11. Introduction à la programmation de la résolution numérique d'équations différentielles par les méthodes d'Euler et de Runge-Kutta

1 Équation différentielle du premier ordre

Soit l'équation différentielle suivante :

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

q étant la fonction inconnue de la variable t, $\dot{q}$ sa dérivée par rapport à t. f est une fonction donnée quelconque. C'est donc une équation de forme assez générale, mais il faut que $\dot{q}$ soit explicitement exprimée en fonction de t et q. C'est une équation du premier ordre, donc la donnée de la valeur $q_0 = q(t_0)$ de la fonction $q(t)$ pour une valeur donnée t_0 de la variable t détermine une solution unique.

1.1 Méthode d'Euler

La méthode la plus simple pour calculer numériquement une solution est celle d'Euler. Au premier ordre :

$$q(t+dt) = q(t) + dt\ \dot{q}(t) = q(t) + dt\ f\left[q(t),t\right]$$

expression qui permet de calculer une valeur approchée de $q(t+dt)$ à partir de la donnée de $q(t)$, cette valeur approchée étant d'autant plus exacte que le pas dt est plus petit. On calcule donc $q(t_0+dt)$ à partir de $q(t_0)$, puis en réitérant $q(t_0+2dt)$ à partir de $q(t_0+dt)$, et ainsi de suite (d'où le nom de méthode pas à pas) et on obtient une suite de valeurs approchées de la solution unique de l'équation initiale.

1.2 Programmation de la formule d'Euler

Prenons comme exemple l'équation différentielle $\dot{q} = -t\ q$ avec les conditions aux limites $t_0 = 0$, $q_0 = 1$, dont la solution est $\exp(-t^2/2)$. Évidemment le calcul approché par une méthode numérique de la solution n'est intéressant que dans les cas où on ne connaît pas la solution mathématique. Mais ici la connaissance de cette solution nous permet de vérifier nos calculs.

1.2.1 Programme le plus simple

```
// euler_1.cpp
#include<bibli_fonctions.h>
int main() {
  int i, np;
  double q, qp, t, dt, tfin;
  fstream res;
  res.open("euler_1.res", ios::out);  // fichier pour écrire les résultats
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                       // conditions initiales
  for(i = 1; i <= np; i++) {          // boucle sur la variable t
    res << t << " " << q << endl;
    qp = -t*q;                        // équation différentielle
    q = q + dt*qp;                    // calcul d'Euler
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_1.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
```

```
        << "plot(x, y, '+')\n"
        << "plot(x, exp(-x*x/2.))\n"
        ;
    make_plot_py(pyth);
    return 0;
}
```

où :

`tfin` est la valeur finale choisie pour t (ici 3)
`np` est le nombre de points que l'on veut calculer, y compris les valeurs finale et initiale (ici 30)
`dt` le pas en t ($dt = t_{\text{fin}}/(np-1)$)

La figure suivante montre le résultat obtenu (croix) comparé à la solution exacte (ligne continue).

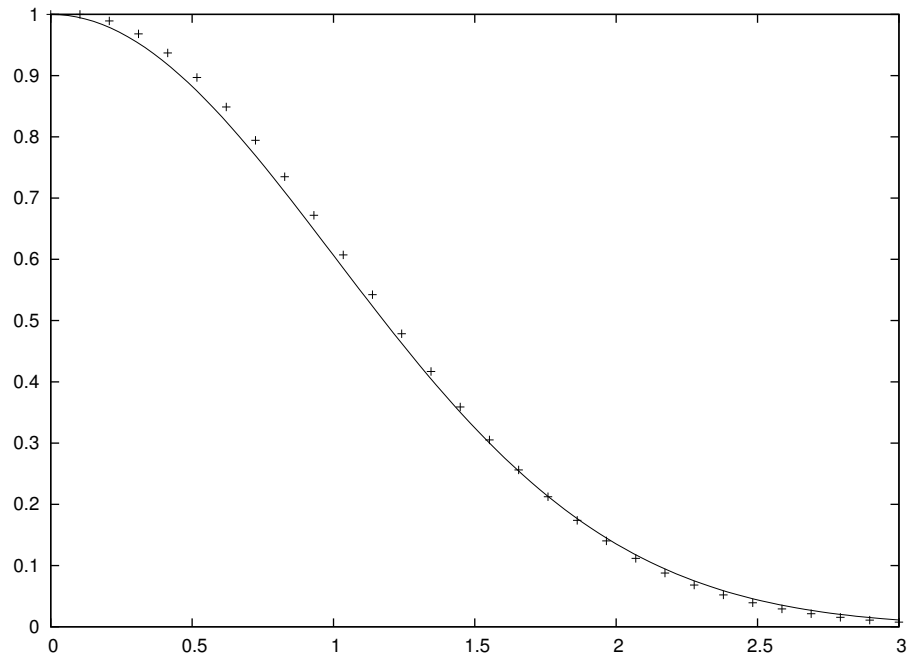

FIGURE 1 –

À partir de ce programme nous faisons maintenant une suite d'améliorations et de généralisations.

1.2.2 On met à part dans une fonction l'expression de l'équation différentielle

Cela isole l'équation différentielle du reste du programme ce qui rend ce dernier plus modulaire donc plus lisible et susceptible d'être modifié plus facilement et avec moins de risques d'erreur.

```
// euler_2.cpp
// On met l'équation différentielle dans une fonction à part
#include<bibli_fonctions.h>
//------------Fonction équation différentielle----------------
double ed(double q, double t) {
  return -t*q;
}
//---------------------------Main-----------------------------
int main() {
  int i, np;
  double q, qp, t, dt, tfin;
  fstream res;
```

```
  res.open("euler_2.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                          // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    qp = ed(q,t);
    q = q + dt*qp;                       // calcul d'Euler
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes-----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_2.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
    << "plot(x, y, '+')\n"
    << "plot(x, exp(-x*x/2.))\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

1.2.3 On met à part dans une fonction le calcul d'Euler

Cela isole le calcul d'Euler du reste du programme avec les mêmes avantages que pour l'équation différentielle.

```
// euler_3.cpp
// On met le calcul d'Euler dans une fonction à part
#include<bibli_fonctions.h>
//---------------Fonction équation différentielle--------------
double ed(double q, double t) {
  return -t*q;
}
//----------------------Fonction Euler-------------------------
double euler(double q, double t, double dt) {
  double qp;
  qp = ed(q,t);
  return q + dt*qp;
}
//-----------------------------Main-----------------------------
int main() {
  int i, np;
  double q, t, dt, tfin;
  fstream res;
  res.open("euler_3.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                          // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(q,t,dt);
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes-----------------------
  ostringstream pyth;
  pyth
```

```
      << "A = loadtxt('euler_3.res')\n"
      << "x = A[:,0]\n"
      << "y = A[:,1]\n"
      << "plot(x, y, '+')\n"
      << "plot(x, exp(-x*x/2.))\n"
      ;
    make_plot_py(pyth);
    return 0;
}
```

Remarque

La variable qp ne figure plus dans le programmme principal, elle est déclarée dans la fonction `euler`.

1.2.4 On met la fonction qui représente l'équation différentielle en argument de la fonction `euler`

Cela permet d'intégrer plusieurs équations différentielles dans une même exécution du programme.

```
// euler_4.cpp
// On met l'équation différentielle à intégrer en argument de la fonction euler
// pour pouvoir intégrer différentes équations différentielles
#include<bibli_fonctions.h>
//--------------Première équation différentielle--------------------
double ed1(double q, double t) {
  return -t*q;
}
//---------------Seconde équation différentielle--------------------
double ed2(double q, double t) {
  return -t*t*q;
}
//-----------------------Fonction Euler-----------------------------
double euler(double(*ed)(double,double), double q, double t, double dt) {
  double qp;
  qp = ed(q,t);
  return q + dt*qp;                    // calcul d'Euler
}
//----------------------------Main----------------------------------
int main()
{
  int i, np;
  double q, t, dt, tfin;
  fstream res;
  res.open("euler_4a.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  // Première équation différentielle
  t = 0; q = 1;                         // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(ed1,q,t,dt);
    t = t+dt;
  }
  res.close();
  res.open("euler_4b.res", ios::out);
  // Seconde équation différentielle
  t = 0; q = 1;                         // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(ed2,q,t,dt);
```

```
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_4a.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
    << "plot(x, y, '+')\n"
    << "plot(x, exp(-x*x/2.))\n"
    << "A = loadtxt('euler_4b.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

2 Système d'équations différentielles du premier ordre

2.1 Méthode d'Euler pour un système

La méthode d'Euler s'applique de la même façon à un système d'équations différentielles du premier ordre. Soit le système différentiel suivant :

$$\begin{aligned} \dot{q}_0 &= f_0(q_0, q_1, \ldots, t) \\ \cdots &= \cdots \\ \dot{q}_i &= f_i(q_0, q_1, \ldots, t) \\ \cdots &= \cdots \\ \dot{q}_{n-1} &= f_{n-1}(q_0, q_1, \ldots, t) \end{aligned}$$

où les q_i sont des fonctions inconnues de la seule variable t, les f_i des fonctions données quelconques, n est la dimension du système [1]. En écrivant que les q_i sont les composantes d'une vecteur $\overrightarrow{q}$ et les f_i celles d'un vecteur $\overrightarrow{f}$ le système s'écrit :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q}, t)$$

Il est du premier ordre, donc la donnée de $\overrightarrow{q}(t_0)$ détermine une solution unique. La méthode d'Euler se généralise à ce système et fournit une valeur approchée de $\overrightarrow{q}(t + dt)$ à partir de $\overrightarrow{q}(t)$:

$$\overrightarrow{q}(t + dt) = \overrightarrow{q}(t) + dt\ \overrightarrow{f}\,[\overrightarrow{q}(t), t]$$

2.2 Programmation de la formule d'Euler pour un système

On considère par exemple le système :

$$\begin{aligned} \dot{q}_0 &= q_1 \\ \dot{q}_1 &= -q_0 \end{aligned}$$

dont la solution ayant pour conditions initiales $q_0(0) = 1$ et $q_1(0) = 0$ est $q_0 = \cos t$, $q_1 = -\sin t$.
On a :

$$\begin{aligned} q_0(t + dt) &= q_0(t) + dt\ q_1(t) \\ q_1(t + dt) &= q_1(t) - dt\ q_0(t) \end{aligned}$$

1. Dans ces équations on fait varier les indices de 0 à $n-1$ plutôt que de 1 à n pour faciliter la transposition en C.

Ici le système ne comprend que deux équations différentielles et on pourrait continuer à utiliser des variables simples `q0` et `q1` par exemple pour mettre les valeurs de q_0 et q_1. Mais lorsque le nombre d'équations est plus grand il devient indispensable d'utiliser un tableau pour mettre les valeurs des q_i. De même pour les $\dot{q}_i$. Nous utilisons donc des tableaux pour que le programme puisse traiter le cas d'un système comportant un nombre quelconque n d'équations.

```
// euler_5.cpp
// On intègre un système d'équations différentielles du premier ordre
#include<bibli_fonctions.h>
//-----------------------Fonction système d'équations différentielles-------------------------
void sda(double* q, double t, double* qp, int n) {
  qp[0] = q[1];
  qp[1] = -q[0];
}
//---------------------------------------Fonction euler-----------------------------------------
void euler(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i;
  double* qp = (double*)malloc(n*sizeof(double)); // ou bien double* qp = D_1(n) si on utilise
  sd(q,t,qp,n);                                   // les fonctions du Magistère
  for(i = 0; i < n; i++)
    q[i] = q[i] + dt*qp[i];                       // calcul d'Euler
  free(qp);   // très important ici parce qu'Euler est appelée un très grand nombre de fois
              // (ou bien f_D_1(qp,n) si on utilise des fonctions du Magistère)
}
//-----------------------------------------Main-------------------------------------------------
int main() {
  int i, np, n = 2;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double)); // ou bien double *q=D_1(n) si on utilise
  fstream res;                                   // les fonctions du Magistère
  res.open("euler_5.res", ios::out);
  np = 100; tfin = 7; dt = tfin/(np-1);          // valeurs des paramètres
  t = 0; q[0] = 1; q[1] = 0;                     // conditions initiales
  for(i = 1; i <= np; i++) {                     // boucle sur t
    res << t << " " << q[0] << " " << q[1] << endl;
    euler(sda,q,t,dt,n);
    //rk4(sda,q,t,dt,n);  // seule ligne à changer si on fait Runge-Kutta au lieu d'Euler
    t = t+dt;             // (voir ci-dessous la présentation de la méthode de Runge-Kutta)
  }
  res.close();
  //---------------------------------------Tracé des courbes---------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_5.res')\n"
    << "t = A[:,0]\n"
    << "x = A[:,1]\n"
    << "y = A[:,2]\n"
    << "plot(t, x, '+')\n"
    << "plot(t, cos(t))\n"
    << "plot(t, y, '+')\n"
    << "plot(t, -sin(t))\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

ce qui donne le résultat :

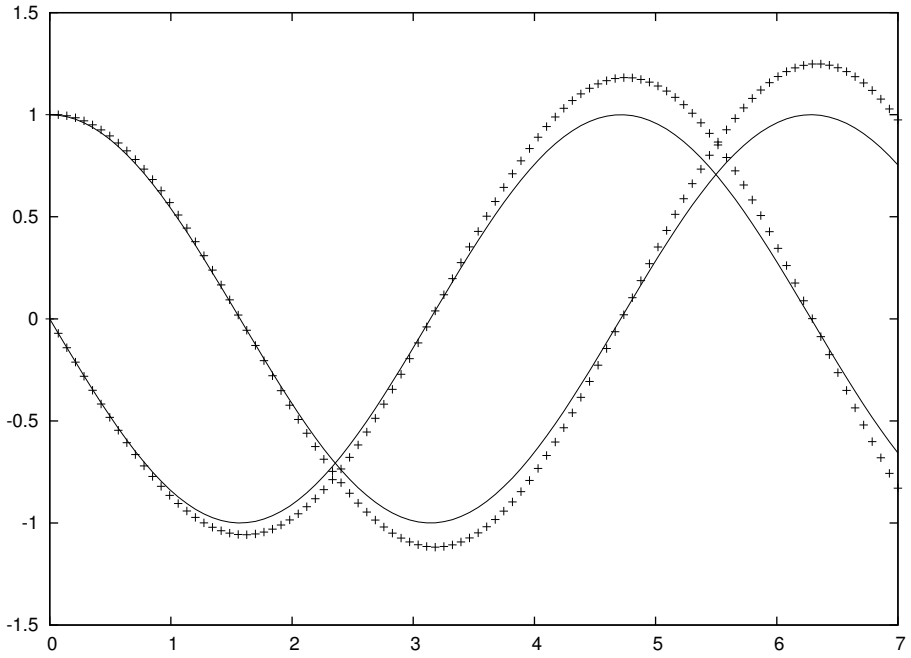

FIGURE 2 –

Les croix représentent le calcul d'Euler, les courbes continues la solution exacte.

Remarques

1. La fonction `euler` reçoit les valeurs des `q` à l'instant `t` ainsi que les valeurs de `t`, `dt` et `n`. Elle appelle alors la fonction `sd` qui calcule et renvoit les `qp` à `euler`. À l'aide de ces `qp`, `euler` calcule les valeurs des `q` à l'instant `t+dt` et les renvoit au programme principal. Finalement l'appel de la fonction `euler` remplace les valeurs des `q` à l'instant `t` par celles à l'instant `t+dt`.
2. Comme dans le cas d'une seule équation, les `qp` ne figurent pas dans le programmme principal, ils sont déclarés par une allocation dynamique dans la fonction `euler`.
3. Dans le cas de cet exemple `n` n'est pas utilisé par la fonction `sd` et il semble inutile de lui transmettre sa valeur. Mais dans le cas où le système différentiel comporte beaucoup d'équations on est amené à mettre des boucles utilisant `n` pour le calcul des `qp`. On le fait donc figurer ici dans un souci de généralité.
4. Toujours dans le cas de cet exemple il ne sert à rien de mettre le temps en argument comme cela est fait car il n'est utilisé ni dans `euler` ni dans `sd`. Mais il y aura des cas où il interviendra dans `sd` comme par exemple dans le système différentiel correspondant à une particule soumise à une force dépendant explicitement du temps et il faudra donc le transmettre à `sd` par l'intermédiaire d'`euler`. On l'introduit donc quand même pour que le programme soit le plus général possible.

3 Système d'équations différentielles du second ordre

Remarque

> Ne pas confondre l'ordre d'un système différentiel, qui est l'ordre de dérivation le plus élevé qui y figure, avec le nombre d'équations qu'il comprend.

Les systèmes d'équations différentielles que nous avons étudiés sont d'ordre un puisqu'ils ne contiennent que les dérivées premières. La méthode se généralise à des systèmes d'ordre supérieur. Considérons par exemple le système de n équations à n fonctions inconnues :

$$\ddot{\vec{r}} = \vec{g}(\vec{r}, \dot{\vec{r}}, t)$$

en notant maintenant $\overrightarrow{r}$ [2] au lieu de $\overrightarrow{q}$ les fonctions inconnues et $\overrightarrow{g}$ au lieu de $\overrightarrow{f}$ le système différentiel.
On peut écrire, comme dans le cas du premier ordre :

$$\overrightarrow{r}(t+dt) = \overrightarrow{r}(t) + dt\ \dot{\overrightarrow{r}}(t)$$

et, de plus :

$$\dot{\overrightarrow{r}}(t+dt) = \dot{\overrightarrow{r}}(t) + dt\ \ddot{\overrightarrow{r}}(t) = \dot{\overrightarrow{r}}(t) + dt\ \overrightarrow{g}\left[\overrightarrow{r}(t), \dot{\overrightarrow{r}}(t), t\right]$$

Donc si on connaît $\overrightarrow{r}(t)$ et $\dot{\overrightarrow{r}}(t)$ on peut calculer $\overrightarrow{r}(t+dt)$ et $\dot{\overrightarrow{r}}(t+dt)$.
Introduisons un vecteur $\overrightarrow{q}$ à 2n composantes définies comme suit :

$$\begin{array}{lcl} q_0 & = & r_0 \\ q_1 & = & r_1 \\ ... & = & ... \\ q_{n-1} & = & r_{n-1} \\ q_n & = & \dot{r}_0 \\ q_{n+1} & = & \dot{r}_1 \\ ... & = & ... \\ q_{2n-1} & = & \dot{r}_{n-1} \end{array}$$

c'est à dire :

$$\begin{array}{lcll} q_i & = & r_i & \text{pour } i = 0...(n-1) \\ q_i & = & \dot{r}_{i-n} & \text{pour } i = n...(2n-1) \end{array}$$

On peut alors écrire :

$$\begin{array}{lcl} \dot{q}_0 & = & q_n \\ \dot{q}_1 & = & q_{n+1} \\ ... & = & ... \\ \dot{q}_i & = & q_{n+i} \\ ... & = & ... \\ \dot{q}_{n-1} & = & q_{2n-1} \\ \dot{q}_n & = & g_0(\overrightarrow{q}, t) \\ \dot{q}_{n+1} & = & g_1(\overrightarrow{q}, t) \\ ... & = & ... \\ \dot{q}_i & = & g_{i-n}(\overrightarrow{q}, t) \\ ... & = & ... \\ \dot{q}_{2n-1} & = & g_{n-1}(\overrightarrow{q}, t) \end{array}$$

Si on introduit une fonction $\overrightarrow{f}$ à $2n$ composantes définies comme suit :

$$\begin{array}{lcl} f_0(\overrightarrow{q}, t) & = & q_n \\ f_1(\overrightarrow{q}, t) & = & q_{n+1} \\ ... & = & ... \\ f_{n-1}(\overrightarrow{q}, t) & = & q_{2n-1} \\ f_n(\overrightarrow{q}, t) & = & g_0(\overrightarrow{q}, t) \\ f_{n+1}(\overrightarrow{q}, t) & = & g_1(\overrightarrow{q}, t) \\ ... & = & ... \\ f_{2n-1}(\overrightarrow{q}, t) & = & g_{n-1}(\overrightarrow{q}, t) \end{array}$$

c'est à dire :

$$\begin{array}{lcll} f_i(\overrightarrow{q}, t) & = & q_{n+i} & \text{pour } i = 0...(n-1) \\ f_i(\overrightarrow{q}, t) & = & g_{i-n}(\overrightarrow{q}, t) & \text{pour } i = n...(2n-1) \end{array}$$

2. Cette notation $\overrightarrow{r}$ n'a évidemment en général rien à voir avec la notation $\overrightarrow{r} = \overrightarrow{OM}$ du rayon-vecteur d'un point dans l'espace à deux ou trois dimensions.

On peut alors écrire :

$$\begin{array}{lcl}
\dot{q}_0 & = & f_0(\overrightarrow{q},t) \\
\dot{q}_1 & = & f_1(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_i & = & f_i(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_{n-1} & = & f_{n-1}(\overrightarrow{q},t) \\
\dot{q}_n & = & f_n(\overrightarrow{q},t) \\
\dot{q}_{n+1} & = & f_{n+1}(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_i & = & f_i(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_{2n-1} & = & f_{2n-1}(\overrightarrow{q},t)
\end{array}$$

soit :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q},t)$$

et le système d'ordre deux (de dimension n) est ramené à un système d'ordre un (de dimension $2n$). On peut généraliser à un système d'ordre quelconque pourvu que les dérivées d'ordre le plus élevé soient exprimées explicitement en fonction de celles d'ordre moins élevé.
Exemple :

$$\begin{array}{lcl}
\ddot{r}_0 & = & r_1 - \frac{t}{4}\dot{r}_1 + 6t \\
\ddot{r}_1 & = & r_0 + 4\dot{r}_0 - \frac{\dot{r}_1}{4}
\end{array}$$

On pose :

$$\begin{array}{lcl}
q_0 & = & r_0 \\
q_1 & = & r_1 \\
q_2 & = & \dot{r}_0 \\
q_3 & = & \dot{r}_1
\end{array}$$

et le système différentiel s'écrit :

$$\begin{array}{lcl}
\dot{q}_0 & = & q_2 \\
\dot{q}_1 & = & q_3 \\
\dot{q}_2 & = & q_1 - \frac{t}{4}q_3 + 6t \\
\dot{q}_3 & = & q_0 + 4q_2 - \frac{q_3}{4}
\end{array}$$

Puisqu'on s'est ramené à la résolution d'un système du premier ordre la programmation du calcul pas à pas des solutions entre dans le cadre du programme `euler_5.cpp` précédemment écrit dans lequel il suffit de changer :

— l'expression du système différentiel dans la fonction `sda`
— le nombre d'équations et les conditions initiales dans le `main`.

Ce qui donne :

```
// euler_6.cpp
// On intègre un système d'équations différentielles du second ordre ramené au premier ordre
#include<bibli_fonctions.h>
//-----------------------Fonction système d'équations différentielles-------------------------
void sda(double* q, double t, double* qp, int n) {
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = q[1] - t*q[3]/4 + 6*t;
  qp[3] = q[0] + 4*q[2] - q[3]/4;
}
//--------------------------------------Fonction euler-----------------------------------------
void euler(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i;
  double* qp = (double*)malloc(n*sizeof(double)); // ou bien double* qp = D_1(n) si on utilise
  sd(q,t,qp,n);                                   // les fonctions du Magistère
  for(i = 0; i < n; i++)
```

```
    q[i] = q[i] + dt*qp[i];                        // calcul d'Euler
  free(qp);   // très important ici parce qu'Euler est appelée un très grand nombre de fois
              // (ou bien f_D_1(qp,n) si on utilise des fonctions du Magistère)
}
//----------------------------------------Main-----------------------------------------------
int main() {
  int i, np, n = 4;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double)); // ou bien double* q = D_1(n) si on utilise
  fstream res;                                   // les fonctions du Magistère
  res.open("euler_6.res", ios::out);
  np = 100; tfin = 1.5; dt = tfin/(np-1);        // valeurs des paramètres
  t = 0; q[0] = 0; q[1] = 0; q[2] = 0; q[3] = 0; // conditions initiales
  for(i = 1; i <= np; i++) {                     // boucle sur t
    res << t << " " << q[0] << " " << q[1] << endl;
    euler(sda,q,t,dt,n);
    //rk4(sda,q,t,dt,n);  // seule ligne à changer si on fait Runge-Kutta au lieu d'Euler
    t = t+dt;             // (voir ci-dessous la présentation de la méthode de Runge-Kutta)
  }
  res.close();
  //----------------------------------------Tracé des courbes----------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_6.res')\n"
    << "t = A[:,0]\n"
    << "x = A[:,1]\n"
    << "y = A[:,2]\n"
    << "plot(t, x, '+')\n"
    << "plot(t, t**3)\n"
    << "plot(t, y, '+')\n"
    << "plot(t, t**4)\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

avec le résultat :

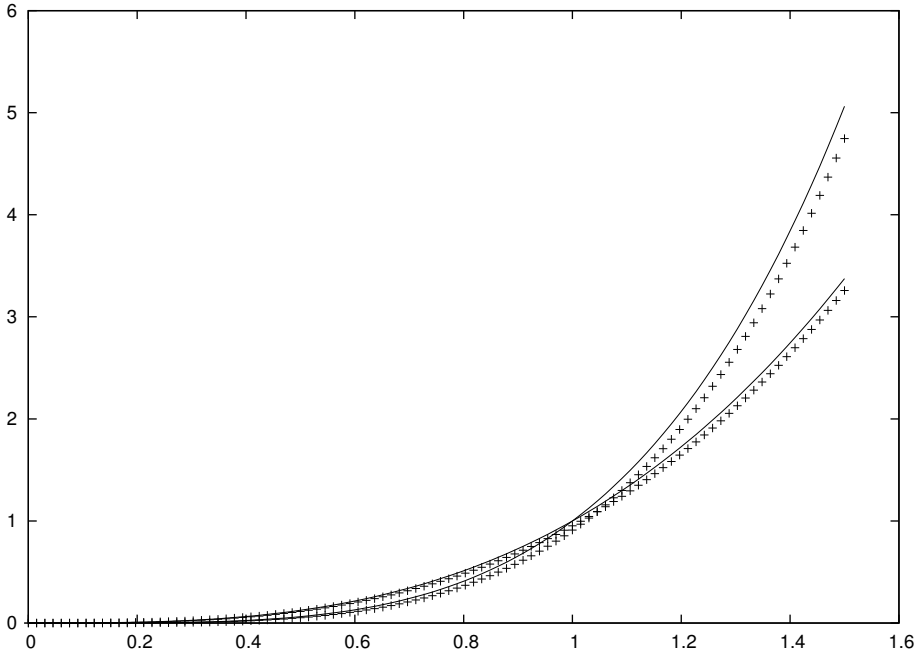

FIGURE 3 –

Les croix représentent le calcul d'Euler, les courbes continues la solution exacte ($r_0 = t^3, r_1 = t^4$).

Remarques

1. Le programme `euler_5.cpp` (ou aussi bien `euler_6.cpp`) permet ainsi en principe de résoudre un système d'un nombre quelconque d'équations et d'ordre quelconque, dès lors que les équations sont résolues par rapport aux dérivées d'ordre le plus élevé. Il suffit d'écrire le système dans la fonction système différentiel, notée ici `sda`, et de donner les conditions initiales avant la boucle sur la variable `t`.
2. Jusqu'ici on ne s'est pas posé la question de savoir comment choisir la valeur du pas dt. S'il est trop grand l'approximation pas à pas est trop imprécise, s'il est trop petit le calcul est inutilement long ou faussé par les erreurs de troncature. Il faut choisir un pas d'essai en se guidant sur les propriétés mathématiques (ou physiques s'il s'agit d'un problème de physique) du système d'équations étudié et faire un premier calcul avec ce pas. Ensuite on fait un second calcul avec un pas deux fois plus petit et on compare les résultats des deux calculs. S'ils diffèrent notablement on continue à diviser le pas par deux jusqu'à ce que les résultats se stabilisent. Cette procédure peut éventuellement être exécutée par le programme lui même.
   En pratique dans le programme `euler_5.cpp` ce n'est pas directement le pas dt qu'on choisit mais l'intervalle $[t_{\text{initial}}, t_{\text{final}}]$ sur lequel on veut intégrer le système d'équations différentielles et le nombre n_p de points que l'on veut calculer. Le pas en découle par la formule $dt = \dfrac{t_{\text{final}} - t_{\text{initial}}}{n_p - 1}$.

4 Méthode de Runge-Kutta d'ordre 4

C'est, comme la méthode d'Euler, une méthode pas à pas permettant de calculer une valeur approchée de $q(t+dt)$ à partir de celle de $q(t)$. La formule d'itération est plus compliquée que celle de la méthode d'Euler mais l'approximation est meilleure pour une même valeur du pas. Elle s'applique aux systèmes de même type que ceux étudiés par la méthode d'Euler :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q}, t)$$

Les formules d'itération sont les suivantes :

$$\overrightarrow{q}(t+dt) = \overrightarrow{q}(t) + \frac{dt}{6}(\overrightarrow{p}_1 + 2\,\overrightarrow{p}_2 + 2\,\overrightarrow{p}_3 + \overrightarrow{p}_4)$$

avec :

$$\overrightarrow{p}_1 = \overrightarrow{f}(t, \overrightarrow{q}(t))$$

$$\overrightarrow{p}_2 = \overrightarrow{f}(t + \frac{dt}{2}, \overrightarrow{q}(t) + \frac{dt}{2}\overrightarrow{p_1})$$

$$\overrightarrow{p}_3 = \overrightarrow{f}(t + \frac{dt}{2}, \overrightarrow{q}(t) + \frac{dt}{2}\overrightarrow{p_2})$$

$$\overrightarrow{p}_4 = \overrightarrow{f}(t + dt, \overrightarrow{q}(t) + dt\,\overrightarrow{p_3})$$

Il y a donc quatre quantités intermédiaires à calculer ($\overrightarrow{p}_1$, $\overrightarrow{p}_2$, $\overrightarrow{p}_3$, $\overrightarrow{p}_4$) pour obtenir $\overrightarrow{q}(t+dt)$.
La fonction C, qui calcule $\overrightarrow{q}(t+dt)$ en fonction de $\overrightarrow{q}(t)$ à l'aide de ces formules est donnée en annexe. Elle est nommée `rk4(f,q,t,dt,n)` et s'emploie exactement comme `euler(f,q,t,dt,n)`. A chaque appel, les valeurs de $\overrightarrow{q}(t)$, contenues dans le tableau `q`, sont remplacées par les valeurs de $\overrightarrow{q}(t+dt)$, `f` étant la fonction qui définit le système différentiel.

Considérons à nouveau le système :

$$\begin{aligned} \dot{q}_0 &= q_1 \\ \dot{q}_1 &= -q_0 \end{aligned}$$

précédemment étudié avec la méthode d'Euler. Pour faire le calcul avec la méthode Runge-Kutta d'ordre 4 il suffit de remplacer l'appel `euler(ed1,q,t,dt,n);` par `rk4(ed1,q,t,dt,n);` [3] dans la boucle sur t du programme précédent `euler_5.cpp`. On obtient le le résultat suivant :

3. La fonction `rk4` fait partie de la bibliothèque du Magistère

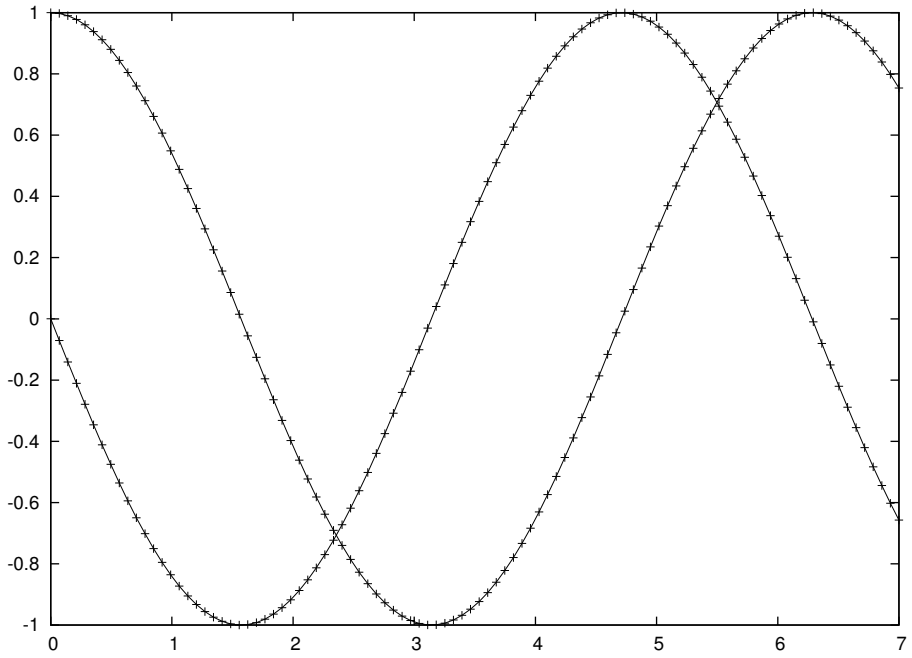

FIGURE 4 –

Les croix représentent le calcul RK4, les pointillés la solution exacte. Les croix suivent les pointillés à la précision de la figure : on vérifie ainsi que, pour un pas identique, la précision du calcul RK4 est meilleure que celle du calcul d'Euler.

Dans la suite on va montrer l'application de la méthode de Runge-Kutta d'ordre 4 à un certain nombre d'exemples pris en physique. On emploiera systématiquement la méthode RK4, bien plus précise que celle d'Euler, cette dernière ayant été utilisée au début du chapitre uniquement parce qu'elle permet d'expliquer simplement le principe des méthodes pas à pas et de leur programmation.
La méthode d'ajustement du pas est la même pour RK4 que pour Euler.

5 Exemples d'application de la méthode de Runge-Kutta d'ordre 4

5.1 Exemple : régime transitoire et régime permanent d'un oscillateur

Une masse m liée sans frottement à une tige rectiligne est rappelée par un ressort de raideur k. Elle est soumise de plus :

à une force de frottement fluide proportionnelle à la vitesse
à une force extérieure sinusoïdale $f \sin \omega t$.

Sa position $q(t)$ obéit à l'équation différentielle du second ordre :

$$\ddot{q} = -\frac{l}{m}\dot{q} - \frac{k}{m}q + \frac{f}{m}\sin \omega t$$

On pose :

$$q_0 = q$$
$$q_1 = \dot{q}$$

et :

$$a = \frac{l}{m}$$
$$\omega_0 = \sqrt{\frac{k}{m}}$$

L'équation différentielle du second ordre est équivalente au système du premier ordre :

$$\dot{q}_0 = q_1$$
$$\dot{q}_1 = -aq_1 - \omega_0^2 q_0 + \frac{f}{m}\sin\omega t$$

qui peut être résolu numériquement en utilisant la fonction `rk4` dans le programme suivant :

```
// oscill.cpp
#include<bibli_fonctions.h>
void osc(double* q, double t, double* qp, int n) {
  static double l = 0.1, om0 = 3.7, f = 2., om = 1.;
  qp[0] = q[1];
  qp[1] = -l*q[1] - om0*om0*q[0] + f*sin(om*t);
}
int main() {
  int i, np, n = 2;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double));
  fstream res;
  res.open("oscill.res", ios::out);
  t = 0.; q[0] = 1.; q[1] = 0.;
  np = 2000; tfin = 120.; dt = tfin/(np-1);
  for(i = 1; i <= np; i++) {
    res << t << " " << q[0] << " " << q[1] << endl;
    rk4(osc,q,t,dt,n);
    t = t+dt;
  }
  res.close();
  //---------------------------------------Tracé des courbes--------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('oscill.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

On obtient la courbe suivante pour $q(t)$:

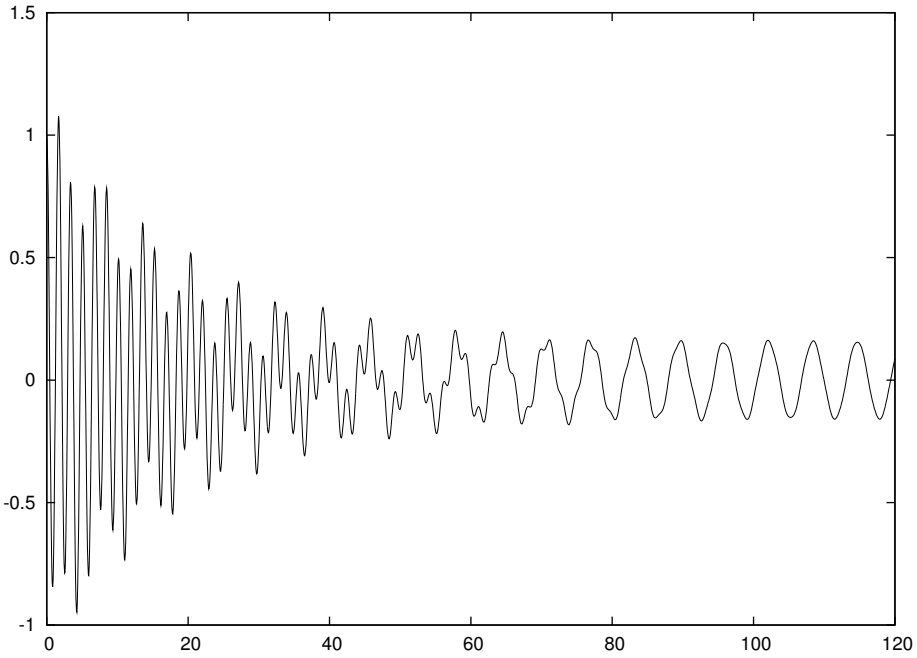

FIGURE 5 –

montrant le passage progressif des oscillations libres amorties aux oscillations entretenues et en particulier le changement de fréquence.

5.2 Pendule à deux degrés de liberté

Un pendule est constitué d'une tige rigide de longueur l et de masse négligeable attachée à un point fixe O et dont l'autre extrémité porte une masse ponctuelle m. La tige tourne librement autour du point O. On lance le pendule depuis une position quelconque avec une vitesse quelconque, ce qui fait qu'en général le mouvement ne se fait pas dans un plan. On ne fait pas l'approximation des petites oscillations.

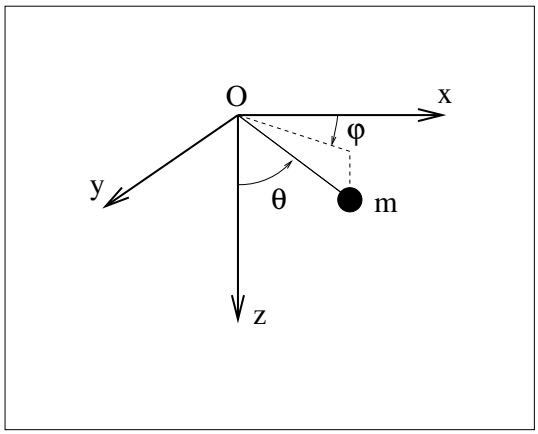

FIGURE 6 –

Dans ces conditions les angles θ et φ qui repèrent la position du pendule obéissent au système d'équations différentielles :

$$\ddot{\theta} = \frac{1}{2}\sin 2\theta\, \dot{\varphi}^2 - \frac{g}{l}\sin\theta$$
$$\ddot{\varphi} = -\frac{2\,\dot{\theta}\,\dot{\varphi}}{\tan\theta}$$

Le programme peut s'écrire :

```
// pendule_2_degres_liberte.cpp
#include<bibli_fonctions.h>
double g = 9.81, l;
void el(double* q, double t, double* qp, int n) {
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = sin(2*q[0])*q[3]*q[3]/2 - g/l*sin(q[0]);
  qp[3] = -2/tan(q[0])*q[2]*q[3];
}
int main() {
  int i, np, n = 4;
  double t,dt,tfin;
  double* q = (double*)malloc(n*sizeof(double));
  fstream res;
  res.open("pendule_2_degres_liberte.res", ios::out);
  // Unités : SI
  l = g/2/M_PI/2/M_PI;
  // Conditions initiales
  t = 0; q[0] = M_PI/2; q[1] = 0; q[2] = 0; q[3] = 0.2;
  np = 100000; tfin = 20; dt = tfin/(np-1);
  // Boucle sur le temps
  for(i = 1; i <= np; i++) {
    res << l*sin(q[0])*cos(q[1]) << " " << l*sin(q[0])*sin(q[1]) << endl;
    rk4(el,q,t,dt,n);
    t += dt;
  }
  res.close();
  //---------------------------------------Tracé des courbes---------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('pendule_2_degres_liberte.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

et on obtient la courbe suivante pour la projection de la position de la masse dans le plan xOy :

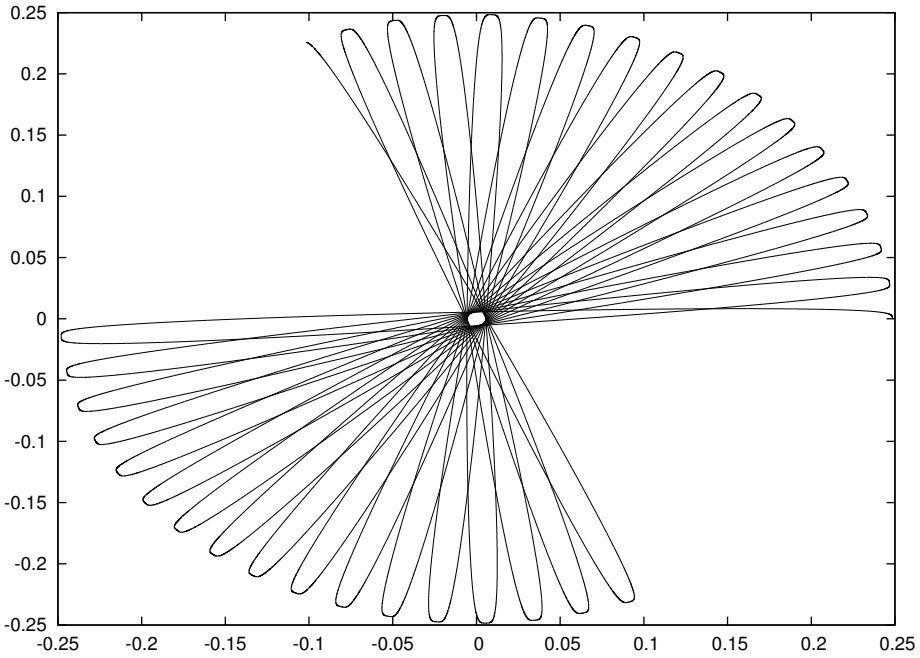

FIGURE 7 –

5.3 Mouvement d'un satellite en présence de frottement

Un satellite est soumis à l'attraction de la terre et à une force de frottement fluide proportionnelle au carré de sa vitesse.

$$\ddot{\overrightarrow{r}} = -gr_t^2 \frac{\overrightarrow{r}}{r^3} - \frac{k}{m}\dot{r}\dot{\overrightarrow{r}}$$

avec :

- $\overrightarrow{r}$ vecteur position du satellite, de composantes x et y
- g accélération de la pesanteur au niveau du sol
- r_t rayon de la terre
- m masse du satellite

C'est un système de deux équations du second ordre que l'on peut ramener à quatre équations du premier ordre. On pose :

$$a = gr_t^2$$
$$b = \frac{k}{m}$$

et :

$$q_0 = x$$
$$q_1 = y$$
$$q_2 = \dot{x}$$
$$q_3 = \dot{y}$$

Le système s'écrit :

$$\dot{q}_0 = q_2$$
$$\dot{q}_1 = q_3$$
$$\dot{q}_2 = -a\frac{q_0}{(q_0^2+q_1^2)^{\frac{3}{2}}} - b\sqrt{q_2^2+q_3^2}\, q_2$$
$$\dot{q}_3 = -a\frac{q_1}{(q_0^2+q_1^2)^{\frac{3}{2}}} - b\sqrt{q_2^2+q_3^2}\, q_3$$

La fonction définissant l'équation différentielle peut s'écrire :

```
void sa(double* q, double t, double* qp, int n) {
  double r3, v;
  r3 = pow(q[0]*q[0]+q[1]*q[1], 3./2.);
  v = sqrt(q[2]*q[2]+q[3]*q[3]);
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = -a*q[0]/r3 - b*v*q[2];
  qp[3] = -a*q[1]/r3 - b*v*q[3];
}
```

et on obtient, comme attendu, des trajectoires du type :

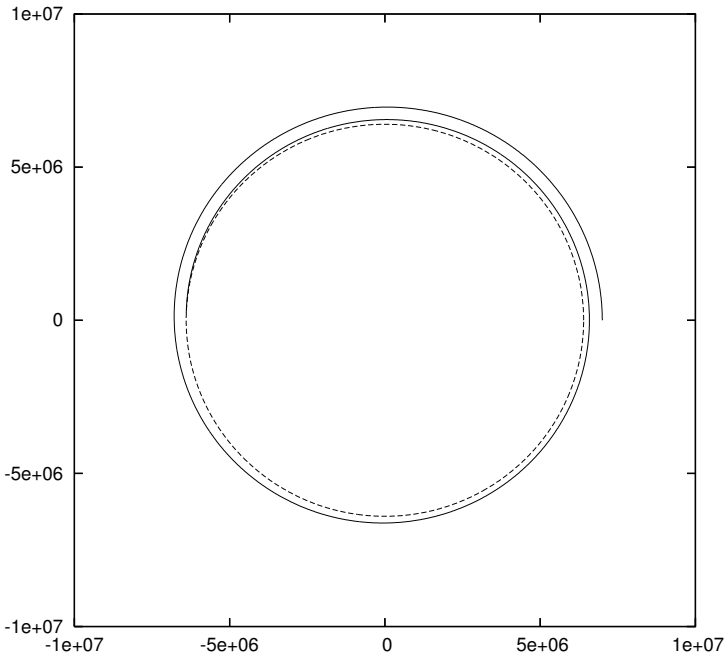

FIGURE 8 – La surface de la terre est représentée en pointillés

Remarque

On n'a pas tenu compte de l'augmentation de la densité de l'air et donc du frottement quand l'altitude diminue. Il serait très facile de le faire en remplaçant le coefficient `b` par une fonction de l'altitude.

5.4 Électron soumis à l'attraction d'un noyau fixe et à un champ homogène

Les équations sont voisines de celles du satellite :

$$\ddot{\overrightarrow{r}} = -\frac{ze^2}{4\pi\varepsilon_0 m}\frac{\overrightarrow{r}}{r^3} + \frac{e}{m}\overrightarrow{E}$$

e est la charge algébrique de l'électron, ze celle de la charge fixe, $\overrightarrow{E}$ le champ homogène.
On pose :

$$a = -\frac{ze^2}{4\pi\varepsilon_0 m}$$

$$b_x = \frac{e}{m}E_x$$

$$b_y = \frac{e}{m}E_y$$

et, comme pour le satellite :

$$\begin{aligned} q_0 &= x \\ q_1 &= y \\ q_2 &= \dot{x} \\ q_3 &= \dot{y} \end{aligned}$$

Le système s'écrit :

$$\dot{q}_0 = q_2$$

$$\dot{q}_1 = q_3$$

$$\dot{q}_2 = a\frac{q_0}{(q_0^2+q_1^2)^{\frac{3}{2}}} + b_x$$

$$\dot{q}_3 = a\frac{q_1}{(q_0^2+q_1^2)^{\frac{3}{2}}} + b_y$$

la fonction qui définit le système différentiel s'écrit :

```
void el(double* q, double t, double* qp, int n) {
  double r3;
  r3 = pow(q[0]*q[0]+q[1]*q[1], 3./2.);
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = a*q[0]/r3 + bx;
  qp[3] = a*q[1]/r3 + by;
}
```

On obtient des courbes qui ont une grande variété de formes en fonction des conditions initiales et de $\overrightarrow{E}$. Un exemple :

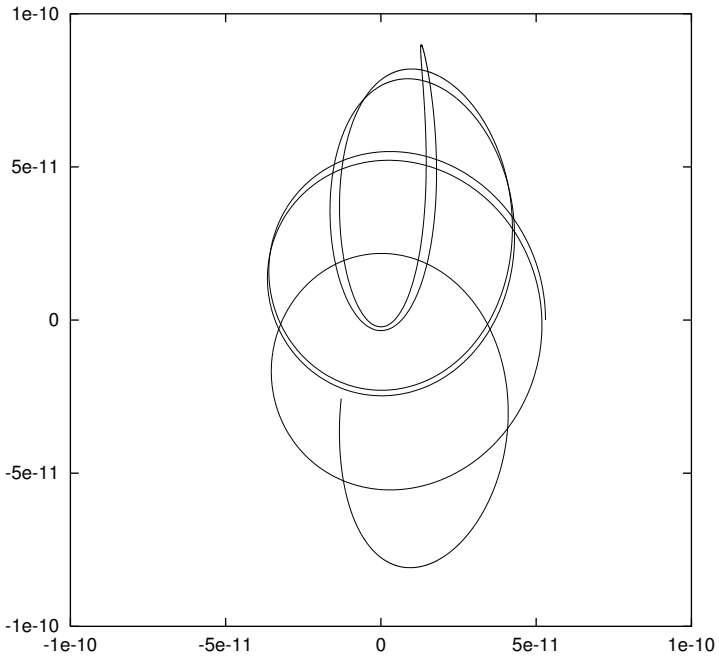

FIGURE 9 –

Remarque

Bien entendu ce système relève de la mécanique quantique (effet Stark) et non de la mécanique classique. Mais il est souvent intéressant d'étudier si des propriétés d'un système classique se retrouvent dans le système quantique correspondant.
D'autre part, les équations précédentes et les trajectoires en résultant sont exactement celles d'un satellite

placé dans le champ gravitationnel de la terre et soumis à une pression de radiation comme celle du rayonnement solaire (voile solaire).

6 Conclusion

Les formules de Runge-Kutta d'ordre 4 sont un outil adapté à l'étude de nombreux problèmes de physique. Mais :

- il faudrait ajouter un ajustement automatique du pas d'itération : s'il reste fixe il est en général inutilement petit pour certaines parties du calcul (perte de temps) et trop grand pour d'autres (perte de précision)
- même avec un ajustement du pas, l'erreur due aux approximations peut augmenter très rapidement au fur et à mesure des itérations. Dans le cas de systèmes instables, cela peut interdire tout calcul. Le résultat doit toujours être examiné d'un œil critique à défaut d'une vraie étude numérique garantissant une certaine précision. Par exemple il faut vérifier, quand c'est possible, si certains invariants du système sont respectés.

Par ailleurs il existe d'autres méthodes de résolution numérique des équations différentielles, plus efficaces pour certains problèmes.
Enfin, un autre domaine très important pour la physique est celui des équations aux dérivées partielles. Il n'est pas abordé ici et exige des techniques nettement plus compliquées.

Annexe : programmation des formules de Runge-Kutta d'ordre 4, fonction `rk4`

```
#include<bibli_fonctions.h>
void rk4(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i, k, p, PM = 4;
  static const double c2 = 1./2, c3 = 1./3, c6 = 1./6;
  /* Allocations et initialisations */
  double** a = (double**)malloc(PM*sizeof(double*));
  for(i = 0; i < PM; i++)
    a[i] = (double*)malloc(PM*sizeof(double));
  ini_D_2(a,PM,PM,c2,0.,0.,0.,0.,c2,0.,0.,0.,0.,1.,0.,c6,c3,c3,c6);
  double* b = (double*)malloc(PM*sizeof(double));
  ini_D_1(b,PM,0.,c2,c2,1.);
  double** y = (double**)malloc((PM+1)*sizeof(double*));
  for(i = 0; i < PM+1; i++)
    y[i] = (double*)malloc(n*sizeof(double));
  double** z = (double**)malloc(PM*sizeof(double*));
  for(i = 0; i < PM; i++)
    z[i] = (double*)malloc(n*sizeof(double));
  /* Calcul */
  for(i = 0; i < n; i++)
    y[0][i] = q[i];
  for(p = 1; p <= PM; p++) {
    sd(y[p-1], t+b[p-1]*dt, z[p-1], n);
    for(i = 0; i < n; i++)
      y[p][i] = q[i];
    for(k = 0; k < p; k++)
      for(i = 0; i < n; i++)
        y[p][i] = y[p][i] + dt*a[p-1][k]*z[k][i];
  }
  for(i = 0; i < n; i++)
    q[i] = y[PM][i];
  /* Desallocations */
  f_D_2(a,PM,PM); f_D_1(b,PM); f_D_2(y,PM+1,n); f_D_2(z,PM,n);
}
```

Remarque

Il ne faut pas recopier cette fonction. Elle est disponible en mettant la directive :

```
#include<bibli_fonctions.h>
```

en tête du programme qui l'utilise et en compilant et exécutant par *ccc*.

Annexe : exemple de trois masses en interaction gravitationnelle

On considère trois particules ponctuelles de masses quelconques, s'attirant selon la loi de la gravitation universelle. On se place dans un repère galiléen quelconque, d'origine O. La relation fondamentale de la dynamique appliquée à chacune des particules s'écrit :

$$m_\alpha \frac{d^2\overrightarrow{OM_\alpha}}{dt^2} = Gm_\alpha \sum_{\beta\neq\alpha} \frac{m_\beta \overrightarrow{M_\alpha M_\beta}}{(M_\alpha M_\beta)^3} \qquad \text{avec} \quad \alpha = 0,1,2 \quad \beta = 0,1,2$$

$M_\alpha M_\beta$ est la distance des particules M_α et M_β, G la constante de la gravitation universelle $6.67\ 10^{-11}$ MKSA. On note $x_{i\alpha}$ la $i^{\text{ème}}$ coordonnée cartésienne de la $\alpha^{\text{ème}}$ particule. On a alors :

$$\ddot{x}_{i\alpha} = G \sum_{\beta\neq\alpha} \frac{m_\beta (x_{i\beta} - x_{i\alpha})}{(M_\alpha M_\beta)^3} = f_{i\alpha}(x_{00}, x_{10}, \ldots, x_{22}) \qquad \text{avec} \quad i = 0,1,2 \qquad \text{(A)}$$

Posons :

$$\begin{aligned} q_0 &= x_{00} \\ q_1 &= x_{10} \\ \cdots &= \cdots \\ q_3 &= x_{01} \\ q_4 &= x_{11} \\ \cdots &= \cdots \\ q_8 &= x_{22} \end{aligned}$$

et :

$$\begin{aligned} q_9 &= \dot{q}_0 \\ q_{10} &= \dot{q}_1 \\ \cdots &= \cdots \\ q_{17} &= \dot{q}_8 \end{aligned}$$

Les équations (A) peuvent alors s'écrire :

$$\begin{aligned} \dot{q}_0 &= q_9 \\ \dot{q}_1 &= q_{10} \\ \cdots &= \cdots \\ \dot{q}_8 &= q_{17} \\ \dot{q}_9 &= f_{00}(q_0 \ldots q_8) \\ \dot{q}_{10} &= f_{10}(q_0 \ldots q_8) \\ \cdots &= \cdots \\ \dot{q}_{17} &= f_{22}(q_0 \ldots q_8) \end{aligned}$$

On est donc ramené à un système de la forme $\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q})$ avec un vecteur $\overrightarrow{q}$ à 18 composantes [4], dont les solutions peuvent être calculées en utilisant la fonction `rk4`, compte-tenu des réserves énoncées dans la conclusion du chapitre.

4. Dans ce cas particulier la fonction $\overrightarrow{f}$ ne dépend pas explicitement du temps.