TP5 : La programmation orientée objet (3)

Objectifs et conseils

Qu'est ce que la POO ?

Qu'est ce qu'un objet ? Il s'agit d'un mélange de plusieurs variables et fonctions. Imaginez que vous avez créé un programme qui permet de résoudre des équations différentielles ordinaires : vous pouvez afficher vos solutions, calculer des ordres de convergence, comparer deux méthodes ... Le code est complexe : il aura besoin de plusieurs fonctions qui s'appellent entre elles, ainsi que de variables pour mémoriser la solution au cours du temps, la méthode utilisée, le pas de temps choisi ... Au final, votre code est composé de plusieurs fonctions et variables. Votre code sera difficilement accessible par quelqu'un qui n'est pas un expert du sujet : Quelle fonction il faut appeler en premier ? Quelles valeurs doit-on envoyer à quelle fonction pour afficher la solution ? etc ... Votre solution est de concevoir votre code de *manière orientée objet*. Ce qui signifie que vous placerez tout votre code dans une grande boîte. Cette boîte c'est ce qu'on appelle un **objet**. L'objet contient toutes les fonctions et variables mais elles sont masquées pour l'utilisateur. Seulement quelques outils sont proposés à l'utilisateur comme par exemple : définir mon pas de temps, mon intervalle de calcul et ma méthode, calculer l'ordre de la méthode, afficher la solution ...

En quelques lignes :

- La programmation orientée objet est une façon de concevoir son code dans laquelle on manipule des objets.
- Les objets peuvent être complexes mais leur utilisation est simplifiée. C'est un des avantages de la programmation orientée objet.
- Un objet est constitué d'attributs et de méthodes, c'est-à-dire de variables et de fonctions membres.
- On appelle les méthodes de ces objets pour les modifier ou obtenir des informations.

Qu'est ce qu'une classe ? Pour créer un objet, il faut d'abord créer une classe. Créer une classe consiste à définir les plans de l'objet. Une fois que la classe est faite (le plan), il est possible de créer autant d'objets du même type. Vocabulaire : on dit qu'un objet est une **instance** d'une classe.

Objectifs

- Surcharge de fonctions (connu aussi comme du polymorphisme statique)
- Héritage
- Polymorphisme (dynamique)

À travers l'implémentation d'une classe de solveurs d'équations différentielles ordinaires.

Héritage et Polymorphisme sur un exemple simple

Télécharger le code en suivant ce lien. L'objectif est de vous montrer l'évolution d'un code avec utilisation de la POO sur un exemple simple. Nous avons 4 étudiants qui font partie du même groupe de travail et nous souhaitons calculer la moyenne de chaque étudiant en sachant que même si toutes les semaines ils sont censés rendre le même devoir, certains perdent des points : soit parce qu'ils arrivent toujours en retard (-1 sur la note), soit parce qu'ils oublient de rendre le devoir (0 pour la note). Le dossier contient 5 versions du code permettant à chaque fois d'aller un cran plus loin.

Version 1 : On a une classe *Student* qui a pour variables privées le nom le prénom, le nombre de notes et la moyenne de l'étudiant(e). À chaque fois que l'on ajoute une nouvelle note on recalcule la moyenne de l'étudiant(e) (chaque note a le même coefficient).
Cette version est loin d'être complète. En effet tous les étudiants ont la même note alors que Jules et Alice sont toujours en retard (et donc devraient avoir un point de moins) et Louise oublie une fois sur quatre de rendre son devoir.
De plus, les noms de famille de Louise et Jules ne sont pas connus, ce qui oblige l'initialisation de leur nom de famille par un string vide :

```
students[1]->Initialize("Jules", ""); // Son prénom seulement est connu
```

Version 2 : La version 2 permet d'éviter l'utilisation du string vide dans la fonction *main* et vous montre une première utilisation de la **surcharge de fonctions**. Dans la classe *Student* deux fonctions *Initialize* cohabitent à présent :

```
// Si le nom et le prénom sont connus
void Initialize(std::string first_name, std::string last_name);
// Si seulement le prénom est connu
void Initialize(std::string first_name);
```

Dans la fonction *main* on appelle l'une ou l'autre (en fonction de l'étudiant(e)) et le compilateur arrive à reconnaitre quelle fonction il doit prendre (en fonction des attributs donnés). L'idée de la **surcharge de fonctions** est de créer du code qui fonctionne de différentes manières selon qui l'utilise. Le choix de la fonction à appeler se fait lors de la compilation. Il n'y a pas de surcoût, on parle de **polymorphisme statique**.

Version 3 : La version 3 est complète au niveau du code. La fonction *main* a été adaptée pour pouvoir prendre en compte les particularités de chaque étudiant(e) : Jules et Alice perdent un point sur chaque note (donc un point sur leur moyenne) et Louise a 0 une fois sur quatre (quand elle ne rend pas son devoir). Cependant la boucle sur les étudiants est devenue plus compliquée :

```
if (i == 0) // Étudiant(e)s sérieux(ses)
  students[i]->AddMark(marks[j]);
else if ((i == 1) || (i == 3)) // Étudiant(e)s en retard
  students[i]->AddMark(marks[j]-1);
else if (i == 2) // Étudiant(e)s oubliant de rendre sa copie 1 fois sur 4
{
  if ( (j % 4) != 0)
    students[i]->AddMark(marks[j]);
  else
    students[i]->AddMark(0.);
}
```

Cela devient vite impossible à gérer avec plus d'étudiants.

Version 4 : La version 4 vous permet de découvrir l'**héritage**, une technique qui permet de créer une classe à partir d'une autre classe qui lui sert de modèle. Cela permet d'éviter d'avoir à réécrire un même code source plusieurs fois mais en même temps de pouvoir avoir des comportements différents pour différents objets de la classe modèle dite classe **mère**. Ici deux classes **fille** *SeriousStudent* et *LateStudent* de la classe mère *Student* ont été créés (la classe fille *DazedStudent* est créée dans la version 5 donc le cas de Louise est toujours traité à part dans la fonction *main*).

Tout d'abord, le fichier *Student.h* a été modifié de la façon suivante :

- Les variables privées *_marks_number* et *_average* sont devenues *protégées* pour être accessibles aux classes *filles*.
- Un destructeur a été créé. Les destructeurs sont appelés quand vous faites un *delete* ou lors de la fin d'existence d'une variable (accolade fermante). Vous êtes obligés d'implémenter vous même un destructeur dans ces deux cas :
  - si vous avez une variable privée de votre classe qui est un pointeur dont l'allocation est gérée dans la classe. Il faut que la variable pointeur soit absolument supprimée dans le destructeur créé.
  - Et si jamais vous avez des fonctions virtuelles (car le destructeur doit dans ce cas être virtuel)...
- ... ce qui est le cas ici pour la fonction *AddMark* qui est virtuelle pure :

```
// "virtual" = fonction virtuelle (pouvant être définie dans la classe fille)
// "= 0" = fonction virtuelle pure (DOIT être définie dans la classe fille :
// elle ne l'est pas dans la classe mère)
virtual void AddMark(double mark) =0;
```

- Les deux classes filles qui dérivent de la classe mère *Student* ont été ajoutées. Le prototype de la fonction *AddMark* est dans chacune des classes puisqu'elle est virtuelle pure.

Ensuite concernant le fichier *Student.cpp* :

- La définition de la fonction *AddMark* de la classe mère a été retirée et celles des deux classes filles ont été ajoutées. La définition de la fonction diffère entre les deux classes filles ...

Puis finalement dans le fichier *main.cc* :

- Chaque étudiant est maintenant un pointeur de la classe fille à laquelle il appartient :

```
students[0] = new SeriousStudent(); // Premier(e) étudiant(e)
students[1] = new LateStudent(); // Second(e) étudiant(e)
students[2] = new SeriousStudent(); // Troisième étudiant(e)
students[3] = new LateStudent(); // Quatrième étudiant(e)
```

  Pour le moment Louise est une étudiante sérieuse puisque la classe *DazedStudent* n'a pas encore été créée. Son cas est encore géré dans la boucle comme précédemment.

Ici on utilise le **polymorphisme dynamique** (ou simplement appelé polymorphisme), le type de students[0] peut changer au cours de l'exécution. Le choix de la fonction *AddMark* à appeler ne se fait pas lors de la compilation comme dans le cas du **polymorphisme statique**, mais lors de l'exécution. Il y a donc un surcoût d'appel de la fonction virtuelle *AddMark* par rapport à une fonction classique.

Version 5 : La version 5 contient la troisième classe fille *DazedStudent*. La fonction *main* a donc été simplifiée. Cette classe est un peu plus compliquée :

- Elle contient une variable privée *_dazed* : l'étudiant(e) oublie de rendre son devoir toutes les *_dazed* fois.
- Les fonctions *Initialize* sont devenues virtuelles (mais pas virtuelles pures !) dans la classe *mère*

```
// "virtual" = fonction virtuelle
// (pouvant être définie dans la classe fille ou pas)
virtual void Initialize(std::string first_name, std::string last_name);
virtual void Initialize(std::string first_name);
```

  afin d'initialiser dans le cas d'un(e) étudiant(e) étourdi(e) la variable privée *_dazed* :

```
void DazedStudent::Initialize(std::string first_name)
{
  // On fait appel au constructeur de la classe mère
  Student::Initialize(first_name);
  // L'étudiant(e) oublie toutes les 4 fois de rendre son devoir
  _dazed = 4;
}
```

  La version plus classique de cette fonction continue d'exister dans la classe mère et est utilisée par les classes filles qui ne l'ont pas définies (*SeriousStudent* et *LateStudent*).
- La fonction *ForgotOrNot* est une fonction spécifique de cette classe fille (le prototype de cette fonction n'apparait pas dans la classe mère !). Elle permet de savoir si c'est une note pour laquelle l'étudiant(e) a oublié de rendre son devoir. Elle est accessible seulement dans la classe fille *DazedStudent* ou seulement pour des objets de cette classe fille.

Ce code est très simple et ne sert à rien à part illustrer sur un exemple simple comment fonctionne l'héritage et le polymorphisme. Ne pas hésiter à jouer avec pour bien se l'approprier, par exemple en ajoutant d'autres *catégories* d'étudiants ...

Nous allons à présent utiliser l'héritage et le polymorphisme dans le cadre du calcul scientifique !

Exercices

Télécharger la version initiale du code en suivant ce lien. Nous allons continuer de développer ce code afin de pouvoir résoudre plusieurs **systèmes** d'équations différentielles ordinaires s'écrivant sous la forme :

$$X' = f(t, X).$$

Nous souhaitons pouvoir permettre à l'utilisateur de choisir entre 4 schémas en temps :

• Euler Explicite • Runge Kutta 3 et 4 • Adams Bashforth 3.

Comme nous nous intéressons à des **systèmes**, nous avons besoin d'utiliser des vecteurs. Nous allons utiliser les vecteurs de la librairie *Eigen* plutôt que la librairie standard (<vector>) car beaucoup d'opérateurs ne sont pas surchargés (comme par exemple la multiplication, l'affichage ...) dans la librairie standard.

Ce TP est noté. Un compte rendu est attendu. Il doit permettre de :

- **montrer votre compréhension du code et du C++,**
- **présenter vos résultats d'un point de vue calcul scientifique.**

Exercice 1 - Prise en main de la version initiale du code

Lire le code pour ne pas perdre du temps par la suite !

1. Le code comprend deux classes : *TimeScheme* et *OdeSystem*. La première classe correspond à la définition de votre schéma en temps et la deuxième classe à la définition du système que vous souhaitez résoudre. Nous allons commencer par compiler le code :

```
g++ -std=c++11 -I Eigen/Eigen/ -o run TimeScheme.cpp OdeSystem.cpp main.cc
```

Exécuter le code. Pour le moment, votre code est assez limité, en effet vous pouvez seulement résoudre le système :

$$X' = X,\ X(0) = X_0,$$

avec $t \in [0, t_{final}]$ et $X_0 \in \mathbb{R}^n$ en utilisant le schéma d'Euler. Ce système correspond à $f(t, X) = X$. Regarder le fichier créé *solution.txt*.

2. Nous allons valider l'implémentation de la méthode d'Euler en vérifiant l'ordre de celle-ci. **Rappel** : l'ordre est obtenu par la formule suivante

$$\log 2(e_{dt}/e_{dt/2}),$$

où e_{dt} et $e_{dt/2}$ correspondent à l'erreur pour un pas de temps dt et $dt/2$ respectivement. Enlever les commentaires à la fin du fichier *main.cc*. Vérifier que vous obtenez le bon ordre pour la méthode d'Euler.

3. Afficher les différentes courbes :

$$t \mapsto X_i(t)$$

pour $i = 1, 2, 3, 4$ en utilisant `Gnuplot` et la commande *using* :

```
//Trace la courbe (t,X_1)
plot "solution.txt" using 1:2
//Trace la courbe (t,X_2)
plot "solution.txt" using 1:3
etc ...
```

Exercice 2 - Classe OdeSystem

Nous allons améliorer la classe *OdeSystem* pour pouvoir résoudre les 4 systèmes suivants :

- Premier Exemple

$$X' = X,\ X_0 \in \mathbb{R}^n,\ \text{c'est à dire } f(t, X) = X,\ X \in \mathbb{R}^n.$$

- Second Exemple

$$\left\{ \begin{array}{rcl} x' & = & -y, \\ y' & = & x \end{array} \right. \quad \text{c'est à dire } f\left(t, \begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} -y \\ x \end{pmatrix}.$$

- Troisième Exemple

$$x' = tx^2 \quad \text{c'est à dire } f(t, x) = tx^2.$$

- Le système de Lotka-Volterra (proie-prédateur)

$$\left\{ \begin{array}{rcl} x' & = & x(a - by), \\ y' & = & y(cx - d) \end{array} \right.$$

- L'équation du pendule avec ou sans frottement (à écrire sous la forme d'un système d'ordre 1)

$$\theta'' = -\frac{g}{l}\sin(\theta) - \frac{k}{ml^2}\theta'$$

Nous allons utiliser deux concepts très importants de la POO : l'héritage et le polymorphisme.

Héritage : L'héritage est une technique qui permet de créer une classe à partir d'une autre classe qui lui sert de modèle. Cela permet d'éviter d'avoir à réécrire un même code source plusieurs fois. Vous pouvez faire un héritage quand vous pouvez dire que A est un B : ici par exemple *Lotka-Volterra* est un *système d'EDO*. Nous allons donc créer des classes **filles** de la classe **mère** *OdeSystem*. Elles vont pouvoir utiliser les méthodes de la classe mère mais aussi avoir leurs propres méthodes.
Par exemple, pour chaque système d'EDO, les méthodes suivantes sont les mêmes :

```
OdeSystem();
~OdeSystem();
void InitializeFileName(const std::string file_name);
void SaveSolution(const double t, const Eigen::VectorXd & sol);
const Eigen::VectorXd & GetF() const;
```

Nous ne souhaitons donc pas recopier leurs prototypes et leurs définitions plusieurs fois.

Polymorphisme : l'idée est de créer du code qui fonctionne de différentes manières selon qui l'utilise. Ici, nous allons permettre par exemple à chaque classe fille donc à chaque système d'EDO d'avoir sa propre méthode qui construit le vecteur $f(t, X)$:

```
void BuildF(const double t, const Eigen::VectorXd & sol);
```

La puissance du polymorphisme est que la fonction s'appellera toujours *BuildF* même si elle est différente pour chaque système.

I. Premier exemple

Nous allons commencer par créer une classe **fille** de la classe **mère** *OdeSystem* qui permet de résoudre le système (qui est le même que celui de la classe mère *OdeSystem*) :

$$X' = X,\ X_0 \in \mathbb{R}^n.$$

Bien suivre les étapes suivantes (très détaillées) !

1. **Modifications à faire dans le fichier *OdeSystem.h***

a) Ajouter en dessous de la classe *OdeSystem* (après l'accolade et le point virgule) la définition de la classe fille *FirstExampleOdeSystem* :

```
// Classe fille publique d'OdeSystem
class FirstExampleOdeSystem : public OdeSystem
{
  public:
    void BuildF(const double t, const Eigen::VectorXd & sol); //f(X,t) = X
};
```

Nous ne souhaitons pas ajouter de nouvelles fonctions, seulement changer la définition de $\_f$.

b) Nous allons avoir besoin dans la classe fille du vecteur $\_f$. Pour cela nous ne pouvons pas la laisser en *private* dans la classe mère : il faut la passer en *protected* (accessibles seulement à la classe mère et aux classes filles) :

```
protected:
  Eigen::VectorXd _f; // Votre fonction f
private:
  std::ofstream _file_out; // Écriture du fichier
```

c) Ensuite pour pouvoir utiliser le **polymorphisme** il faut rajouter devant la fonction *BuildF* de la classe **mère** le mot clé *virtual* et informer qu'elle sera complétée dans les classes filles en rajoutant $= 0$ à la fin :

```
virtual void BuildF(const double t, const Eigen::VectorXd & sol) = 0;
```

d) Rajouter aussi le mot clé *virtual* devant le destructeur :

```
virtual ~OdeSystem(); // Destructeur par défaut
```

Si le destructeur n'est pas virtuel c'est le destructeur de la classe mère qui sera appelé. Il s'agit d'une règle importante : **un destructeur doit toujours être virtuel si on utilise le polymorphisme.**

2. **Modifications à faire dans le fichier *OdeSystem.cpp***

a) Supprimer la fonction *OdeSystem*::*BuildF*.

b) Construire la fonction *FirstExampleOdeSystem*::*BuildF* :

```
// Ne pas mettre le mot clé "virtual"
void FirstExampleOdeSystem::BuildF(const double t, const VectorXd & sol)
{
   _f = sol; // f(t,X) = X pour résoudre X' = X
}
```

3. **Modifications à faire dans le fichier *main.cc***
   Au lieu de définir un pointeur vers un objet de la classe mère *OdeSystem*, nous allons définir un pointeur vers un objet de la classe fille *FirstExampleOdeSystem* :

```
OdeSystem* sys = new FirstExampleOdeSystem(); // Pointeur de OdeSystem
```

4. Vérifier que les résultats obtenus sont les mêmes.

II. Deuxième exemple

1. Faire une classe fille *SecondExampleOdeSystem* pour résoudre le système :

$$\begin{cases} x' &= -y, \\ y' &= x. \end{cases}$$

2. Utiliser la solution exacte de ce système pour valider la classe *fille* : $x = x_0\cos(t) - y_0\sin(t)$ et $y = y_0\cos(t) + x_0\sin(t)$. Il est facile de remarquer que $x^2 + y^2 = x_0^2 + y_0^2$.

3. Augmenter le temps final ($t = 30$). Tracer la trajectoire $t \mapsto (x(t), y(t))$ avec `Gnuplot` :

```
plot "solution.txt" using 2:3
```

Que remarquez vous ? Que vaut $x_n^2 + y_n^2$ avec la méthode d'Euler ? Conclure.

4. Diminuer le pas de temps pour atténuer le phénomène. Zoomer pour voir qu'il persiste...

III. Troisième exemple

1. Faire une classe fille *ThirdExampleOdeSystem* pour résoudre le système : $x' = tx^2$.

2. Utiliser la solution exacte de ce système pour valider la classe *fille* : $x(t) = 2x_0/(2 - t^2x_0)$. Dans cet exemple f dépend de t (très important pour valider le schéma numérique).

3. Proposer à l'utilisateur de choisir entre les 3 systèmes en utilisant la fonction *main* suivante :

```
int main()
{
  double t0(0.), tfinal(4.), dt(0.005); // temps initial, final, pas de temps
  int nb_iterations = int(ceil(tfinal/dt)); // Définition du nombre d'itérations
  dt = tfinal / nb_iterations; // Recalcul de dt
  string results; // nom du fichier résultat
  int userChoiceSys; // Choix de l'utilisateur
  VectorXd sol0, exactSol; // Condition initiale et Solution exacte
  cout << "-----------------------------------" << endl;
  cout << "Choississez le système : " << endl;
  cout << "1) X' = X"<< endl;
  cout << "2) x' = -y et y' = x" << endl;
  cout << "3) x' = x^2 t" << endl;
  cin >> userChoiceSys;
  OdeSystem* sys(0);
  switch(userChoiceSys)
  {
    case 1:
       sys = new FirstExampleOdeSystem();
       sol0.resize(4);
```

```
      sol0(0) = 2.; sol0(1) = 3.1; sol0(2) = -5.1; sol0(3) = 0.1;
      exactSol = sol0*exp(tfinal);
      results = "first_ex.txt";
      break;
    case 2:
      sys = new SecondExampleOdeSystem();
      sol0.resize(2); exactSol.resize(2); sol0(0) = 1; sol0(1) = -1;
      exactSol(0) = sol0(0)*cos(tfinal)-sol0(1)*sin(tfinal);
      exactSol(1) = sol0(1)*cos(tfinal)+sol0(0)*sin(tfinal);
      results = "second_ex.txt";
      break;
    case 3:
      sys = new ThirdExampleOdeSystem();
      sol0.resize(1); exactSol.resize(1); sol0(0) = -2;
      exactSol(0) = 2*sol0(0)/(2-tfinal*tfinal*sol0(0));
      results = "third_ex.txt";
      break;
    default:
      cout << "Ce choix n'est pas possible ! Veuillez recommencer !" << endl;
      exit(0);
  }

  TimeScheme time_scheme; // Objet de TimeScheme
  time_scheme.Initialize(t0, dt, sol0, results, sys); // Initialisation
  time_scheme.SaveSolution(); // Sauvegarde condition initiale

  for (int n = 0; n < nb_iterations; n++)
  { // Boucle en temps
    time_scheme.Advance();
    time_scheme.SaveSolution();
  }

  if ((userChoiceSys == 1) || (userChoiceSys == 2) || (userChoiceSys == 3))
  {
    VectorXd approxSol = time_scheme.GetIterateSolution(); // Temps final
    double error = ((approxSol-exactSol).array().abs()).sum();
    cout << "Erreur = " << error<< " pour dt = " << dt << endl;
    time_scheme.Initialize(t0, dt/2., sol0, results, sys);
    for (int n = 0; n < nb_iterations*2; n++)
      time_scheme.Advance();

    approxSol = time_scheme.GetIterateSolution(); // Temps final
    double error2 = ((approxSol-exactSol).array().abs()).sum();
    cout << "Erreur = " << error2<< " pour dt = " << dt/2. << endl;
    cout << "Ordre de la méthode = " << log2(error/error2) << endl;
  }
  delete sys;
  return 0;
}
```

IV. Lotka-Volterra

À présent nous allons faire la classe fille *LotkaVolterraOdeSystem* pour résoudre :

$$\begin{cases} x' &= x(a - by), \\ y' &= y(cx - d) \end{cases}$$

1. Par rapport aux deux précédentes classes dérivées, le système dépend de 4 paramètres que nous souhaitons pouvoir initialiser dans le constructeur de la classe fille *LotkaVolterraOdeSystem*. Dans le prototype de cette classe fille qui est dans le fichier *OdeSystem.h* il faut :

- définir 4 variables privées de type *double* que nous noterons $\_a$, $\_b$, $\_c$ et $\_d$,
- ajouter un constructeur spécifique à cette classe qui va définir les 4 variables privées :

```
LotkaVolterraOdeSystem(double a, double b, double c, double d);
```

- comme précédemment ajouter le prototype de la fonction virtuelle *BuildF*.

2. Dans le fichier *.cpp*, définir le constructeur *LotkaVolterraOdeSystem* et la fonction *BuildF*.

3. Proposer ce nouveau choix à l'utilisateur dans la fonction *main*. Attention cette fois-ci, la solution exacte n'est pas connue.

4. Afficher la trajectoire $t \mapsto (x(t), y(t))$ obtenue : $x_0 = \frac{d}{c}$ et $y_0 = \frac{a}{b}$. Que remarquez-vous ?

5. Afficher la trajectoire pour $0 < x_0 < \frac{d}{c}$ et $0 < y_0 < \frac{a}{b}$ et un temps final suffisamment grand pour voir plusieurs périodes. Que remarquez vous ?

V. Le pendule

Enfin nous allons faire la classe fille *PendulumOdeSystem*.

$$\theta'' = -\frac{g}{l}\sin(\theta) - \frac{k}{ml^2}\theta'$$

1. Écrire l'équation sous la forme d'un système d'ordre 1.

2. Nous souhaitons pouvoir résoudre le pendule sans frottement (k $= 0$) et le pendule avec frottement (k > 0). Pour cela nous allons dans le prototype de la classe :

- définir 3 variables privées de type *double* : $\_l$, $\_m$ et $\_k$,
- ajouter deux constructeurs spécifiques à cette classe :

```
PendulumOdeSystem(double m, double l); // _k=0
PendulumOdeSystem(double m, double l, double k);
```

- comme précédemment ajouter le prototype de la fonction virtuelle *BuildF*.

3. Dans le fichier *.cpp*, définir les deux constructeurs *PendulumOdeSystem* et la fonction *BuildF* (rappel : $g = 9.81\text{m.s}^{-2}$).

4. Proposer ce nouveau choix à l'utilisateur dans la fonction *main*. Pour le moment, considérer le pendule sans frottement.

5. Afficher la solution $t \mapsto \theta(t)$ avec `Gnuplot` pour $l = 1m$, $m = 0.1kg$, $\theta_0 = \frac{\pi}{5}$, $\theta'_0 = 0$ et $t_{final} = 5$. Que remarquez vous ? Bien que le système étudié soit conservatif, le schéma d'Euler ne respecte pas la conservation de l'énergie. En effet, l'amplitude des oscillations doit rester constante, contrairement à ce que l'on observe. On peut montrer qu'avec le schéma d'Euler E_{n+1},

l'énergie approchée au temps t_{n+1} est supérieure à E_n. Que se passe t-il si $\_k$ est strictement positif ?

6. Nous souhaitons à présent pouvoir afficher la trajectoire de la bille qui est au bout du pendule :

$$t \mapsto (\sin(\theta(t)), -\cos(\theta(t)))$$

Nous souhaitons donc pouvoir sauvegarder dans le fichier solution les trois variables suivantes θ, $\sin(\theta)$ et $-\cos(\theta)$. La fonction *SaveSolution* de la classe mère doit alors être virtuelle. Cependant contrairement à la fonction *BuildF*, nous souhaitons qu'elle soit différente seulement pour la classe fille *PendulumOdeSystem*.

a) Dans ce cas, il faut juste ajouter le mot clé *virtual* devant le prototype de la fonction *SaveSolution* dans la classe mère et laisser sa définition dans le fichier *.cpp*.

b) Ensuite dans le prototype de la classe fille *PendulumOdeSystem*, ajouter le prototype de la fonction *SaveSolution* :

```
void SaveSolution(const double t, const Eigen::VectorXd & sol);
```

et la définir dans le fichier *.cpp*

```
void PendulumOdeSystem::SaveSolution(const double t, const VectorXd & sol)
{
  _file_out << ..... // À compléter !
}
```

c) Si vous essayez de compiler vous allez obtenir l'erreur suivante :

```
OdeSystem.cpp:88:5: error: '_file_out' is a private member of 'OdeSystem'
    _file_out << t;
    ^
```

En effet pour pouvoir modifier *_file_out* dans une classe *fille* il faut que ce soit une variable *protected* de la classe *mère* et non une variable *private*.

d) Finalement, afficher la trajectoire de la bille en utilisant la commande suivante :

```
plot "pendule.txt" using 3:4:1 with points palette
```

qui permet aux points de la trajectoire d'être colorés par la variable temps.

Exercice 3 - Classe TimeScheme

À présent nous allons améliorer la classe *TimeScheme* pour pouvoir proposer à l'utilisateur 4 schémas en temps :

• Euler Explicite • Runge Kutta 3 et 4 • Adams Bashforth 3.

1. Tout d'abord comme dans la première partie de l'exercice 2, nous allons créer une classe *EulerScheme* qui dérive de la classe *TimeScheme* :

- la fonction *Advance* doit devenir virtuelle et être définie toujours dans les classes filles (= 0 dans la classe mère),
- dans la fonction *main*, l'objet *time_scheme* est maintenant un objet de la classe fille.

Vérifier que vous obtenez les mêmes résultats que précédemment.

2. Construire la classe fille *RungeKuttaScheme3* en implémentant le schéma en temps d'ordre 3 :

$$X_{n+1} = X_n + \frac{h}{6}(k_1 + 4k_2 + k_3) \text{ avec}$$

$$k_1 = f(t_n, X_n),\ k_2 = f\big(t_n + \frac{h}{2}, X_n + \frac{h}{2}k_1\big) \text{ et } k_3 = f\big(t_n + h, X_n - hk_1 + 2hk_2\big).$$

Tester votre code sur les trois premiers systèmes d'EDO. Vérifier que le schéma est d'ordre 3.

3. De même construire la classe fille *RungeKuttaScheme4* en implémentant le schéma en temps d'ordre 4 :

$$X_{n+1} = X_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \text{ avec}$$

$$k_1 = f(t_n, X_n),\ k_2 = f\big(t_n + \frac{h}{2}, X_n + \frac{h}{2}k_1\big),\ k_3 = f\big(t_n + \frac{h}{2}, X_n + \frac{h}{2}k_2\big) \text{ et } k_4 = f(t_n + h, X_n + hk_3).$$

Tester votre code sur les 3 premiers systèmes d'EDO. Vérifier que le schéma est d'ordre 4.

3. Implémenter un switch pour proposer à l'utilisateur de choisir son schéma en temps : utiliser un pointeur pour le définir (ne pas oublier le *delete* !). Modifier le nom du fichier résultat pour qu'il contienne le nom du système et le nom du schéma en temps.

4. Les méthodes de Runge-Kutta sont rapidement coûteuses en temps puisque pour avoir un ordre n il faut faire appel n fois à la fonction *BuildF*. Une autre stratégie pour avoir un ordre n consiste à utiliser une méthode *multipas* comme par exemple les méthodes d'Adams-Bashforth. Construire la classe fille *AdBashforthScheme3* en implémentant le schéma en temps d'ordre 3 :

$$X_{n+1} = X_n + \frac{h}{12}(23f_n - 16f_{n-1} + 5f_{n-2}),\ \forall n \geq 3, \text{ avec } f_i = f(t_i, X_i).$$

Quelques conseils :
a) **Rappel :** il n'y a qu'un seul appel à faire à la fonction *BuildF* à chaque itération d'Adams-Bashforth donc il faut conserver f_{n-1} et f_{n-2}.
b) Ce qui implique que la méthode doit être initialisée... L'initialisation des schémas à pas multiple est très importante si on veut conserver le bon ordre : pour $n = 1$ et $n = 2$ recopier le code de Runge Kutta 3.
Tester votre code sur les 3 premiers systèmes d'EDO. Vérifier que votre schéma est bien d'ordre 3.

Exercice 4 - Comparaison des schémas en temps

Faire une comparaison des 3 schémas en temps pour les différents systèmes (en vrac) :

- Afficher les courbes des différentes solutions.
- Comment se comporte Runge-Kutta 4 et/ou Adams Bashforth 3 par rapport à Euler explicite ? En particulier en ce qui concerne les différentes faiblesses d'Euler évoquées dans l'exercice 2.
- Pour quel pas de temps obtient-on des erreurs similaires entre les 3 schémas ?
- Faire des comparaisons de temps de calcul ...

Conclure.

Exercice 5 (En bonus !) - Aller plus loin : templater le code

Objectif : pouvoir résoudre aussi bien des systèmes d'EDO complexes que des systèmes d'EDO réels sans recopier tout le code ...

Il y a plusieurs façons de *templater* dans le cas d'un héritage. Ici on souhaite pouvoir *templater* :

- La classe *OdeSystem* mais pas ses différentes classes filles puisqu'elles représentent soit une EDO réelle, soit une EDO complexe.
- La classe *TimeScheme* et ses différentes classes filles puisqu'un schéma en temps est le même que ce soit une EDO réelle ou une EDO complexe.

1. *Templater* la classe mère *OdeSystem* et **adapter** ses classes filles (sans les *templater* !)

Quelques conseils :

- Dans la classe **mère** *OdeSystem* il faut remplacer tous les vecteurs *VectorXd* (qui sont des vecteurs de double) par des matrices de *Eigen* qui sont *templatées* :

```
Matrix<T, Eigen::Dynamic, 1>
```

- Dans les classes **filles** de la classe *OdeSystem*. Il faut préciser que ce sont des classes qui dérivent de *OdeSystem<double>* puisque pour le moment nous n'avons pas implémenté de systèmes complexes. Pour cela il faut faire :

```
class FirstExampleOdeSystem : public OdeSystem<double>
{
  public:
    void BuildF(double t, const Eigen::VectorXd & sol);
};
```

- Pour ne pas alourdir le fichier *.h* nous allons utiliser la méthode Q.7, Ex.1, TP.4. Nous souhaitons pouvoir résoudre des EDO réelles et complexes donc il faut ajouter à la fin *.cpp* :

```
template class OdeSystem<double>;
template class OdeSystem<std::complex<double> >;
```

2. *Templater* la classe mère *TimeScheme* ainsi que toutes ses classes filles.

Quelques conseils :

- Dans la classe *TimeScheme* et dans toutes ses classes dérivées il faut remplacer tous les vecteurs *VectorXd* par des matrices de *Eigen templatées*.
- Cette fois-ci comme les classes filles sont elle aussi *templatées* elles vont dériver de la version *templatée* de la classe mère :

```
template<class T>
class EulerScheme : public TimeScheme<T>
{
  public:
    void Advance();
};
```

- Ne pas oublier que la classe *OdeSystem* a été *templatée*.
- Nous allons utiliser la même méthode que pour la classe *OdeSystem*. Ici la classe mère mais aussi les classes filles ont été templatées donc il ne faut pas en oublier :

```
template class TimeScheme<double>;
template class TimeScheme<complex<double> >;
template class EulerScheme<double>;
template class EulerScheme<complex<double> >;
etc ...
```

3. Adapter votre fichier *main.cc*. Pour le moment tous nos systèmes sont des systèmes d'EDO réels.

4. À présent compiler le code. Vous obtenez de nombreuses erreurs du style suivant :

```
TimeScheme.cpp:52:3: error: use of undeclared identifier '_t'
TimeScheme.cpp:52:9: error: use of undeclared identifier '_dt'
TimeScheme.cpp:53:3: error: use of undeclared identifier '_sys'
etc ...
```

Cela vient du fait que les variables *protected* de la classe mère *TimeScheme* ne sont pas reconnues dans les classes filles dérivées. En effet la classe mère *templatée* n'est pas instanciée durant la phase de compilation. Il faut dire explicitement au compilateur que ces variables sont dépendantes de l'instanciation de la classe mère. Une solution consiste à mettre devant les variables qui sont héritées de la classe mère **dans les fonctions des classes filles** le mot clé *this->* comme par exemple :

```
this->_t
```

5. Vérifier que vous obtenez bien les mêmes résultats que précédemment.

6. Nous souhaitons pouvoir résoudre l'équation différentielle complexe suivante :

$$z' = iz, \text{ avec } z_0 \in \mathbb{C}.$$

Pour cela implémenter la classe *ComplexExampleOdeSystem* qui dérive de la classe *OdeSystem* :

```
class ComplexExampleOdeSystem : public OdeSystem<std::complex<double> >
{
  public:
    // f(t,Z) = i Z
    void BuildF(double t, const Eigen::VectorXcd & sol);
    // solution : Z = a + i b ; sauvegarde sur 3 colonnes : t, a, b
    void SaveSolution(double t, const Eigen::VectorXcd & sol);
};
```

Construire un nouveau fichier *main_ complex.cc* pour résoudre cette ODE. Vérifier votre code avec la solution exacte.
Remarque : ce système est équivalent au système défini dans la classe *SecondExampleOdeSystem* avec : $z = x + iy$. Pour résoudre ce système avec votre code, quelle est la meilleure stratégie ?