la.det(A)               # déterminant
vals, vecs = la.eig(A)      # valeurs/vecteurs propres
Pinv, L, U = la.lu(A)       # déc. LU: Pinv^(-1) A = L U
Q, R = la.qr(A)             # décomposition QR
```

Le travail de calcul dans cette bibliothèque reste sur des routines en C, C++ et FORTRAN optimisées qui sont beaucoup plus vite que des routines en Python.