10. Résolution d'équations différentielles

Presque tous les systèmes dynamiques sont décrits par des équations différentielles (ED). Les résoudre est l'une des applications de la programmation les plus importantes pour les physiciens. On verra dans ce chapitre deux méthodes pour le faire : **Euler** (simple mais pas très précise) et **Runge-Kutta** (plus précise mais plus compliquée). Les deux méthodes ne s'appliquent qu'aux systèmes d'ED ordinaires **du premier ordre** de la forme :

$$\dot{\vec{Q}}(t) = \vec{F}(\vec{Q}(t), t) \tag{1}$$

Elles peuvent donc être non linéaires, couplées et non autonomes, mais il faut pouvoir isoler les $\dot{Q}_i$.
Souvent les ED décrivant un système dynamique sont **du second ordre**,

$$\ddot{\vec{q}} = \vec{f}(\vec{q}, \dot{\vec{q}}, t) \tag{2}$$

Par exemple, avec $\vec{q}(t) = (x(t), y(t))$:

$$\begin{cases} \ddot{x} = \dot{x}\dot{y} - x + 3 \\ \ddot{y} = -5\dot{y}^3 + txy + t^2 \end{cases} \tag{3}$$

Heureusement on peut toujours réécrire un système de N ED du second ordre (où l'on peut isoler les $\ddot{q}_i$) comme un système de $2N$ ED du premier ordre (avec les $\dot{q}_i$ isolés). (Plus généralement N ED d'ordre p $\rightarrow$ pN ED d'ordre 1.)

$$\begin{cases} \ddot{x} = \dot{x}\dot{y} - x + 3 \\ \ddot{y} = -5\dot{y}^3 + txy + t^2 \end{cases}$$

Dans l'exemple, on définit un nouveau vecteur $\vec{Q}$ de dimension 4 qui contient aussi les dérivées premières : $\vec{Q} = (Q_0, Q_1, Q_2, Q_3) = (x, y, \dot{x}, \dot{y})$.
Pour $\vec{Q}$ on a alors le système d'ED suivant, qui est équivalent au système (3) mais bien de la forme (1) :

$$\begin{cases} \dot{Q}_0 = Q_2 \\ \dot{Q}_1 = Q_3 \\ \dot{Q}_2 = Q_2 Q_3 - Q_0 + 3 \\ \dot{Q}_3 = -5 Q_3^3 + t Q_0 Q_1 + t^2 \end{cases} \tag{4}$$

Dans le cas général (2), on définit

$$\vec{Q} = (q_0, q_1, \ldots, q_{N-1}, \quad \dot{q}_0, \dot{q}_1, \ldots, \dot{q}_{N-1})$$

Ainsi on voit que $\dot{Q}_i = Q_{i+N}$ pour $i < N$ et $\dot{Q}_i = f_{i-N}(\vec{Q}, t)$ pour $i \geq N$.
En définissant

$$\vec{F} = (Q_N, Q_{N+1}, \ldots, Q_{2N-1}, \quad f_0(\vec{Q}, t), f_1(\vec{Q}, t), \ldots, f_{N-1}(\vec{Q}, t))$$

on retrouve alors un système de la forme (1).

Méthode d'Euler

La méthode d'Euler est basée sur le développement limité au premier ordre pour calculer la solution au temps $t + dt$ à partir de la solution au temps t :

$$\vec{Q}(t + dt) = \vec{Q}(t) + \dot{\vec{Q}}(t)\, dt \tag{5}$$

Ainsi la résolution d'un système d'ED avec la méthode d'Euler consiste en deux étapes par itération :

1. D'abord on utilise le système d'ED (1) pour calculer $\dot{\vec{Q}}(t)$ à partir de $\vec{Q}(t)$ (qui est donné par les conditions initiales pour la première itération).
2. Puis on utilise (5) pour calculer $\vec{Q}(t + dt)$ à partir de $\vec{Q}(t)$ et $\dot{\vec{Q}}(t)$.

Ensuite on itère ces deux étapes jusqu'à ce que le temps final soit atteint.

- Il faut choisir la valeur du pas dt en fonction des demandes de temps et de précision : si dt est trop petit, le programme prend trop de temps, mais si dt est trop grand, le résultat n'est pas assez précis.
- Généralement une bonne procédure est de commencer par un pas assez grand mais raisonnable (vu les propriétés physiques du système) et puis refaire le calcul avec un pas deux fois plus petit pour voir si les résultats ne changent pas trop (en fonction de la précision demandée). On continue à diviser le pas par deux jusqu'à ce que les résultats se stabilisent.
- Si le système a des invariants (par ex. l'énergie mécanique) on peut également tester la précision des calculs en regardant si cette invariance est respectée.

La méthode d'Euler se programme dans sa forme la plus basique comme suivant (exemple du système (4)) :

```
#include<stdlib.h>
#include<fstream>
using namespace std;
int main() {
  int i, j, n = 4, Nt = 1000;   // n = nombre d'ED, Nt = nombre de temps
  double t = 0, tfin = 3, dt = (tfin - t) / (Nt - 1);
  double* q = (double*)malloc(n * sizeof(double));
  double* qp = (double*)malloc(n * sizeof(double));  // qp = "q-point"
  fstream fich("solution_eqdiff.res", ios::out);
  q[0] = 1.; q[1] = 1.;          // Conditions initiales pour x et y
  q[2] = 0.; q[3] = 0.;          // Conditions initiales pour dx/dt et dy/dt
  for (i = 0; i < Nt; i++) {     // Calcul des valeurs de q(dt) a q(tfin)
    fich << t << " " << q[0] << " " << q[1] << endl;
    qp[0] = q[2];                // Etape 1 : calcul de qp(t) avec systeme (4)
    qp[1] = q[3];
    qp[2] = q[2]*q[3] - q[0] + 3;
    qp[3] = -5*q[3]*q[3]*q[3] + t*q[0]*q[1] + t*t;
    for (j = 0; j < n; j++)
      q[j] = q[j] + dt * qp[j]; // Etape 2 : calcul de q(t+dt) avec Euler (5)
    t += dt; }
  fich.close(); free(q); free(qp);
  return 0; }
```

Par contre, on peut rendre le programme plus facile à généraliser (pour d'autres systèmes d'ED ou pour d'autres méthodes de calcul) en mettant le système d'ED et la méthode d'Euler dans des fonctions dédiées :

```
#include<bibli_fonctions.h>
void systeme(double* q, double t, double* qp, int n) { // Systeme d'ED
  qp[0] = q[2]; qp[1] = q[3];
  qp[2] = q[2]*q[3] - q[0] + 3;
  qp[3] = -5*q[3]*q[3]*q[3] + t*q[0]*q[1] + t*t; }
void euler(void (*syst_ed)(double*,double,double*,int),//Pointeur sur fonction
           double* q, double t, double dt, int n){//qui decrit le systeme d'ED
  int i; double* qp = (double*)malloc(n * sizeof(double));
  syst_ed(q, t, qp, n);          // Etape 1
  for (i = 0; i < n; i++)
    q[i] = q[i] + dt * qp[i];  // Etape 2
  free(qp); } // Tres important, car euler est appelee un grand nombre de fois
int main() {
  int i, n = 4, Nt = 1000;
  double t = 0, tfin = 3, dt = (tfin - t) / (Nt - 1);
  double* q = (double*)malloc(n * sizeof(double));
  fstream fich("solution_eqdiff.res", ios::out);
  q[0] = 1.; q[1] = 1.; q[2] = 0.; q[3] = 0.;     // Conditions initiales
  for (i = 0; i < Nt; i++) {   // Calcul des valeurs de q(dt) a q(tfin)
    fich << t << " " << q[0] << " " << q[1] << endl;
    euler(systeme, q, t, dt, n);
    t += dt; }
  fich.close(); free(q); ostringstream pyth;
  pyth << "A = loadtxt('solution_eqdiff.res')\n"
       << "plot(A[:,0], A[:,1])\n"                    // Tracer la solution
       << "plot(A[:,0], A[:,2])\n";
  make_plot_py(pyth);
  return 0; }
```

Méthode de Runge-Kutta (d'ordre 4)

La méthode de Runge-Kutta d'ordre 4 est une méthode plus précise que celle d'Euler pour résoudre le même système d'ED (1). La simple formule (5) est remplacée par une expression plus compliquée :

$$\vec{Q}(t+dt) = \vec{Q}(t) + \frac{dt}{6}\left(\vec{p}_1 + 2\vec{p}_2 + 2\vec{p}_3 + \vec{p}_4\right) \qquad (6)$$

avec $\vec{p}_1 = \vec{F}(\vec{Q}(t), t)$; $\vec{p}_2 = \vec{F}(\vec{Q}(t) + \frac{dt}{2}\vec{p}_1, t + \frac{dt}{2})$;
$\vec{p}_3 = \vec{F}(\vec{Q}(t) + \frac{dt}{2}\vec{p}_2, t + \frac{dt}{2})$ et $\vec{p}_4 = \vec{F}(\vec{Q}(t) + dt\,\vec{p}_3, t + dt)$.

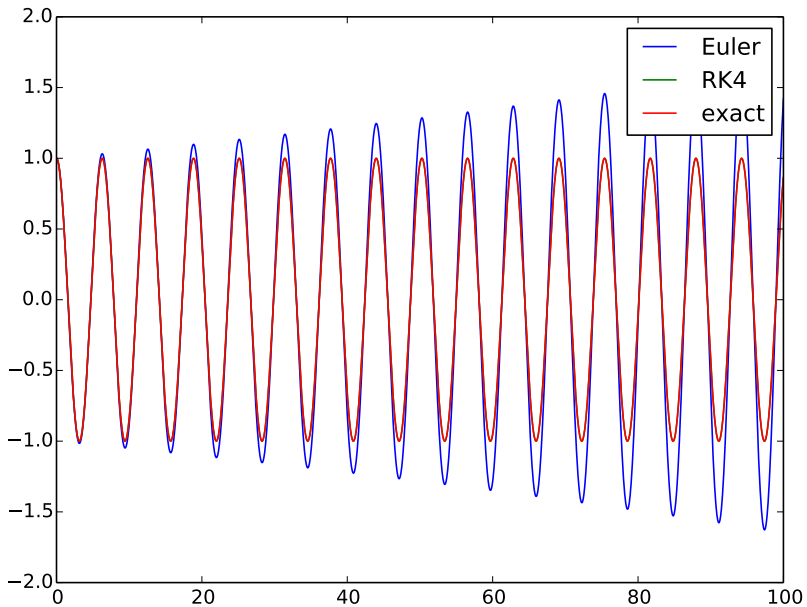
Les différences avec la méthode d'Euler sont donc que dans la première étape d'une itération on utilise (1) (qui définit $\vec{F}$) pour calculer les quatre quantités $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{p}_4$ au lieu de la seule quantité $\dot{\vec{Q}}$ (note $\vec{p}_1 = \dot{\vec{Q}}$) et que dans la deuxième étape on utilise (6) au lieu de (5).

La méthode de RK4 est déjà programmée pour vous dans la bibliothèque du magistère (`#include<bibli_fonctions.h>`) et s'utilise exactement de la même façon que la fonction `euler` dans l'exemple précédent : vous avez besoin de la même fonction `systeme` et puis vous remplacez tous les appels à `euler(systeme,q,t,dt,n)` par `rk4(systeme,q,t,dt,n)`.

On teste la précision des méthodes d'Euler et de RK4 pour l'oscillateur harmonique $\ddot{q} + q = 0$, dont on connaît la solution exacte.

```
#include<bibli_fonctions.h>
void systeme(double* q, double t, double* qp, int n) { // Systeme d'ED
  qp[0] = q[1];
  qp[1] = -q[0];
}
void euler(void (*syst_ed)(double*,double,double*,int),// Meme fct. qu'avant
           double* q, double t, double dt, int n) {
  int i;
  double* qp = (double*)malloc(n * sizeof(double));
  syst_ed(q, t, qp, n);
  for (i = 0; i < n; i++)
    q[i] = q[i] + dt * qp[i];
  free(qp);
}
int main() {
  int i, n = 2, Nt = 10000;
  double t = 0, tfin = 100, dt = (tfin - t) / (Nt - 1);
  double* q_euler = (double*)malloc(n * sizeof(double));
  double* q_rk4 = (double*)malloc(n * sizeof(double));
  fstream fich("comparaison_euler_rk4.res", ios::out);
  q_euler[0] = 1.; q_euler[1] = 0.;        // Conditions initiales
  q_rk4[0] = 1.; q_rk4[1] = 0.;
```

```
for (i = 0; i < Nt; i++) {   // Calcul des valeurs de q(dt) a q(tfin)
  fich << t << " " << q_euler[0] << " " << q_rk4[0] << endl;
  euler(systeme, q_euler, t, dt, n);
  rk4(systeme, q_rk4, t, dt, n);
  t += dt; }
free(q_euler); free(q_rk4);
fich.close(); ostringstream pyth;
pyth << "A = loadtxt('comparaison_euler_rk4.res')\n"
     << "plot(A[:,0], A[:,1], label='Euler')\n"
     << "plot(A[:,0], A[:,2], label='RK4')\n"
     << "plot(A[:,0], cos(A[:,0]), label='exact')\n" // cos(t) sol. exacte
     << "legend()\n";
make_plot_py(pyth);
return 0; }
```



Les courbes 'RK4' et 'exact' sont identiques. Par contre, la précision de la méthode d'Euler n'est pas très bonne.
Il ne faut jamais utiliser la méthode d'Euler pour des applications sérieuses.

Ex. : On met 100 charges ponctuelles fixes à des endroits aléatoires distribués uniformément dans une région circulaire (rayon 1). Puis on regarde la trajectoire d'une autre charge identique lâchée sans vitesse initiale au centre du cercle.

```
#include<bibli_fonctions.h>
double** p;  // Pour les positions des 100 charges. Variables globales parce
int Np = 100;// qu'on ne peut pas rajouter des arguments a la fonction systeme
double K = pow(1.6e-17,2)/(4*M_PI*8.85e-12*9.11e-31);  // Q^2/(4πε_0 m)
void init_charges() {    // Initialise les positions des 100 charges
  int i; double r, th;
  p = (double**)malloc(Np * sizeof(double*));
  for(i = 0; i < Np; i++) {
    p[i] = (double*)malloc(2 * sizeof(double)); // Pour les coordonnees x et y
    r = sqrt(drand48());   // Pour une distribution uniforme dans le cercle
    th = 2.*M_PI * drand48();
    p[i][0] = r * cos(th);
    p[i][1] = r * sin(th); }
}
void systeme(double* q, double t, double* qp, int n) {     // Systeme d'ED
  int i; double* r = (double*)malloc(Np * sizeof(double)); // Tableau de
  for(i = 0; i < Np; i++)  // distances entre la charge q et les 100 charges p
    r[i] = sqrt((q[0]-p[i][0])*(q[0]-p[i][0]) + (q[1]-p[i][1])*(q[1]-p[i][1]));
  qp[0] = q[2];              // q = (x, y, ẋ, ẏ)
  qp[1] = q[3];
  qp[2] = 0; qp[3] = 0;
  for(i = 0; i < Np; i++)
    qp[2] += K*(q[0] - p[i][0])/r[i]/r[i]/r[i];
    qp[3] += K*(q[1] - p[i][1])/r[i]/r[i]/r[i]; }
  free(r); }
```

```
int main() {
  int i, n = 4, Nt = 10000;
  double t = 0, tfin = 10./sqrt(K), dt = (tfin - t) / (Nt - 1);
  double* q = (double*)malloc(n * sizeof(double));
  fstream fich("solution_charges.res", ios::out);
  srand48(time(NULL));
  init_charges();
  q[0] = 0; q[1] = 0; q[2] = 0; q[3] = 0; // Cond.ini.: au centre sans vitesse
  for (i = 0; i < Np; i++)  // On ecrit dans le meme fichier d'abord positions
    fich << i << " " << p[i][0] << " " << p[i][1] << endl; // des 100 charges
  for (i = 0; i < Nt; i++) {// puis positions de charge q a chaque instant
    fich << t << " " << q[0] << " " << q[1] << endl;
    if(q[0]*q[0]+q[1]*q[1] > 2) break; // On arrete calcul si q s'est echappee
    rk4(systeme, q, t, dt, n);
    t += dt; }
  for(i = 0; i < Np; i++) free(p[i]);
  free(p); free(q);
  fich.close(); ostringstream pyth;
  pyth << "A = loadtxt('solution_charges.res')\n"
       << "plot(A[:" << Np << ",1], A[:" << Np << ",2], 'o')\n"
       << "plot(A[" << Np << ":,1], A[" << Np << ":,2])\n"
       << "x = linspace(-1,1,100)\n"
       << "plot(x,sqrt(1-x*x),'r')\n"  // Pour tracer le cercle
       << "plot(x,-sqrt(1-x*x),'r')\n"
       << "xlim(-1,1)\n"
       << "ylim(-1,1)\n";
  make_plot_py(pyth);
  return 0; }
```

Exemple : l'espace-temps autour d'un **trou noir** (en O) **en rotation** (autour de l'axe Oz) est décrit par la métrique de Kerr. On regarde la trajectoire d'une particule en **chute libre** dans le plan Oxy (à cause des symétries la trajectoire restera dans ce plan). Le mouvement correspond au sytème d'ED suivant :

$$\begin{cases} \dfrac{dt}{d\tau} = -\dfrac{1}{r^2}\left(a^2 - \dfrac{(r^2+a^2)^2}{r^2-2Mr+a^2}\right) \\ \dfrac{dr}{d\tau} = -\sqrt{\dfrac{2M(r^2+a^2)}{r^3}} \\ \dfrac{d\phi}{d\tau} = -\dfrac{a}{r^2}\left(1 - \dfrac{r^2+a^2}{r^2-2Mr+a^2}\right) \end{cases}$$

(avec $G = c = 1$) où r et ϕ sont les coordonnées polaires de la particule et t sa coordonnée temporelle (toutes définies dans le référentiel d'un observateur à l'infini), τ est le temps propre de la particule, M est la masse du trou noir et $a \equiv J/M$ est son moment cinétique par unité de masse.
(Dans la limite $a \to 0$ on retrouve les expressions pour le trou noir statique décrit par la métrique de Schwarzschild.)

L'espace-temps lui-même est entraîné en rotation par le trou noir, ce qui cause la rotation de la particule. Dans l'ergorégion (entre $r_- = M + \sqrt{M^2 - a^2}$ et $r_+ = 2M$) il est même impossible pour une particule de ne pas être en rotation dans le même sens que le trou noir, même en lui donnant une vitesse angulaire initiale dans l'autre sens. Dans la région $r \leq r_-$ aucune particule (même un photon) ne peut plus échapper au trou noir. Dans cette région les coordonnées t et ϕ ne sont plus utilisables.

```
#include<bibli_fonctions.h>
double M = 1., a = sqrt(3)/2.;  // Donc r- = 1.5 et r+ = 2
void systeme(double* q, double tau, double* qp, int n) {
  double z1 = q[1]*q[1] + a*a;  // q = (t, r, phi)
  double z2 = z1 - 2.*M*q[1];
  qp[0] = -1./(q[1]*q[1]) * (a*a - z1*z1/z2);
  qp[1] = -sqrt(2.*M*z1 / (q[1]*q[1]*q[1]));
  qp[2] = -a/(q[1]*q[1]) * (1. - z1/z2); }
int main() {
  int i, n = 3, Ntau = 100000, ic = -1;
  double tau = 0, taufin = 6, dtau = (taufin-tau)/(Ntau-1), tp;
  fstream fich("kerr.res", ios::out);
  double* q = (double*)malloc(n * sizeof(double));
  q[0] = 0; q[1] = 5; q[2] = 0; // cond.ini.: t(0)=0, r(0)=5, phi(0)=0
  for(i = 0; i < Ntau; i++) {
    tp = q[0];
    fich << tau << " " << q[0] << " " << q[1] << " " << q[2] << endl;
    rk4(systeme, q, tau, dtau, n);
    if(ic == -1 && q[0] < tp) ic = i;// Pour determiner quand t n'est plus une
    tau += dtau; }                   // bonne coordonnee (commence a diminuer)
  free(q); fich.close(); ostringstream pyth;
  pyth << "A = loadtxt('kerr.res')\n"
       << "ic = " << ic << "\n"
       << "plot(A[:,0], A[:,2])\n"
       << "figure()\n"           // Pour creer une nouvelle figure
       << "plot(A[:ic,1], A[:ic,2])\n"
       << "plot(A[:ic,1], A[:ic,3])\n";
  make_plot_py(pyth); return 0; }
```