UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES

Méthodes informatiques pour physiciens

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

Corrigé 5

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.* **(CRM)**
Méthodes numériques

W.H. Press, S.A. Teukolsky *et al.*
Numerical Recipes

`http://dpnc.unige.ch/~bravar/C++2015/L5` :
pour les notes du cours, les exercices et les corrigés

5.1 Problèmes d'intégration numérique

1. **Intégrale d'un fonction quelconque**
   On cherche ici à intégrer une fonction. On utilise une fonction `C++` pour définir la fonction à intégrer. On a choisi un polynôme du second degré avec les coefficients fixés. Les bornes de l'intervalle d'intégration, ainsi que le nombre de divisions de l'intervalle sont entrés par l'utilisateur. On utilise la méthode du trapèze et la méthode de Simpson pour l'intégration.
   Integrale.cpp

```cpp
#include <iostream>
#include <cmath>

using namespace std;

//declaration de la fonction a integrer
double fonc(double);

int main() {
  //saisi des parametres
  double min, max;
  cout << "Entrez un interval d'integration : " << endl;
  cout << "min = ";   cin >> min;
  cout << "max = ";   cin >> max;
  int div;
  cout << "Entrez un nombre de suos-divisions :" << endl;
  cout << "N = ";   cin >> div;
  double deltax = (max-min)/div;

  double intTrapeze = 0.;
  double intSimpson = 0.;
  for (int i=0; i<=div; i++) {
    //methode du trapeze
    intTrapeze += (fonc(min+i*deltax) +
                   fonc(min+(i+1)*deltax));
    //methode de Simpson
    intSimpson += (fonc(min+deltax*i) +
                   4.*fonc(min+(i+0.5)*deltax) +
                   fonc(min+(i+1)*deltax));
  }
  intTrapeze *= deltax/2.;
  intSimpson *= deltax/6.;
  cout << "Integrale avec la methode du trapeze: " << intTrapeze << endl;
  cout << "Integrale avec la methode de Simpson: " << intSimpson << endl;

  return 0;
}

//definition de la fonction a integrer
double fonc(double x) {
  //dans cet example nous avons choisi un polynome de 2eme degree;
  double a, b, c;
  a = 10.;
  b = -3.;
  c = 7.;

  double y = a*(x*x) + b*x + c;

  return y;
}
```

2. **Intégrale des fonctions sinus et cosinus sur l'intervalle** $[0 - \pi]$
   Le programme ci-dessous calcule l'intégrale des fonctions `sin(x)` et `cos(x)` à l'aide des méthodes du trapèze et de Simpson respectivement.
   Les bornes de l'intervalle d'intégration sont fixées $[0 - \pi]$, mais le nombre de division de l'intervalle est rentré par l'utilisateur.
   IntSinus.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  int main() {
7    cout << "Integration de sin(x) et cos(x) sur [0-Pi]" << endl;
8
9    const int N = 100000;
10   double deltax = M_PI/N;
11
12   //integral du sinus et du cosinus avec la methode du rectangle (a gauche)
13   double intSinR = 0., intCosR = 0.;
14   for (int i=0; i<N; i++) {
15     intSinR += sin(deltax*i);
16     intCosR += cos(deltax*i);
17   }
18   intSinR *= deltax;
19   intCosR *= deltax;
20   cout << "Integrale de sin(x) avec la methode du rectangle : "
21        << intSinR << endl;
22   cout << "Integrale de cos(x) avec la methode du rectangle : "
23        << intCosR << endl;
24
25   //integral du sinus et du cosinus avec la methode du trapeze
26   double intSinT = 0., intCosT = 0.;
27   for (int i=0; i<=N; i++) {
28     intSinT += sin(deltax*i) + sin(deltax*(i+1));
29     intCosT += cos(deltax*i) + cos(deltax*(i+1));
30   }
31   intSinT *= deltax/2.;
32   intCosT *= deltax/2.;
33   cout << "Integrale de sin(x) avec la methode du trapeze : "
34        << intSinT << endl;
35   cout << "Integrale de cos(x) avec la methode du trapeze : "
36        << intCosT << endl;
37
38   //integral du sinus et du cosinus avec la methode de Simpson
39   double intSinS=0., intCosS=0.;
40   for (int i=0; i<=N; i++) {
41     intSinS += sin(deltax*i) + 4*sin(deltax*(i+0.5)) + sin(deltax*(i+1));
42     intCosS += cos(deltax*i) + 4*cos(deltax*(i+0.5)) + cos(deltax*(i+1));
43   }
44   intSinS *= deltax/6.;
45   intCosS *= deltax/6.;
46   cout << "Integrale de sin(x) avec la methode de Simpson : "
47        << intSinS << endl;
48   cout << "Integrale de cos(x) avec la methode de Simpson : "
49        << intCosS << endl;
50
51   return 0;
52 }
```

Pour un même nombre des divisions, l'on peut constater que les valeurs obtenues avec la méthode de Simpson sont plus proches des valeurs exactes des intégrales (i.e. 2 pour l'intégrale du sinus et 0 pour l'intégrale du cosinus). En outre, plus N augmente, plus on se rapproche des valeurs exactes (on augmente la précision du calcul).

3. **Dérivation des formules du trapèze et de Simpson**
La **formule du trapèze** s'obtient facilement en considérant l'aire du trapèze formé par les quatre points (x_i, 0), (x_i+Δx, 0), (x_i, f(x_i)) et (x_i+Δx, f(x_i+Δx)). x_i est donné par $x_i = a + i \times \Delta x$ et $\Delta x = \frac{|a-b|}{N}$. En additionnant l'aire de la partie *rectangulaire* et l'aire de la partie *triangulaire* on obtient :

$$A_i = A_i^R + A_i^T = f(x_i)\Delta x + \frac{f(x_i + \Delta x) - f(x_i)}{2}\Delta x \tag{1}$$

c'est-à-dire :

$$A_i = \frac{(f(x_i) + f(x_i + \Delta x)}{2}\Delta x \tag{2}$$

Dans cette méthode, on interpole la fonction considérée par un polynôme de degré 1 sur l'intervalle d'intégration. Ceci ne requiert que deux points d'interpolation, à savoir les bornes de l'intervalle. L'aire définiée par le polynôme d'interpolation est donc ici simplement un trapèze, d'où le nom de la méthode. L'intégrale s'obtient en sommant les aires de tous les trapèzes sous la courbe et donc :

$$I = \frac{\Delta x}{2}\sum_{i=0}^{N-1}[f(x_i) + f(x_i + \Delta x)] \, . \tag{3}$$

On peut réduire le nombre des calcules et donc améliorer l'algorithme, en réécrivant la formule précédente comme :

$$I = f(a)\frac{\Delta x}{2} + \Delta x \sum_{i=1}^{N-1} f(x_i) + f(b)\frac{\Delta x}{2} \, . \tag{4}$$

Pour dériver la **formule de Simpson**, on commence avec l'expansion de la fonction $f(x)$ en série de Taylor autour du point x_0. La valeur de f en x est donnée par :

$$f(x) = f(x_0) + f(x_0)'(x - x_0) + \frac{1}{2}f(x_0)''(x - x_0)^2 + \frac{1}{6}f(x_0)'''(x - x_0)^3 + ... \tag{5}$$

Ensuite on intègre l'expansion de la fonction en série de Taylor sur l'intervalle $\left[x_0 - \frac{\Delta x}{2},\ x_0 + \frac{\Delta x}{2}\right]$, symétrique autour de x_0 :

$$\int_{x_0-\frac{\Delta x}{2}}^{x_0+\frac{\Delta x}{2}} f(x)dx = \int_{x_0-\frac{\Delta x}{2}}^{x_0+\frac{\Delta x}{2}} \left(f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \frac{1}{6}f'''(x_0)(x - x_0)^3 + ... \right) dx \tag{6}$$

En raison de la symétrie autour de x_0, seule les intégrales avec dérivées paires sont différents de zéro :

$$\int_{x_0-\frac{\Delta x}{2}}^{x_0+\frac{\Delta x}{2}} f(x)dx = f(x_0)\Delta x + \frac{1}{6}f''(x_0)\frac{2\Delta x^3}{8} + O(\Delta x^5) \tag{7}$$

Maintenant nous avons besoin d'une expression numérique pour la dérivée seconde $f''(x_0)$. On approxime la fonction $f(x)$ en $x_0 - \frac{\Delta x}{2}$ et $x_0 + \frac{\Delta x}{2}$ avec l'expansion en série de Taylor :

$$f(x_0 - \frac{\Delta x}{2}) = f(x_0) - f'(x_0)\frac{\Delta x}{2} + \frac{1}{2}f''(x_0)\left(\frac{\Delta x}{2}\right)^2 - \frac{1}{6}f'''(x_0)\left(\frac{\Delta x}{2}\right)^3 + ... \quad (8)$$

$$f(x_0 + \frac{\Delta x}{2}) = f(x_0) + f'(x_0)\frac{\Delta x}{2} + \frac{1}{2}f''(x_0)\left(\frac{\Delta x}{2}\right)^2 + \frac{1}{6}f'''(x_0)\left(\frac{\Delta x}{2}\right)^3 + ... \quad (9)$$

et on additionne les deux :

$$f''(x_0) = \frac{f(x_0 - \frac{\Delta x}{2}) - 2f(x_0) + f(x_0 + \frac{\Delta x}{2})}{(\frac{\Delta x}{2})^2} + O(\Delta x^2) \quad (10)$$

Après substitution de $f''(x_0)$ dans l'intgrale précédente avec l'expression donnée ci-dessus, on arrive à ($\Delta x^3 \times O(\Delta x^2) = O(\Delta x^5)$:

$$\begin{aligned}\int_{x_0-\frac{\Delta x}{2}}^{x_0+\frac{\Delta x}{2}} f(x)dx &= f(x_0)\Delta x + \frac{1}{6}\frac{f(x_0 - \frac{\Delta x}{2}) - 2f(x_0) + f(x_0 + \frac{\Delta x}{2})}{\Delta x^2/4}\frac{2\Delta x^3}{8} + O(\Delta x^5) \\ &= f(x_0)\Delta x + \frac{1}{6}\left(f(x_0 - \frac{\Delta x}{2}) - 2f(x_0) + f(x_0 + \frac{\Delta x}{2})\right)\Delta x + O(\Delta x^5) \\ &= \frac{1}{6}\left(f(x_0 - \frac{\Delta x}{2}) + 4f(x_0) + f(x_0 + \frac{\Delta x}{2})\right)\Delta x + O(\Delta x^5) \quad (11)\end{aligned}$$

En conclusion, ignorant les termes d'ordre supérieurs à Δx^4, l'intégrale de $f(x)$ sur l'intervalle $\left[x_0 - \frac{\Delta x}{2},\ x_0 + \frac{\Delta x}{2}\right]$ est donnée par

$$\int_{x_0-\frac{\Delta x}{2}}^{x_0+\frac{\Delta x}{2}} f(x)dx = \frac{1}{6}\left(f(x_0 - \frac{\Delta x}{2}) + 4f(x_0) + f(x_0 + \frac{\Delta x}{2})\right)\Delta x \quad (12)$$

Enfin, on obtient l'inégrale de $f(x)$ sur l'intervalle $[a,\ b]$:

$$\int_a^b f(x)dx = \frac{\Delta x}{6}\sum_{i=0}^{N-1}\left(f(x_i - \frac{\Delta x}{2}) + 4f(x_i) + f(x_i + \frac{\Delta x}{2})\right) \quad (13)$$

avec $\Delta x = \frac{(b-a)}{N}$, où N est le nombre de divisions de l'intervalle $[a,\ b]$ et $x_i = a + \frac{\Delta x}{2} + i \times \Delta x$. A noter les coefficients de f : le point au milieu pèse 4 fois plus que les points aux extrémités.

4. **Volume de rotation**
Comme pour une aire, dans le cas d'un volume de révolution, nous avons $V = \int_a^b dV$ où $dV = \pi f^2(x)dx$ est le volume élémentaire (c. à d. un cylindre de rayon $f(x)$ et d'épaisseur dx) lors de la rotation autour de l'axe des x de la surface limitée par l'axe x, la courbe $y = f(x)$ et les droites $x = a$ et $x = b$. Soit :

$$V = \int_a^b \pi f^2(x)dx \quad (14)$$

Cette intégrale est calculée à l'aide de la méthode du trapèze et de la méthode de Simpson.
VolumeRotation.cpp

```
1  //volume de rotatation
2  #include <iostream>
3  #include <cmath>
4  #include <iomanip>
5
6  using namespace std;
7
8  //prototype de la fonction
9  double fonc(double x);
10
11 int main() {
12   //definition de l'intervalle d'intgration
13   double xmin, xmax;
14   xmin = 0.0;
15   xmax = 1.0;
16
17   int div;
18   cout << "Entrez le nombre de divisions :";
19   cin >> div;
20   double deltax = (xmax-xmin)/div;
21
22   double intTrapeze = 0.;
23   double intRectangle = 0.;
24   for(int i=0; i<div; i++) {
25     intRectangle += fonc(xmin + i* deltax + deltax/2.) * deltax;
26     intTrapeze += (fonc(xmin+deltax*i) + fonc(xmin+deltax+deltax*i)) *
27                   deltax/2.;
28   }
29   cout << "Le volume de revolution (meth. rectangle) est : "
30        << setprecision(8) << intRectangle << endl;
31   cout << "Le volume de revolution (meth. trapeze) est : "
32        << setprecision(8) << intTrapeze << endl;
33
34   return 0;
35 }
36
37 // definition de la fonction
38 double fonc(double x) {
39   double y;
40   y = sin(x) + x*x + 2.;
41
42   double vol = M_PI * y*y;
43   return vol;
44 }
```

5. **Calcul de la distance parcourue par une particule**
   Afin d'obtenir la distance d parcourue par une particule se déplacant à la vitesse $v(t)$, on intégre la vitesse sur l'intervalle de temps choisi avec une méthode numérique (méthode du trapèze et méthode de Simpson) :

$$d = \int_{t_1}^{t_2} v(t)\mathrm{d}t$$

   Ensuite, en utilisant cette fois uniquement la methode de Simpson (que l'on sait plus précise), on cherche à effectuer le calcul de cette distance avec une précision de 1% sur le résultat. Dans la mesure ou l'on ne connait pas la valeur exacte de cette distance, on considère la précision de 1% atteinte lorsque le résultat de l'itération ne diffère pas plus

de 1% du resultat de l'itération précédente. Comme on l'a vu, le résultat de l'intégrale dépend du nombre de divisions utilisé dans le calcul. Pour un grand nombre de divisions, une précision de 1% sera atteinte rapidement.
Dans le programme ci-dessous, on commence l'intégration avec seulement 3 divisions de l'intervalle $[t_1, t_2]$.

Vitesse.cpp

```
#include <iostream>
#include <cmath>

using namespace std;

//declaration de la fonction vitesse v(t) = 3*t /sqrt(1 + 3*t*t)
double fonc(double t);

int main() {
  cout << "Calcul de la distance par integration de la vitesse" << endl;
  //saisi des parametres
  cout << "Entrez un temps d'integration : ";
  double time;   cin >> time;
  cout << "Entrez un nombre de sousdivisions : ";
  int div;   cin >> div;
  double deltat = time / div;

  double intTrapeze = 0.;
  double intSimpson = 0.;
  for (int i=0; i<div; i++) {
    intTrapeze += (fonc(deltat*i) + fonc(deltat*(i+1)));
    intSimpson +=
        (fonc(deltat*i) + 4.*fonc(deltat*(i+0.5)) + fonc(deltat*(i+1)));
  }
  intTrapeze *= deltat/2.;
  intSimpson *= deltat/6.;

  double distanceT = intTrapeze;
  double distanceS = intSimpson;
  cout << "Integrale de la vitesse avec la methode du trapeze : "
       << distanceT << " metres" << endl;
  cout << "Integrale de la vitesse avec la methode de Simpson : "
       << distanceS << " metres" << endl;

  //Calcul de l'integrale avec 1% de precision avec la methode de Simpson :
  //on definit la precision a 1% comme etant l'iteration au-dela de
  //laquelle le resultat ne differe pas plus d'1% du resultat de
  //l'iteration precedente
  double distOld = 0.1;
  while ((fabs(distanceS-distOld)/distOld) > 0.001) {
    div +=1;
    deltat = time / div;
    distOld = distanceS;
    distanceS = 0.;

    for (int i=0; i<div; i++)
      distanceS +=
          (fonc(deltat*i) + 4.*fonc(deltat*(i+0.5)) + fonc(deltat*(i+1)));
    distanceS *= deltat/6.;

    cout << "N = " << div << " , distance = " << distanceS
         << " , precision = " << fabs(distanceS-distOld)/distOld << endl;
  } //fin de la boucle while
```

```
54
55    cout << "Precision de 0.1% atteinte pour " << div
56         << " sousdivisions" << endl;
57
58    return 0;
59  }
60
61  //definition de la fonction vitesse v(t) = 3*t /sqrt(1 + 3*t*t)
62  double fonc(double t) {
63    double vitesse;
64    vitesse = 3.*t / sqrt(1 + 3*t*t);
65
66    return vitesse;
67  }
```

6. **Frottement**
   On considère ici le cas unidimensionnel. La loi de Newton nous donne :

$$\frac{\ddot{x}}{\dot{x}} = -\frac{C}{m} \tag{15}$$

L'intégration de t_0 à t donne donc :

$$\dot{x}(t) = \dot{x}(t_0)e^{-\frac{C}{m}(t-t_0)} \tag{16}$$

Le temps théorique pour lequel la vitesse atteinte vaut $\dot{x}(0)/\sqrt{2}$ est donc :

$$t = \frac{m}{C}\ln(\sqrt{2}) \tag{17}$$

On n'utilisera pas ici la solution de l'équation différentielle, mais on intégrera l'équation du mouvement pour obtenir la vitesse. Pour intégrer l'équation du mouvement on utilise l'algorithme suivant :

$$v_{i+1} = v_i - v_i C\Delta t = v_i(1 - C\Delta T)\,, \qquad t = (i+1)\times\Delta t \tag{18}$$

à partir d'un temps initial $t_0 = 0$. L'utilisateur entre un premier choix pour le temps d'intégration. En fonction de la différence avec le rapport demandé le programme effectue l'intégrale par pas de 0.001 s et affiche le temps pour lequel la valeur de l'intégrale passe en-dessus de $1/\sqrt{2}$.

Frottement.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  int main() {
7    //on choisit des valeurs pour avoir un effet de l'ordre de 1%
8    const double C = 1.;      //coefficient de frottement
9    const double M = 100.;    //masse
10   //vitesse initiale : en realite le resultat ne depend pas de v0 !
11   double v0 = 100.;
12
13   double dt;
14   cout << "Entrez un pas d'integration (~0.01 sec.) : ";
15   cin >> dt;
16
```

```
17    double vitesse = v0;
18    double time = 0.;
19    while (vitesse > v0/sqrt(2.)) {
20      vitesse = vitesse*(1. - C/M*dt);
21      time += dt;
22    }
23
24    cout << "La vitesse demandee est atteinte apres "
25         << time << " secondes." << endl;
26    cout << "Le temps theorique est = " << M/C*log(sqrt(2.))
27         << " secondes." << endl;
28
29    return 0;
30  }
```

7. **Champ gravitationnel**
   On calcule le champ gravitationnel de la Terre en un point quelconque P en tenant compte du fait que la Terre n'est pas parfaitement sphérique. Pour cela on considère une ellipsoïde. Le champ dans le point P est calculé en sommant les contributions des sous-volumes utilisés pour diviser le volume complet (discrétisation de l'intégrale). Le programme utilise deux fonctions : la première fonction (`verifierVolume`) vérifie si un point se trouve ou non dans le volume de l'ellipsoïde, la deuxième fonction (`fieldComp`) calcule les composantes du champ dû à un sous-volume particulier.
   Vous pouvez tester le programme par exemple, en rentrant les dimensions d'une sphère parfaite, ou en rentrant les coordonnées d'un point sur un plan de symétrie.
   ChampG.cpp

```
1   //calcul du champ gravitationnel genere par un ellipsoide dans le point X
2   #include <iostream>
3   #include <cmath>
4
5   using namespace std;
6
7   bool verifierVolume(double x[], double a[]);
8   void fieldComp(double xsub[], double xp[], double f[]);
9
10  //constante globale
11  const double GN = 6.673e-11;    //m^3 kg^-1 s^-2
12
13  int main() {
14    //saisi des parametres
15    double a[3];
16    cout << "Champ gravitationnel genere par un ellipsoid dans le point X"
17         << endl << endl;
18    cout << "Entrez les dimensions de l'ellipsoide (axes en metres) :"
19         << endl;
20    cout << "a = ";    cin >> a[0];
21    cout << "b = ";    cin >> a[1];
22    cout << "c = ";    cin >> a[2];
23
24    double M = 10e20;
25    cout << "Entrez le masse de l'objet (~10e20 kg) :" << endl;
26    cin >> M;
27
28    double xp[3];
29    cout << "Entrez les coordonnees du point X (en metres) :" << endl;
30    cout << "(le point X peut etre situe a l'interieur de l'ellipsoide)"
31         << endl;
```

```
32    cout << "x = ";    cin >> xp[0];
33    cout << "y = ";    cin >> xp[1];
34    cout << "z = ";    cin >> xp[2];
35
36    int div;
37    cout << "Entrez le nombre de divisions de l'axe majeur (a) :" << endl;
38    cout << "N = ";    cin >> div;
39    double dx[3];
40    for (int i=0; i<3; i++)
41      dx[i] = a[i]/div;
42
43    //boucle sur les "sous-volumes" de l'ellipsoide :
44    //si le centre d'un sous-volume est dans l'ellipsoide
45    //on additionne sa contribution au champ total
46    double xsub[3];
47    int sousVolumes = 0;
48    double f[3];
49    double field[3] = {0.};
50    bool inside;
51    //on commence par le cote negatif des axes
52    for (int ix=0; ix<(2*div); ix++) {    //axe x
53      for (int iy=0; iy<(2*div); iy++) {    //axe y
54        for (int iz=0; iz<(2*div); iz++) {    //axe z
55          //calcule de la position du suos-volume en 3 dim
56          for (int i=0; i<3; i++)
57            xsub[i] = -a[i] + dx[i]/2. + ix*dx[i];
58
59          //on verifie que le point est dans le volume considere
60          inside = verifierVolume(xsub,a);
61          if (inside) {
62            sousVolumes += 1;
63            //contribution de ce sousvolume au champ gravitationnel :
64            //on considere GN = 1 et m = 1: F = G*m/(r*r) = 1/(r*r)
65            //il faut maintenant determiner les composantes du vecteur
66            //pour pouvoir sommer toutes les contributions
67            fieldComp(xsub,xp,f);
68            for (int i=0; i<3; i++)
69              field[i] += f[i];
70          }
71        }
72      }
73    }
74
75    cout << sousVolumes << " sous-volumes sont definis." << endl;
76    //on normalise le champ a la masse de l'objet ;
77    //M / # sous-volumes = masse sous-volume
78    double ff = 0.;
79    for (int i=0; i<3; i++) {
80      field[i] *= -GN*M/sousVolumes;
81      ff += pow(field[i],2);
82    }
83    ff = sqrt(ff);
84
85    cout << "Estimation du champ : |F| = " << ff << " m/s2 ." << endl;
86    cout << " Fx = " << field[0] << endl;
87    cout << " Fy = " << field[1] << endl;
88    cout << " Fz = " << field[2] << endl;
89
90    return 0;
91  }
```

```
92
93  bool verifierVolume(double x[], double a[]) {
94    bool inside = false;
95
96    //equation de l'ellipsoide
97    if ((pow(x[0]/a[0],2) + pow(x[1]/a[1],2) + pow(x[2]/a[2],2)) <= 1.)
98    inside = true;
99
100   return inside;
101 }
102
103 //calcul du champ gravitationnel en X
104 void fieldComp(double x[], double p[], double f[]) {
105   //calcul de la distance entre le centre du sous-volume
106   //et le point X
107   double dist = 0.;
108   for (int i=0; i<3; i++)
109     dist += pow(p[i]-x[i],2);
110   dist = sqrt(dist);
111
112   //composantes de G non normalisees
113   for (int i=0; i<3; i++)
114     f[i] = (p[i]-x[i]) / pow(dist,3);
115
116   return;
117 }
```

8. **Moment d'inertie**
   Le programme calcule le moment d'inertie pour une sphère à densité non-uniforme et de rayon R. L'axe de rotation de la sphère étant z, on définit le moment d'inertie par rapport à l'axe de rotations tel quel :

$$I_z = \int a^2 dm = \int a^2 \rho(r) dV = \int\int\int_{(x^2+y^2+z^2)<R^2} (x^2+y^2)\ \rho_0\ e^{-r/R}\ dx\ dy\ dz \quad (19)$$

   où $a^2 = x^2 + y^2$ est la distance par rapport à l'axe d'inertie et $r = \sqrt{x^2+y^2+z^2}$ est la distance au centre de la sphère.
   Comme la valeur de l'expression à intégrer est la même si on échange $x \to -x$, $y \to -y$, ou $z \to -z$, on peut calculer l'intégrale pour la partie de la sphère où $x > 0$, $y > 0$, et $z > 0$, qui représente 1/8 du volume total (le résultat est le même que pour les autres éléments $x < 0$, $y > 0$, et $z > 0$; $x > 0$, $y < 0$, et $z > 0$; etc.)

$$I_z = 8 \times \int_0^R \int_0^R \int_0^R (x^2+y^2)\ \rho_0\ e^{-r/R}\ dx\ dy\ dz \quad (20)$$

$$\approx 8 \times \sum_{i=0}^{N}\sum_{j=0}^{N}\sum_{k=0}^{N} (x_i^2+y_j^2)\ \rho_0\ e^{-\sqrt{(x_i^2+y_j^2+z_k^2)}/R}\ (\Delta x)^3 \quad (21)$$

   avec la condition $(x^2+y^2+z^2) < R^2$. Le programme calcule le moment d'inertie pour ce huitième en discrétisant la sphère par des cubes de petit volume. L'utilisateur choisit le nombre de divisions N des axes (on a donc $\Delta x = \Delta y = \Delta z = \mathrm{R/N}$), et le volume de chaque petit cube est alors :

$$\Delta V = \left(\frac{R}{N}\right)^3 \quad (22)$$

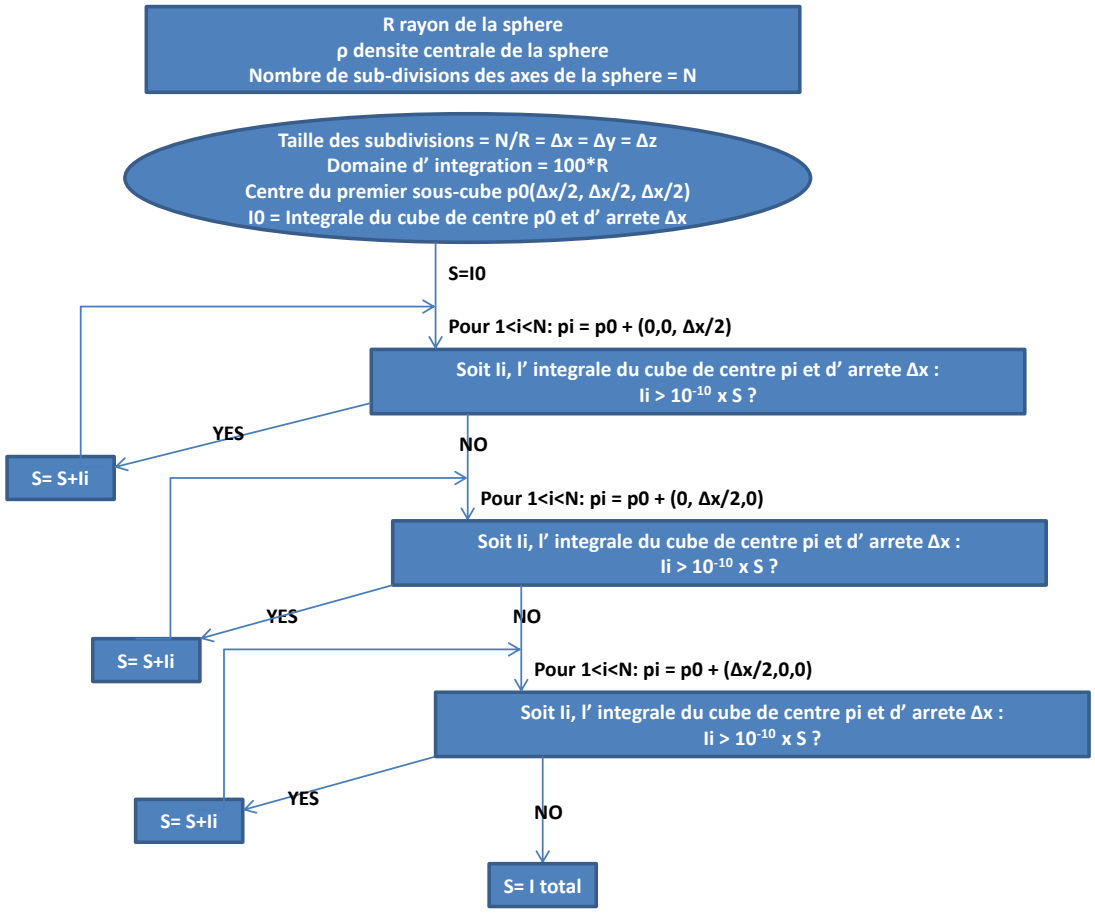

FIGURE 1 – Diagramme de flux pour le calcule du moment d'inertie.

La somme commence avec le cube de centre $(\Delta x/2, \Delta y/2, \Delta z/2)$, la densité du cube et la distance de son centre à l'axe de rotation sont calculées. A chaque changement de cube nous ajoutons R/N à une coordonée, en commmençant avec celle de l'axe z, jusqu'à la valeur limite R, ensuite la valeur de z revient à $\Delta x/2$ et nous ajoutons R/N à la coordonnée y, etc. Voir aussi le diagramme de flux montré dans la figure 1.

MomentdInerite.cpp

```
1  //moment d'inerite d'une sphere
2  #include <iostream>
3  #include <cmath>
4  #include <iomanip>
5
6  using namespace std;
7
8  //fonction densite
9  double densite(double r, double R, double rho0);
10 //distance du centre de la sphere
11 double r(const double point[]);
12 //distance a l'axe de rotation au carre
13 double perp2(const double point[]);
14
15 int main() {
16   //saisi des parametres
17   double rho0;
18   cout << "Quelle est la densite de la sphere au centre (en kg/m^3) ? ";
```

```
19    cin >> rho0;
20    double R;     //rayon de la sphre
21    cout << "et le rayon de la sphere (en m) ? ";
22    cin >> R;
23    int div; //divisions de l'axe x
24    cout << "En combien de partie voulez vous diviser le rayon de la sphere ?
          ";
25    cin >> div;
26
27    double dx = R/div;
28    double dV = pow(dx,3);    //sous-volume
29
30    double mInertie = 0.;
31    int iter = 0;    //# iterations
32    double point[3];    //point a l'interieur de la sphere
33    //on calcule le moment d'inertie d'un octant de la sphere
34    point[0] = dx/2.;
35    for (int i=0; i<div; i++) {
36      point[1] = dx/2.;
37      for (int j=0; j<div; j++) {
38        point[2] = dx/2.;
39        for (int k=0; k<div; k++) {
40          //d'abord il faut verifier si le point est a l'interieur
41          if (r(point)>R) break;
42        mInertie += densite(r(point),R,rho0) * dV * perp2(point) ;
43        point[2] += dx;
44        iter++;
45        }
46        point[1] += dx;
47      }
48      point[0] += dx;
49    }
50
51    cout << "Moment d'inertie de l'octant calcule : " << setw(10) << mInertie
52         << " kg m^2" << endl
53         << "et nombre d'iterations computees : " << iter << endl;
54
55    cout << "Le moment d'inertie est egal a "
56         << setw(15) << setprecision(10) << scientific << 8.*mInertie
57         << " kg m^2" << endl;
58
59    return 0;
60  }
61
62  //fonction densite
63  double densite(double r, double R, double rho0) {
64    return rho0*exp(-r/R);
65  }
66
67  //distance du centre de la sphere
68  double r(const double point[]) {
69    return sqrt(pow(point[0],2) + pow(point[1],2) + pow(point[2],2));
70  }
71
72  //distance a l'axe de rotation au carre
73  double perp2(const double point[]) {
74    return pow(point[0],2) + pow(point[1],2);
75  }
```

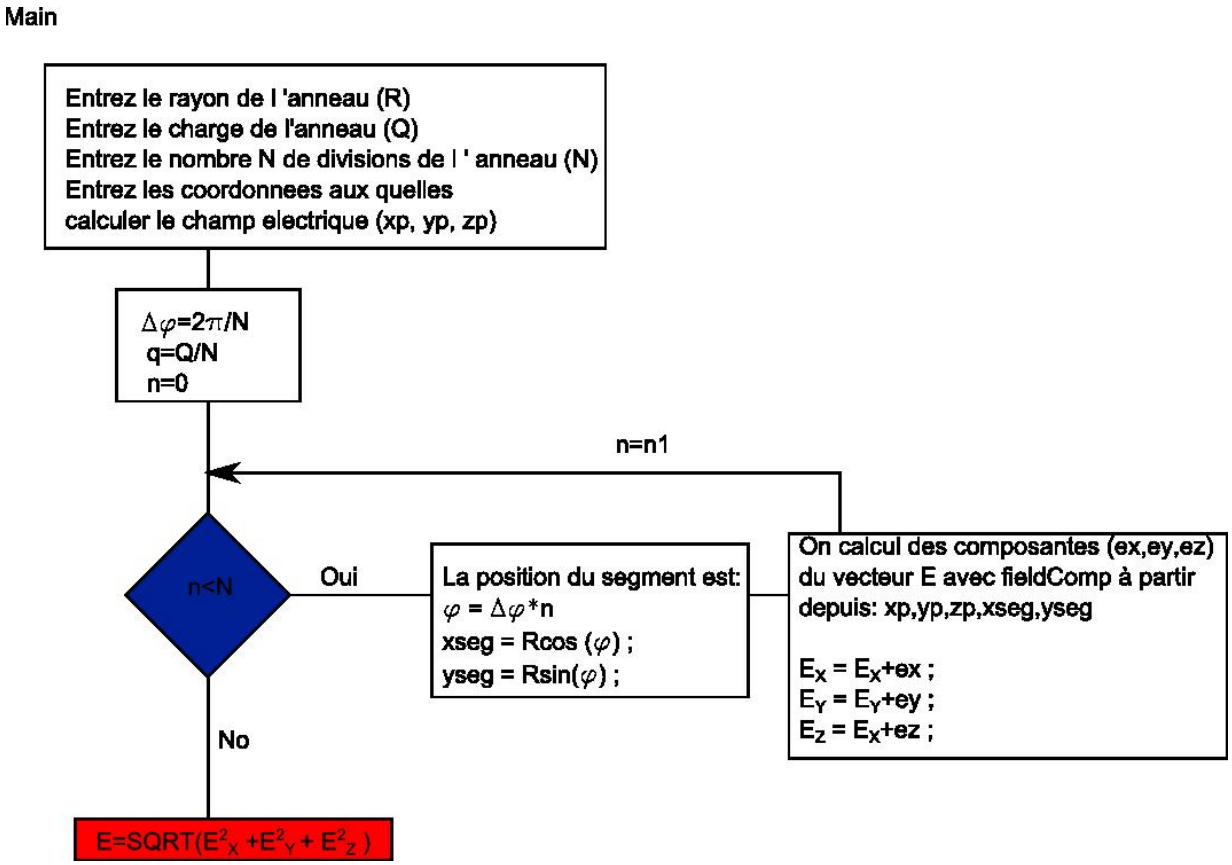

FIGURE 2 – Diagramme de flux pour le calcule du champ électrique.

9. **Champ électrique**

On cherche à calculer le champ électrique **E** produit par un anneau de rayon R et de charge Q en tout point quelconque X. Pour ce faire on considère un système de coordonnées cartesiennes dont l'origine est le centre de l'anneau, et l'on divise l'anneau en segments de longueur $I = R\Delta\Phi$ avec $\Delta\Phi = 2\pi/n$ (modélisation du problème). Le rayon de l'anneau, la charge et le nombre de divisions sont fournis par l'utilisateur.

En utilisant le principe de superposition, on calcule le champ généré au point X par chaque segment et l'on additionne ensuite les contributions des différents segments pour obtenir le champs **E** total. Voir aussi le diagramme de flux montré dans la figure 2.

ChampElect.cpp

```
1  //champ electrique dans un point quelconque
2  #include <iostream>
3  #include <cmath>
4
5  using namespace std;
6
7  void fieldComp(double x, double y, double z,
8                 double xp, double yp, double zp,
9                 double q, double &ex, double &ey, double &ez);
10 double distanceToPoint(double x, double y, double z,
11                         double xp, double yp, double zp);
12
13 //constante dielectrique
14 const double eps0=8.854e-12;
15
16 int main() {
17   double R;
```

```
18    cout << "Entrez le rayon de l'anneau (en m) : ";
19    cout << "R = ";    cin >> R;
20    double Q;
21    cout << "Entrez le charge de l'anneau (en C) : ";
22    cout << "Q = ";    cin >> Q;
23    int div;
24    cout << "Entrez le nombre N de divisions de l'anneau : ";
25    cin >> div;
26
27    double xp, yp, zp;
28    cout << "Entrez les coordonnees du point X (en m) : " << endl;
29    cout << "x = ";    cin >> xp;
30    cout << "y = ";    cin >> yp;
31    cout << "z = ";    cin >> zp;
32
33    /* On imagine l'anneau dans le plan xy, unidimensionnel
34      1) l'anneau est divise en N segments
35      2) on "boucle" sur les segments de l'anneau
36      3) on calcule le champ genere par chaque segment
37      4) on additionne les contributions au champ total. */
38    double deltaPhi = 2.*M_PI/div;
39    //charge du segment (q et Q sont 2 variables differentes !)
40    double q = Q/div;
41
42    double xseg, yseg, zseg;
43    double phi;
44    double distance=0.;
45    double ex, ey, ez;
46    double fieldX=0., fieldY=0., fieldZ=0.;
47    //boucle sur l'anneau
48    for (int iseg=0; iseg<div; iseg++) {
49      phi = deltaPhi/2. + iseg*deltaPhi;
50      xseg = R*cos(phi);
51      yseg = R*sin(phi);
52      zseg = 0.;
53
54      //calcul des composantes du vecteur E
55      fieldComp(xseg,yseg,zseg,xp,yp,zp,q,ex,ey,ez);
56
57      //et on addition le champ genere par chaque segment
58      fieldX += ex;
59      fieldY += ey;
60      fieldZ += ez;
61    }
62
63    double fieldXYZ = sqrt(pow(fieldX,2)+pow(fieldY,2)+pow(fieldZ,2));
64    cout << "Estimation du champ: |E| = " << fieldXYZ << " V/m" << endl;
65    cout << " Ex = " << fieldX << " V/m" << endl;
66    cout << " Ey = " << fieldY << " V/m" << endl;
67    cout << " Ez = " << fieldZ << " V/m" << endl;
68
69    return 0;
70  }
71
72  //calcul du champ electrique en X
73  void fieldComp(double x, double y, double z,
74                 double xp, double yp, double zp,
75                 double q, double &ex, double &ey, double &ez) {
76
77    //calcul de la distance entre le centre du segment et le point X
```

```
78    double dist = distanceToPoint(x,y,z,xp,yp,zp);
79
80    //champ electrique |E| en X
81    double e = 1./(4.*M_PI*eps0) * q / (dist*dist);
82
83    //composantes de E
84    ex = e * (xp-x) / dist;
85    ey = e * (yp-y) / dist;
86    ez = e * (zp-z) / dist;
87  }
88
89  //distance entre deux points
90  double distanceToPoint(double x1, double y1, double z1,
91                         double x2, double y2, double z2) {
92    double distance;
93    double diffx = x2-x1;
94    double diffy = y2-y1;
95    double diffz = z2-z1;
96    distance = sqrt(pow(diffx,2) + pow(diffy,2) + pow(diffz,2));
97
98    return distance;
99  }
```

10. **Champ magnétique**
On cherche à calculer le champ magnétique $\mathbf{B}$ produit par un anneau de rayon R parcouru par un courant I en tout point quelconque X. Il s'agit ici d'un problème très similaire au précedent. Sa résolution est équivalente : définition d'un système de coordonnées, division de l'anneau en segments, calcul de la contribution de chaque segment, somme des contributions. La difficulté supplémentaire ici réside dans le calcul du produit vectoriel $\mathrm{d}\mathbf{l} \times \mathbf{r}$ (loi de Biot-Savart) :

$$\mathrm{d}\mathbf{B} = \frac{\mu_0}{4\pi} I \frac{\mathrm{d}\mathbf{l} \times \mathbf{r}}{r^3} \tag{23}$$

Le segment $\mathrm{d}\mathbf{l}$ est orthogonal au rayon entre le centre du cercle et le centre du segment et sa longueur est donnée par $\Delta\phi \cdot R$:

$$\mathrm{d}\mathbf{l} = (R \cdot (\cos(\phi + \Delta\phi) - \cos(\phi)), R \cdot (\sin(\phi + \Delta\phi) - \sin(\phi)),\ 0) \quad . \tag{24}$$

$\mathbf{r}$ est le vecteur entre le centre du segment et le point X, comme dans le problème précédent. La seule difficult consiste en calculer correctement le produit vectoriel $\mathrm{d}\mathbf{l} \times \mathbf{r}$.

BiotSavart.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   using namespace std;
5
6   //declaration + initialisation des constantes globales
7   const double mu0 = 4.*M_PI*10e-7;
8
9   int main() {
10    //declaration des donnees du probleme
11    double R = 1;         //rayon de l'anneau
12    double I = 1000.;     //intensite du courant
13
14    //sesie du nombre de divisions de l'anneau
15    double N = 1000;      //nombre de suosdivisions de l'anneau
16    cout << "Entrez le nombre de division de l'anneau: " << endl;
```

```
17    cout << "N = ";    cin >> N;
18
19    //sesie du cuorant
20    cout << "Entrez le courant en A: " << endl;
21    cout << "I = ";    cin >> I;
22
23    //sesie de la position aquelle calculer le champ
24    double px, py, pz;  //coordonnees du point choisi
25    cout << "Entrez la position ou calculer le champ magnetique : " << endl;
26    cout << "x = ";    cin >> px;
27    cout << "y = ";    cin >> py;
28    cout << "z = ";    cin >> pz;
29
30    //on verifie que le point ne se situe pas sur l'anneau
31    //(pas vraiment necessaire)
32    if (pz == 0) {
33      if ((px*px + py*py) == R*R) {
34        cout << "Le point est sur l'anneau, entrez d'autres valeurs : "
35             << endl;
36        cout << "x = ";    cin >> px;
37        cout << "y = ";    cin >> py;
38        cout << "z = ";    cin >> pz;
39
40        while ((px*px + py*py) == R*R) {
41          cout << "Le point est toujours sur l'anneau, "
42               << "entrez d'autres valeurs : " << endl;
43          cout << "x = ";    cin >> px;
44          cout << "y = ";    cin >> py;
45          cout << "z = ";    cin >> pz;
46        }
47      }
48    }
49
50    //le champ total est calcule avec une boucle sur les segments de l'anneau
51    //on initialise les composantes a zero en-dehors de la boucle principale
52    double dBx, dBy, dBz; //champ genere par le segment
53    double Bx, By, Bz;    //champ total
54    Bx =0.; By = 0.; Bz = 0.;
55
56    double rx, ry, rz;  //vecteur du segment au point choisi
57    double norme;       //norme du vecteur
58
59    double theta;         //coordonnee polaire du segment de l'anneau
60    double dtheta;
61    dtheta = 2.*M_PI/N;   //incrementation de l'angle
62    double dlx, dly, dlz; //composante du segment
63
64    //boucle sur les segments de l'anneau
65    for (int i=0; i<N; i++) {
66      //angle polaire du segment (commence a zero)
67      theta = i*dtheta;
68
69      //calcul des composantes du segment
70      dlx = R*(cos(theta+dtheta) - cos(theta));
71      dly = R*(sin(theta+dtheta) - sin(theta));
72      dlz = 0.;
73
74      //calcul des composantes du vecteur r reliant le segment et le point
75      rx = px - R*cos(theta);
76      ry = py - R*sin(theta);
```

```
77      rz = pz;
78
79      //norme du vecteur
80      norme = sqrt(rx*rx + ry*ry + rz*rz);
81
82      //calcul des composantes du champ magnetique genere par le segment
83      //les composantes sont donnees par le produit vectoriel dl x r
84      dBx = (mu0/(4.*M_PI))*I*(dly*rz-dlz*ry)/pow(norme,3);
85      dBy = (mu0/(4.*M_PI))*I*(dlz*rx-dlx*rz)/pow(norme,3);
86      dBz = (mu0/(4.*M_PI))*I*(dlx*ry-dly*rx)/pow(norme,3);
87
88      //pour chaque iteration on addition le champ genere per le segment
89      //au champ magnetique total
90      Bx += dBx;
91      By += dBy;
92      Bz += dBz;
93    }
94
95    cout << "Le champ B[T] au point choisi vaut :" << endl;
96    cout << "B = " << sqrt(Bx*Bx + By*By + Bz*Bz) << endl;
97    cout << "Bx = " << Bx << endl;
98    cout << "By = " << By << endl;
99    cout << "Bz = " << Bz << endl;
100
101   return 0;
102 }
```

5.2 Problèmes de différentiation numérique

1. **Calcul de dérivées numériques**
   Le programme suivant calcule la dérivée à gauche, à droite et centrale pour les fonctions sin(x) et cos(x).
   DiffSinus.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  int main() {
7    //saisi des parametres
8    double x;
9    cout << "Entrez l'anlge (en radians) : ";
10   cin >> x;
11   double h;
12   cout << "Entrez h = ";
13   cin >> h;
14
15   cout << "\nsin(x) = " << sin(x) << endl;
16   cout << "derivee dsin(x)/dx = " << cos(x) << endl;
17   cout << "difference finie a droite : "
18        << (sin(x+h)-sin(x))/h << endl;
19   cout << "difference finie a gauche : "
20        << (sin(x)-sin(x-h))/h << endl;
21   cout << "difference finie centrale : "
22        << (sin(x+h)-sin(x-h))/(2.*h) << endl;
23
24   cout << "\ncos(x) = " << cos(x) << endl;
25   cout << "derivee dcos(x)/dx = " << -sin(x) << endl;
```

```
26    cout << "difference finie a droite : "
27         << (cos(x+h)-cos(x))/h << endl;
28    cout << "difference finie a gauche : "
29         << (cos(x)-cos(x-h))/h << endl;
30    cout << "difference finie centrale : "
31         << (cos(x+h)-cos(x-h))/(2.*h) << endl;
32
33    return 0;
34  }
```

2. **Dérivée**
   Le programme calcule les dérivées première et seconde d'un polynôme arbitraire du second degré. Le programme demande à l'utilisateur d'entrer le point x auquel calculer les dérivées et la largeur de l'incrément h. Dans le calcul numérique de la dérivée seconde on retrouve h^2 au dénominateur. Il faut aussi faire attention à la précision finie de l'ordinateur dans le choix de h.

   Prob2_diff.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   using namespace std;
5
6   //prototype de la fonction
7   double fonc(double);
8
9   int main() {
10    double x;
11    cout << "Entrez une valeur a laquelle calculer les derivees :" << endl;
12    cout << "x = ";    cin >> x;
13    double h;
14    cout << "Entrez une valeur de h :" << endl;
15    cout << "h = ";    cin >> h;
16
17    double derivee1, derivee2;
18    derivee1 = (fonc(x+h) - fonc(x-h)) / (2.*h);
19    derivee2 = (fonc(x+h) - 2.*fonc(x) + fonc(x-h)) / (h*h);
20
21    cout << "Au premier ordre en h la derivee premiere vaut :" << endl;
22    cout << "f'(x) = " << derivee1 << endl;
23    cout << "La derivee seconde vaut (attention a h !) : " << endl;
24    cout << "f''(x) = " << derivee2 << endl;
25
26    return 0;
27  }
28
29  //definition de la fonction
30  //dans cet example nous avons choisi un polynome de 2eme degree;
31  double fonc(double x) {
32    double a, b, c;
33    a = 10.;
34    b = -3.;
35    c = 7.;
36
37    double y = a*(x*x) + b*x + c;
38
39    return y;
40  }
```

3. **Activité d'une source de cobalt**
L'activité d'une source radioactive de cobalt 60 ($^{60}_{27}$Co) est donnée par la loi de Poisson :

$$A = -\frac{dN}{dt} = -N_0\, e^{-t/\tau}\, \frac{-1}{\tau} = \frac{N(t)}{\tau} \tag{25}$$

avec $\tau = t_{1/2}/ln(2)$ ($t_{1/2}$ est le temps de demi vie). Le nombre initial d'atomes de cobalt est $N_0 = M_0/A_0 \times N_A$ ou M_0 est la masse initiale de l'échantillon, A_0 est la masse du Cobalt en Kg/mole et N_A est le nombre d'Avogadro, c'est-à-dire le nombre d'atomes dans une mole. Dans le programme nous avons résolu l'équation différentielle de manière numérique, et nous faisons la comparaison avec le résultat exact et la differentiation numérique.
La figure 3 montre le nombre d'atomes (en moles) en fonction du temps pour une masse initiale de 500 Kg sur 50 années (clairement une fonction exponentielle).
Cobalt60.cpp

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

int main() {
  const double nAvogadro = 6.022141e23;
  const double secParAn = 365.*24.*3600.;
  double tau = 5.271/log(2.);      //temps de demi-vie du Co-60 (5.271 ans)

  cout << "*************************************************************\n";
  cout << "Programme pour le calcul de l'activite d'une source de Co60\n";
  const double A0=59.9338222e-3;
  cout << "La masse du Cobalt est A0 = " << A0 << " kg per mole.\n";
  cout << "*************************************************************\n";

  //saisi des parametres
  double M0;
  cout << "Quelle est la masse initiale de la source (en kg) ? ";
  cin >> M0;
  double t0;
  cout << "Apres combien d'ans voulez vous connaitre l'activite de la
      source ? ";
  cin >> t0;
  int n;
  cout << "Combien d'iterations voulez-vous faire pour ce calcul ? ";
  cin >> n;

  double S = M0/A0;    //nombre initial d'atomes / nAvogadro
  double dt = t0/n;
  cout << "valeur exacte " << exp(-t0/tau)/tau*M0/A0*nAvogadro << endl;
  cout << "valeur approx "
       << (exp((-t0+dt)/tau)-exp((-t0-dt)/tau))/(2.*dt)*M0/A0*nAvogadro
       << " (calcul incorrect !)" << endl;
  //faux parce que si on connait exp(-t0+dt)
  //on connait aussi la valeur exacte !
  for(int i=0; i< n; i++)
    S = S*(1.0-dt/tau);

  double activite = S/tau*nAvogadro;
  cout << "Fraction remanente "<< S/M0*A0<<endl;
  cout << "L'activite de la source apres " << t0 << " ans est de "
       << activite << " desintegrations par an" << endl;
```

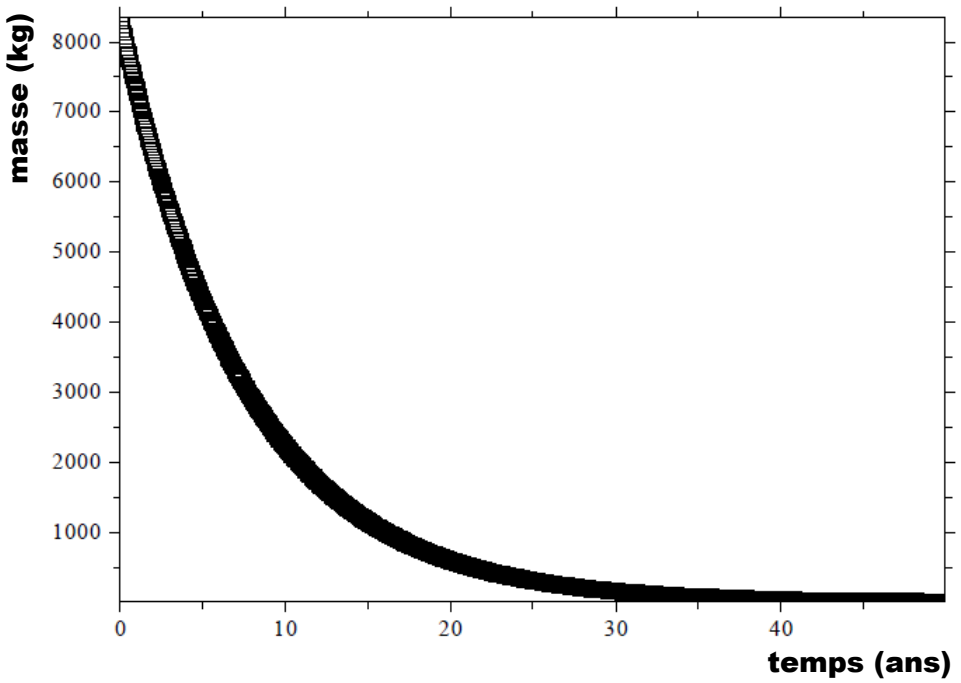

FIGURE 3 – Graphique du nombre de moles d'atomes du Cobalt restant en fonction du temps pour une masse initiale de 8500 Kg.

```
44    cout << "equivalent a " << setprecision(10) << activite/secParAn
45         << " Bq (desintegrations par seconde)." << endl;
46
47    return 0;
48  }
```

4. **Erreur d'arrondi**
   Le programme calcule la derivée de la fonction $f(x) = x^2$ avec la formule de la différence finie centrale :

$$\begin{aligned} f'(x)_{exacte} &= 2x \\ f'_{approx}(x) &= \frac{f(x+h) - f(x-h)}{2h} = \frac{(x+h)^2 - (x-h)^2}{2h} = \frac{4xh}{2h} = 2x \end{aligned}$$

   Dans ce cas, l'approximation est exacte. L'erreur sera purement une erreur d'arrondi, c'est-à-dire une erreur de précision dans le calcul de l'approximation : dans chaque approximation l'approximation l'ordinateur peut rater la valeur exacte. Dans le cas où $h = x$, ou aussi pour $h = 1$ et x entier, l'ordinateur peut trouver la valeur exacte parce qu'il exécutera d'abord la somme $x + h$ et le reste $x - h$. Dans la figure 4 nous voyons que pour $x = 1$ l'approximation est moins bonne avec h plus petit. Mais pour $x = 0.001$, l'erreur d'arrondi diminue jusqu'à $h = x$ et augmente ensuite. Le résultat trouvé dépend aussi de l'ordinateur utilisé.

   DiffFinieCentrale.cpp

```
1  #include <iostream>
2  #include <cmath>
3  #include <iomanip>
4
5  using namespace std;
```

```
6
7  double diffFinieCentral(double x0, double h);
8
9  int main() {
10   const int STEPMAX = 21;
11
12   double h, dfNum, dfCalc;
13   double delta;
14   double x0;
15   cout << "Estimation de la derivee de f(x)=x^3 avec la differece finie
         centrale" << endl;
16   cout << "entrez la valeur de x ou estimer f'(x) : ";
17   cin >> x0;
18
19   cout << endl;
20   cout << "          h          ;     log(dfNum - dfAna)   ;          dfNum"
21        << endl << endl;
22   for (int step=0; step<STEPMAX; step++) {
23     h = pow(10.,-step);
24
25     dfNum = diffFinieCentral(x0,h);    //derivee numerique
26     dfCalc = 3.*x0;    //derivee analytique
27
28     if (fabs(dfNum-dfCalc)!=0)
29       delta = log10(fabs(dfNum-dfCalc));
30     else
31       delta = -9999.99;    //une valeur jamais atteint par le logarithme
32
33     cout << setprecision(20) << fixed << h <<  " "
34          << delta << " " << dfNum <<endl;
35   }
36   cout << endl;
37
38   return 0;
39 }
40
41 double diffFinieCentral(double x0, double h) {
42   double df = (pow(x0+h,3)-pow(x0-h,3)) / (2.*h);
43   return df;
44 }
```

Dans le programme nous avons défini une fonction pour le calcul de la dérivée numérique, et une variable qui prend la valeur exacte de la derivée. Ensuite nous évaluons la différence entre l'approximation et la derivée exacte. Pour éviter une erreur d'exéxecution quand le programme essaie de prendre le logarithme de 0, nous avons mis une condition : si la valeur trouvée par l'approximation est égale à la valeur exacte calculée, on utilise une valeur qui n'est jamais atteinte, que nous utilisons seulement pour faire le graphique. Dans la figure 5 on montre le même calcul pour $f(x) = x^3$, avec :

$$\begin{aligned} f'(x)_{exacte} &= 3x^2 \\ f'_{approx}(x) &= \frac{f(x+h)-f(x-h)}{2h} = \frac{(x+h)^3-(x-h)^3}{2h} = \frac{2h(3x^2+h^2)}{2h} = 3x^2+h^2 \end{aligned}$$

mais on voit que la valeur -30 n'est jamais atteinte, parce qu'on n'a jamais $h = 0$, et pour h trop petit les erreurs d'arrondi dans les opérations de puissance et de reste dans le calcul approché sont grandes.

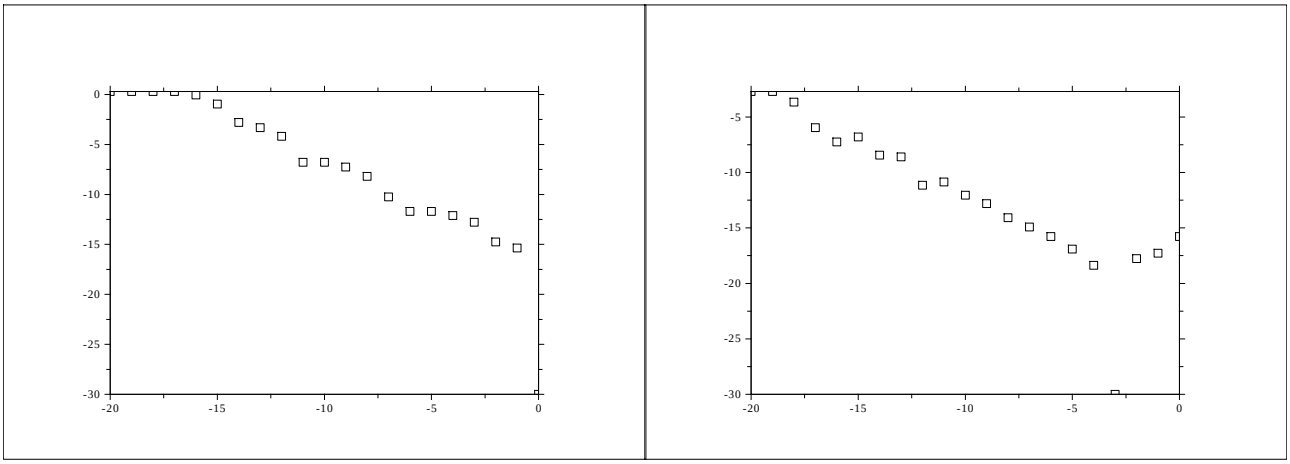

FIGURE 4 – Graphique de l'erreur (en puissance de 10) trouvée lors du calcul de l'approximation de la derivée de $f(x) = x^2$, en fonction du logarithme du paramètre h utilisé. A gauche on montre le graphique pour $x = 1$, à droite pour $x = 0.001$.

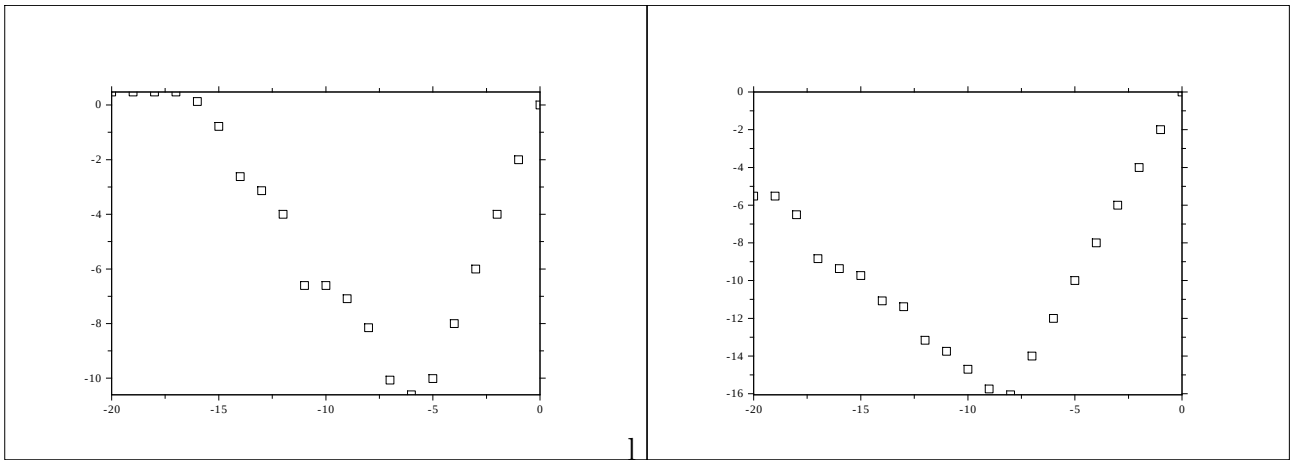

FIGURE 5 – Graphique de l'erreur (en puissances de 10) trouvée lors du calcul de l'approximation de la derivée de $f(x) = x^3$, en fonction du logarithme du paramètre h utilisé. A gauche on montre le graphique pour $x = 1$, à droite pour $x = 0.001$.