UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES

Méthodes informatiques pour physiciens

introduction à C++ et
résolution de problèmes de physique par ordinateur

Corrigé 8

Professeur : Alessandro Bravar
Alessandro.Bravar@unige.ch

Université de Genève
Section de Physique

Semestre de printemps 2015

Références :

M-Y. Bachmann, H. Catin, P. Epiney *et al.*
Méthodes numériques

A. Quateroni, F. Saleri et P. Gervasio
Calcul scientifique

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery
Numerical Recipes

`http://dpnc.unige.ch/~bravar/C++2015/L8` :
pour les notes du cours, les exercices et les corrigés

8.1 Problèmes

Les programmes développés ici sont un peu plus compliqués et plus longs qu'avant. Ce-ci est du au fait que on a implmenté un logiciel graphique pour dessiner les trajectoires en fonction du temps. Autrement, la difficulté est la même.

1. **Désintégration radioactive**
   Voir le programme **Decay.cpp** développé pendant la leçon.

2. **Chaine de désintégration**
   Les équation qui décrivent les populations des noyaux A et B en fonction du temps sont :

$$\frac{\mathrm{d}N_A(t)}{\mathrm{d}t} = -\frac{1}{\tau_A}N_A(t) \tag{1}$$

$$\frac{\mathrm{d}N_B(t)}{\mathrm{d}t} = -\frac{1}{\tau_B}N_B(t) + \frac{1}{\tau_A}N_A(t) \tag{2}$$

   avec les conditionnes initiales (i.e. nombre des noyaux de chaque espèce au départ) $N_A(0) = N_A^0$, $N_B(t) = N_B^0$, $N_C^0 = 0$.

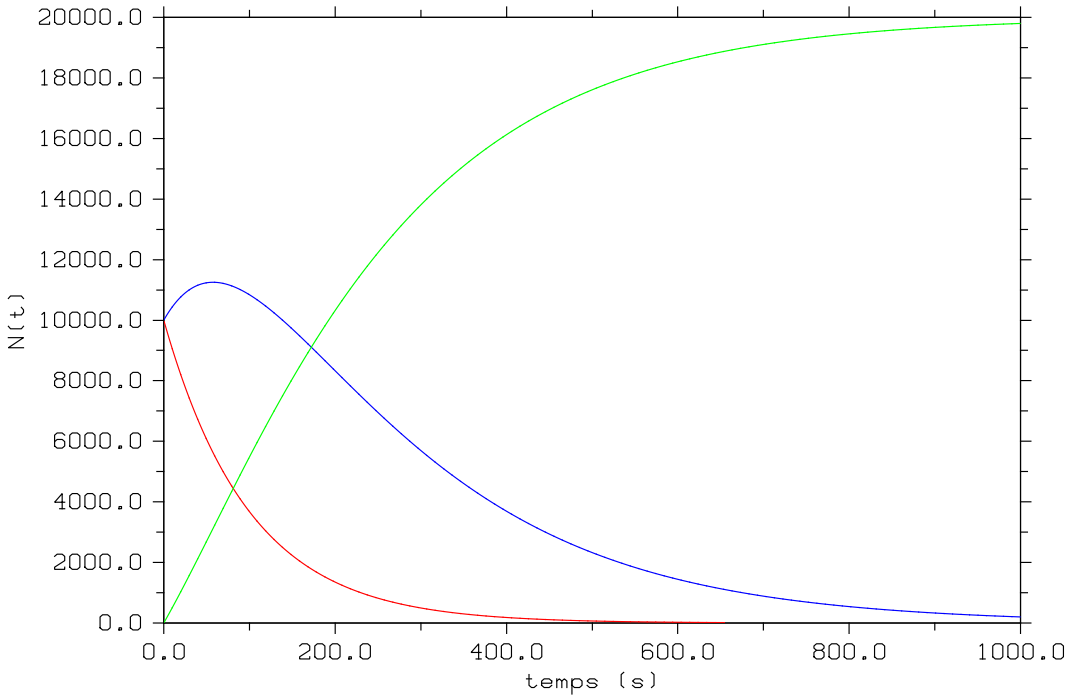

FIGURE 1 – Population des noyaux A (en rouge), B (en bleu) et C (en vert) en fonction du temps. Au début la population des noyaux B augment car $\tau_B > \tau_A$. A la fin, la population des noyaux C est égal à la somme de noyaux initiales A et B car tous les noyaux A et B se sont désintégrés.

Decay2_Euler.cpp

```
1 #include <iostream>
2 #include "dislin.h"
3
4 using namespace std;
5
6 int main () {
```

```
7     const float dt = 0.1;
8     const int nPasMax= int(1000/dt) + 1;
9
10    double A[nPasMax];
11    double B[nPasMax];
12    double C[nPasMax];
13    double temps[nPasMax];
14
15    //conditiones initiales
16    A[0]=10000;
17    B[0]=10000;
18    C[0]=0;
19    const double tauA = 100.;
20    const double tauB = 200.;
21    temps[0]=0.;
22
23    for (int i=0; i<nPasMax-1; i++) {
24      A[i+1] = A[i] - 1./tauA *A[i]*dt;
25      B[i+1] = B[i] - 1./tauB *B[i]*dt;
26      //on calcule C a partir de nayaux B
27    //qui se sont desintegres entre t et t + Delta t
28      C[i+1] = C[i] + 1./tauB *B[i]*dt;
29      //on ajout a B les noyaoux A
30    //qui se sont desintegres entre t et t + Delta t
31      B[i+1] = B[i+1] + 1./tauA *A[i]*dt;
32
33      temps[i+1] = temps[i]+dt;
34    }
35
36    //initialisation DISLIN
37    metafl("XWIN");
38    disini();
39    name("temps (s)", "X");
40    name("N(t)", "Y");
41    graf(0., 1000., 0., 200., 0., 20000., 0., 2000.);
42    title();
43    //dessin noyaux A
44    color("RED");
45    marker(1);
46    curve(temps, A, nPasMax-1);
47    //dessin noyaux B
48    color("BLUE");
49    marker(1);
50    curve(temps, B, nPasMax-1);
51    //dessin noyaux C
52    color("GREEN");
53    marker(1);
54    curve(temps, C, nPasMax-1);
55    //fin de DISLIN
56    disfin();
57
58    return 0;
59  }
```

3. **Frottement**
   Voir les programmes **Frottement.cpp** développés pendant la leçon.
   Pour le cas du parachute, voir le programme **Chute.cpp**. Ce programme calcule la vitesse terminale d'un objet accéléré par la gravité et ralenti par la résistance à l'air. L'utilisateur

doit saisir le coefficient de résistance à l'air et la hauteur initiale ; la vitesse initiale quant à elle est zéro.
En considérant les deux forces qui agissent sur le parachute l'on peut calculer la position verticale et la vitesse à chaque pas, p.ex. en utilisant la méthode de Runge. Avec la méthode de Runge, on modifie d'abord les conditions initiales selon :

$$\begin{aligned} y_0 &= y_0 \\ v_0 &= v_0 - a\ \Delta t/2 \end{aligned}$$

où $a = -G_N$ est l'accélération au temps t_0. Cette modification nous permettra de calculer $v_1 = v(\Delta t/2)$, $v_2(\Delta t + \Delta t/2)$, ... avec le même algorithme utilisé pour prédire $x_1 = x(\Delta t)$, $x_2(2\Delta t)$, ... Ensuite on intègre la trajectoire avec l'algorithme suivant (d'abord la vitesse, puis la position) :

$$\begin{aligned} v_n &= v_{n-1} + \left(\frac{C}{m}\ v_{n-1}^2 - G_N\right)\Delta t \\ y_n &= y_{n-1} + v_n\ \Delta t \end{aligned}$$

où C est le coefficient de frottement à l'air, et Δt le pas d'intégration. Le programme affiche un message quand la vitesse terminale est atteinte. On considère la vitesse terminale atteinte quand la vitesse change moins de 0.01% dans l'intervalle Δt. La boucle continue jusqu'à que le parachutiste atteigne le sol.

Chute.cpp

```
1  //chute d'un objet dans l'air
2  #include <iostream>
3  #include <cmath>
4  #include <fstream>
5
6  using namespace std;
7
8  const double G = 9.81; //acceleration gravitationnelle m/s^2
9
10 int main() {
11   const int STEPS = 50000;
12   double ypos[STEPS];    //position
13   double yvel[STEPS];    //vitesse
14   double temps[STEPS];
15
16   //conditions intiales
17   double h;
18   cout << "Entrez l'hauteur initiale (m) : ";
19   cin >> h;
20   ypos[0] = h;          //position initiale
21   yvel[0] = 0.;         //vitiesse initiale
22
23   //resistance de l'air
24   double drag = 0.1;
25   cout << "Entrez le coefficient de resistance dans l'air : ";
26   cin >> drag;
27
28   //calcul de la trajectoire (chute)
29   double tau;
30   cout << "Entrez un pas de temps pcour l'integration, tau (sec) : ";
31   cin >> tau;
32
33   double yacc;
```

```
34    //methode de Runge
35    yacc = drag * pow(yvel[0],2) - G;
36    yvel[0] = yvel[0] - yacc*tau/2.;
37    //on boucle tant que l'objet ne touche pas le sol
38    int count = 1;
39    bool term = false;
40    for (int istep=1; istep<STEPS; istep++) {
41      //acceleration
42      yacc = drag * pow(yvel[istep-1],2);    //nul si pas de frottement
43      yacc = yacc - G;           //addition de la composante gravitationnelle
44      //mise a jour de la vitesse
45      yvel[istep] = yvel[istep-1] + yacc*tau;
46      //mise a jour de la position
47      ypos[istep] = ypos[istep-1] + yvel[istep]*tau;
48      temps[istep] = istep*tau;
49
50      //velocite termianle : < 0.01% da variation
51      if ((!term) && abs((yvel[istep]-yvel[istep-1])/yvel[istep])<0.0001) {
52        cout << "Veleocite terminale : " << yvel[istep]
53          << " m/s attainte apres " << count*tau << " s a "
54          << ypos[istep] << " m." << endl;
55        term = true;
56      }
57
58      //on sort de la boucle si l'objet atteint le sol
59      if (ypos[istep]<0.) break;
60
61      count++;
62    }
63    cout << "Velocite finale au sol : " << yvel[count] << " m/s apres "
64      << count*tau << " s de chute." << endl;
65    if (!term)
66      cout << "On n'a pas attainte la velocite terminale :-(" << endl;
67
68    //sortie des positions dans un fichier text
69    ofstream trajectory("chute.dat");
70    for (int i=0; i<count; i++)
71      trajectory << ypos[i] << "\t" << yvel[i] << "\t" << temps[i] << endl;
72
73    return 0;
74  }
```

4. **Trajectoire**
   La trajectoire d'un projectile est décrite par l'équation du mouvement suivante :

$$m\frac{d^2\mathbf{r}}{dt^2} = -m\mathbf{g} \tag{3}$$

On peut remplacer les opérateurs différentiels par des différences finies, i.e. :

$$\Delta\mathbf{r} = \mathbf{v}\Delta t \tag{4}$$

$$\Delta\mathbf{v} = \frac{\mathbf{F}}{m}\Delta t \tag{5}$$

L'algorithme d'Euler nous permet donc d'intégrer les équations du mouvement :

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_n\Delta t \tag{6}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \frac{\mathbf{F}(\mathbf{r}_n, \mathbf{r}_n,)}{m}\Delta t \tag{7}$$

Avec les conditions initiales pour $\mathbf{r}_0$ et $\mathbf{v}_0$ on trouve ainsi facilement la solution du problème même pour les forces qui dépendent de la position et de la vitesse de manière compliquée (i.e. $F \propto v^2$).

Voir les programmes **Trajectoire.cpp** developpés pendant le cours.

La première version du programme utilise la méthode d'Euler pour intégrer numériquement la trajectoire d'un projectile, avec frottement et sans frottement. La solution analytique est également développée dans le programme. La deuxième version utilise la méthode de Runge.

Quel que soit la méthode de résolution (analytique, numérique), la position du projectile est initialisé à 0 ($x_0 = 0$ et $y_0 = 0$). L'utilisateur doit saisir la vitesse initiale et l'angle de jetée ϑ, dont on déduit $v_{x,0} = v_0 \cos\vartheta$ et $v_{y,0} = v_0 \sin\vartheta$. Le coefficient de frottement C et le pas Δt sont aussi des paramètres donnés par l'utilisateur.

A chaque pas Δt les coordonnées du projectile sont mise a jour selon l'algorithme :

$$x_{n+1} = x_n + v_{x,n}\ \Delta t$$
$$y_{n+1} = y_n + v_{y,n}\ \Delta t$$

On définit v_n le module de la vitesse $v_n = \sqrt{v_{x,n}^2 + v_{y,n}^2}$, les composantes de la vitesse à l'instant $(n+1)\Delta t$ s'écrivent alors :

$$v_{x,n+1} = v_{x,n} - \frac{C\ v_n\ v_{x,n}}{m}\Delta t$$
$$v_{y,n+1} = v_{y,n} - \left(\frac{C\ v_n\ v_{y,n}}{m} - G_N\right)\Delta t$$

Si l'on néglige les frottements $C = 0$, l'accéleration suivant x devient nulle, reste l'accéleration liée à l'apesanteur suivant y uniquement et la trajectoire est une parabole inversée.

Pour intégrer la trajectoire avec la méthode de Runge, d'abord on redéfinit la vitesse initiale et on inverse l'ordre du calcul : d'abord la vitesse, puis la position.

5. **Le parachute**

Ce problème est une variation identique à la chute d'un objet étudiée plus tôt. La force de frottement, qui ralentit la chute de l'objet, est toujours $\propto \rho v^2$, mais il faut tenir compte du changement de la densité de l'air, qui décroit avec la hauteur selon $\rho = \rho_0 \exp(-h/h_0)$. La force de résistance à l'air est donc :

$$F_{air}(h) = +mK\rho_0 \exp\left(-\frac{h}{h_0}\right) \tag{8}$$

et l'équation différentielle décrivant la chute est :

$$m\frac{\mathrm{d}^2h}{\mathrm{d}t^2} = -mg + mK\rho_0 \exp\left(-\frac{h}{h_0}\right)\left(\frac{\mathrm{d}h}{\mathrm{d}t}\right)^2 . \tag{9}$$

La figure 2 montre la variation de la hauteur h et de la vitesse v en fonction du temps. Vu que la densité de l'air augmente en s'approchant du sol, la vitesse *terminale* décroit dans le temps.

Parachute.cpp

```
1 //chute d'un objet dans l'air
2 #include <iostream>
3 #include <cmath>
```

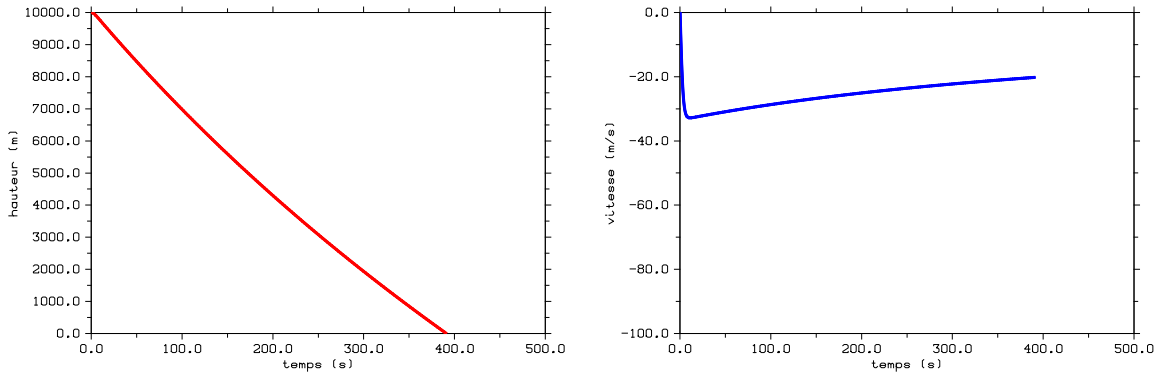

FIGURE 2 – Variation de la hauteur (à gauche) et de la vitesse (à droite) en fonction du temps.

```
4  #include <fstream>
5  #include "dislin.h"
6
7  using namespace std;
8
9  int main() {
10   //parametres
11   const double G = 9.81;
12   const double Rho0 = 1.2;
13   const double H0 = 10000.;
14   const double M = 90.;
15   const double k1 = 2.;
16   const double k2 = 5.;
17
18   const int STEPS = 75000;
19   double ypos[STEPS];    //position
20   double yvel[STEPS];    //vitesse
21   double temps[STEPS];
22
23   //conditions intiales
24   double h = 3500.;
25   cout << "Entrez l'hauteur initiale (m) : ";
26   cin >> h;
27   ypos[0] = h;          //position initiale
28   yvel[0] = 0.;         //vitiesse initiale
29
30   //calcul de la trajectoire (chute)
31   double dT;
32   cout << "Entrez un pas de temps pour l'integration, dT (sec) : ";
33   cin >> dT;
34
35   double drag1, drag2, yacc;
36   //methode de Runge
37   drag1 = k1/M * 1./50. * Rho0 * exp(-ypos[0]/H0);
38   drag2 = k2/M * 1./50. * Rho0 * exp(-ypos[0]/H0);
39   yacc = drag1 * yvel[0] + drag2 * yvel[0]*yvel[0]- G;
40   yvel[0] = yvel[0] - yacc*dT/2.;
41   //on boucle tant que l'objet ne touche pas le sol
42   int count = 1;
43   bool term = false;
44   for (int istep=1; istep<STEPS; istep++) {
45     //acceleration
46     if (temps[istep-1] < 60.) {      //parachute ferme
47       drag1 = k1/M * 1./50. * Rho0 * exp(-ypos[istep-1]/H0);
48       drag2 = k2/M * 1./50. * Rho0 * exp(-ypos[istep-1]/H0);
```

```
49        yacc = drag1 * yvel[istep-1] + drag2 * yvel[istep-1]*yvel[istep-1];
50      } else {      //parachute ouvert
51        drag1 = k1/M * Rho0 * exp(-ypos[istep-1]/H0);
52        drag2 = k2/M * Rho0 * exp(-ypos[istep-1]/H0);
53        yacc = drag1 * yvel[istep-1] + drag2 * yvel[istep-1]*yvel[istep-1];
54      }
55      yacc = yacc - G;          //addition de la composante gravitationnelle
56      //mise a jour de la vitesse
57      yvel[istep] = yvel[istep-1] + yacc*dT;
58      //mise a jour de la position
59      ypos[istep] = ypos[istep-1] + yvel[istep]*dT;
60      temps[istep] = istep*dT;
61
62      //velocite termianle : < 0.01% da variation
63      if ((!term) && abs((yvel[istep]-yvel[istep-1])/yvel[istep])<0.0001) {
64        cout << "Veleocite terminale : " << yvel[istep]
65          << " m/s attainte apres " << count*dT << " s a "
66          << ypos[istep] << " m." << endl;
67        term = true;
68      }
69
70      //on sort de la boucle si l'objet atteint le sol
71      if (ypos[istep]<0.) break;
72
73      count++;
74    }
75    cout << "Velocite finale au sol : " << yvel[count] << " m/s apres "
76      << count*dT << " s de chute." << endl;
77    if (!term)
78      cout << "On n'a pas attainte la velocite terminale :-(" << endl;
79
80    //initialisation de DISLIN
81    metafl("XWIN");       //XWIN ou PDF
82    disini();
83    //dessin de la trajectoire
84    name("temps (s)", "X");
85    name("hauteur (m)", "Y");
86    graf(0.,500.,0.,100.,0.,10000.,0.,1000.);
87    thkcrv(10);
88    color("RED");
89    curve(temps, ypos, count);
90    color("FORE");
91    endgrf();
92    //dessin de la vitesse
93    newpag();
94    name("temps (s)", "X");
95    name("vitesse (m/s)", "Y");
96    graf(0.,500.,0.,100.,-200.,0.,-200.,50.);
97    thkcrv(10);
98    color("BLUE");
99    curve(temps, yvel, count);
100   //fin de DISLIN
101   disfin();
102
103   return 0;
104 }
```

6. Evolution des popoulations

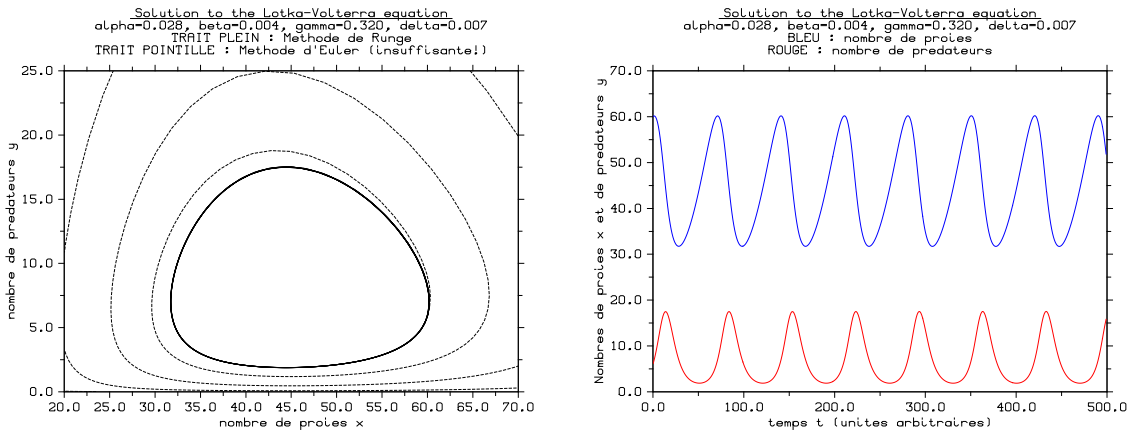

FIGURE 3 – Solutions de l'équation de Lotke-Volterra dans le plan xy (à gauche) et en fonction du temps t (à droite).

Volterra.cpp

```
#include "dislin.h"
#include <iostream>
#include <stdio.h>

//Quelques constantes pour l'equation de Lotka-Volterra
double alpha = 7;
double beta  = 1;
double gamma = 80;
double delta = 1.8;

//Un facteur por modifier les constantes
double f = 0.004;

//Premiere equation de Lotka-Volterra
double dxdt(double x, double y) {
    return f*(+ alpha*x - beta*x*y);
}

//Deuxieme equation de Lotka-Volterra
double dydt(double x, double y) {
  return f*(- gamma*y + delta*x*y);
}

int main() {

  //Tableaux pour les temps
  double t[500] = {0};

  //Tableaux pour les solutions utilisant le methode de Runge
  double x[500] = {0};  //Nombre de proies
  double y[500] = {0};  //Nombre de predateurs

  //Tableaux pour les solutions utilisant le methode d'Euler
  double x_Euler[500] = {0};
  double y_Euler[500] = {0};

  //Conditions initiales
  x[0] = 60;
  y[0] = 6;
```

```
40
41    //Meme conditions initiales pour la solution utilisant le methode d'Euler
42    x_Euler[0] = 60;
43    y_Euler[0] = 6;
44
45    //Boucle dans le temps
46      for (int time=1; time<500; ++time) {
47
48      //Definissez le temps
49      t[time] = time;
50
51      //Euler
52      x_Euler[time] = x_Euler[time-1] + dxdt(x_Euler[time-1], y_Euler[time
            -1]);
53      y_Euler[time] = y_Euler[time-1] + dydt(x_Euler[time-1], y_Euler[time
            -1]);
54
55      //Runge
56      double xm = x[time-1] + 0.5*dxdt(x[time-1],y[time-1]);
57      double ym = y[time-1] + 0.5*dydt(x[time-1],y[time-1]);
58      x[time] = x[time-1] + dxdt(xm,ym);
59      y[time] = y[time-1] + dydt(xm,ym);
60    }
61
62    //Ecrivez les constants aux graphiques
63    char constants[40];
64    sprintf(constants, "alpha=%.3f, beta=%.3f, gamma=%.3f, delta=%.3f", alpha
          *f, beta*f, gamma*f, delta*f);
65
66    //Premiere graphique
67    metafl("PDF");  disini();
68    name("temps t (unites arbitraires)", "x");
69    name("Nombres de proies x et de predateurs y", "y");
70    graf(0.,500.,0.,100.,0.,70.,0.,10.);
71    titlin("Solution to the Lotka-Volterra equation",-1);
72    titlin(constants,2);
73    titlin("BLEU : nombre de proies",3);
74    titlin("ROUGE : nombre de predateurs",4);
75    title();
76    color("blue");  curve(t, x, 500);
77    color("red"); curve(t, y, 500);
78    disfin();
79
80    //Deuxieme graphique
81    metafl("PDF");  disini();
82    name("nombre de proies x", "x");
83    name("nombre de predateurs y", "y");
84    graf(20.,70.,20.,5.,0.,25.,0.,5.);
85    titlin("Solution to the Lotka-Volterra equation",-1);
86    titlin(constants,2);
87    titlin("TRAIT PLEIN : Methode de Runge", 3);
88    titlin("TRAIT POINTILLE : Methode d'Euler (insuffisante!)", 4);
89    title();
90    curve(x, y, 500);
91    dash(); curve(x_Euler, y_Euler, 500);
92    disfin();
93
94    return 0;
95  }
```