[../icons/ujf.gif] [../icons/ujf.gif]

La Méthode de Runge-Kutta

Licence de Physique/M2 ITDD

Table des matières

1 Introduction

La méthode de Runge-Kutta est une méthode permettant l'intégration numérique d'une équation différentielle du premier ordre. La version donnée ici est une **approximation à l'ordre 4 de la dérivée** (il existe des méthodes de Runge-Kutta pour différents ordres, mais le meilleur compromis précision/vitesse est l'ordre 4). Nous donnons ici une *méthode dites à pas variable* : on intègre les équations sur Δt en un pas, puis sur Δt en 2 pas (i.e., en passant par $\frac{\Delta t}{2}$) ; si l'écart entre le 2 résultats est supérieur à une précision donnée, Δt est divisé par 2, sinon, il est multiplié par 2 pour le pas suivant. Cela permet de faire des grands pas de temps quand le système évolue lentement et de faire des pas de temps petits si ces variations sont rapides.

L'algorithme donné ici est une adaptation de celui fourni dans la bibliothèque **Numerical Receipes** pour le C/C++. Nous donnons "2 versions" : la première (C++ uniquement) est à utiliser si vous avez fait une classe pour représenter l'objet que vous voulez faire évoluer. La seconde est utilisable en C standard comme en C++ correspond à une programmation non-objet. Dans les 2 cas, le code peut vous paraître un peu long mais vous n'avez pas grand chose à faire.

2 Runge-Kutta en C++

Supposons que vous ayez écrit une classe **MonSystem** décrivant le système que vous voulez faire évoluer.

2.1 La classe DynamicalSystem

Télécharger les fichiers DynamicalSystem.h et DynamicalSystem.C. Nous donnons le prototype ci-dessous (contenu du fichier **DynamicalSystem.h)**.

```
#ifndef _DYNAMICALSYSTEM_
#define _DYNAMICALSYSTEM_

class DynamicalSystem
{
 protected:
        int NVariable;
        double Precision;
        double Hestimate;
```

```
        double Hmin;
public:
        DynamicalSystem(int n);
        virtual void EDP(double t,double *X,double *dX){}
        void RungeKutta(double *y, double t1,double t2);
        void SetPrecision(double p){Precision=p;}
        void SetHmin(double h){Hmin=h;}
        void SetHestimate(double h){Hestimate=h;}

private:
        double *vector(int N);
        void nrerror(char * error_text);
        void free_vector(double *v);
        int odeint(double *ystart, double x1, double x2, double eps,
                   double h1, double hmin, int *nok, int *nbad);
        void rk4(double *y, double *dydx, double x, double h, double *yout);
        void rkqc(double *y, double *dydx, double *x, double htry,
                  double eps, double *yscal, double *hdid, double *hnext);
};
#endif
```

Dans cette classe, le **constructeur est appelé avec le nombre de variables** (i.e., le nombre d'équations différentielles à intégrer). La variable *Precision* sert à changer le pas au cours de l'intégration. *Hestimate* et *Hmin* sont respectivement les pas supposé et minimum utilisés pour l'intégration. En principe, leurs valeurs par défaut permettent de résoudre la plupart des problèmes. La méthode **RungeKutta** est la méthode que vous appellerez dans votre classe MonSystem ; elle prend comme arguments un pointeur sur des doubles (c'est votre tableau de conditions initiales), le temps initial et le temps final. Les valeurs à l'instant final remplacerons les conditions initiales dans le tableau.

2.2 Utilisation de cette classe

Votre classe **MonSystem** devra être écrite de cette façon :

```
#include "DynamicalSystem.h"
class MonSystem : public DynamicalSystem
{
   ...
public:
   MonSystem(...);
   void EDP(double t, double *X, double *dX);
   ...
};
MonSystem::MonSystem(...) : DynamicalSystem(4)
{
}
...
```

Explications

- La première ligne signifie que **MonSystem** hérite de la classe **DynamicalSystem**, c'est-à-dire de ces attributs et méthodes protégés et publiques.
- La méthode *EDP(double t, double *X, double *dX)* est la méthode contenant vos équations du mouvement (avec t le temps auquel est calculé $\frac{d\vec{X}}{dt}$ qui est représenté par *dX[0]...dX[3]* (ici nous avons 4 variables) et *X[0]...X[3]* représentant les valeurs $\vec{X}(t)$). Cette méthode "écrase" la méthode **DynamicalSystem : :EDP** car elle était *virtuelle*.

— Le constructeur de **MonSystem** appelle d'abord celui de **DynamicalSystem** avec le nombre de variables (ici 4).
— Enfin, il faudra écrire une méthode de **MonSystem** qui appelle **DynamicalSystem : :RungeKutta(...)** pour faire l'intégration (à la place de votre méthode faisant la méthode d'Euler).

2.3 Compilation

La compilation de votre programme se fera en 2 temps. Le premier consiste à **compiler DynamicalSystem.C** sans faire d'édition de lien ; par exemple avec CC, vous taperez

```
CC -c DynamicalSystem.C
```

Ceci est fait une fois pour toute. Ensuite, si votre programme s'appelle *MonPrg.C*, vous taperez

```
CC -o Monprg MonPrg.C DynamicalSystem.o
```

à chaque fois que vous aurez modifier *MonPrg.C*.

3 Runge-Kutta : fonction C utilisable en C++

Télécharger le fichier rk4.C

3.1 Utilisation

Vous **devrez inclure ce fichier** à votre programme (`#include "RK4.C"`). Ensuite, au lieu d'appeler la méthode d'Euler, vous appellerez la fonction **RungeKutta** ; son premier argument est un pointeur sur votre tableau de conditions initiales à $t = t_1$ (*y[0]...y[N-1]*). Ce tableau est de dimension N (argument 2). En sortie, ce tableau contient les valeurs à $t = t_2$. Les 2 arguments t_1 et t_2 sont le temps initial et final pour l'intégration. Le dernier argument est le nom de votre fonction contenant les équations du mouvement. **Si cette fonction s'appelle EDP, son prototype doit être**

```
void EDP(double t, double *X, double *dX);
```

où t est le temps auquel sont données les valeurs $\vec{X}$ (X[0]....) et dX[0]... représentent $\frac{d\vec{X}}{dt}$.