TT2 Génie Logiciel

Constanza Corentin

May 14, 2024

Introduction

Lors de ce TT, nous souhaitons résoudre des systèmes différentiels, en faisant intervenir différents types de schéma d'intégration temporelle. On s'interressera plus particulièrement au schéma d'Adams (explicite et implicite) ainsi que la méthode BDF (implicite).

Descendance de classes

Nous avons *IntegreScheme* qui est la classe mère, elle permet de définir un schéma de manière général.

Nous avons aussi *schemeAdamsExplicit*, classe fille de la classe *IntegreScheme*. Elle utilise les éléments construits précedement par *IntegreScheme*, tout en définissant les élément propre au schéma d'Adams explicite, tel que sont les ordre, et les coefficients multiplicateurs liés aux ordres.

Il en va de même pour *schemeAdamsImplicit* ainsi que *schemeBDF*.

Si l'on souhaitait ajouter de nouveau schéma, il nous suffit d'ajouter une classe, et de définir par ses constructeurs, les éléments qui le différencie et qui sont nécessaire à l'application de ce schéma.

Classe IntegreScheme

La classe IntegreScheme, classe mère represente un schéma abstrait. Elle contient les differentes méthodes permettant d'initialiser les paramètres nécessaires à la mise en place d'un shéma en général, en plus de méthodes applicablent à tout les shémas.

La méthode *setInitialCondition* permet d'intialiser le vecteurs de conditions initilales du système.

La méthode *setTolerence*, qui permet d'initialiser les valeurs des tolérance pouvait être utilisée dans les critères d'arrêt.

La méthode *setStepsize* permet d'initialiser le pas de temps utilisé dans les schémas de résolution.

La méthode *setPtFunction* initialise le pointeur sur la fonction, déterminant l'aspect de notre système. La méthode *IntergreTimeIntervalle*, est une méthode virtuel, permettant de calculer une discretisation sur un intervalle $[t_0, t_n]$ de la solution du système. Ce résultat est stocké dans un vecteur de taille égal au nombre d'itération, soit $\frac{t_n - t_0}{h}$.

Classe schemeAdamsExplicit

Cette classe permet la résolution d'un système par le schéma d'Adams explicite à differents ordre.Il contient les différentes méthodes canoniques de construction et de destruction. La méthode implémenté la plus intéressante est la méthode *IntegreTimeIntervalle*. En voici les détails :

```
template <class T> vector<T> schemeAdamsExplicit<T>::
            integreTimeInterval(double t0, double tf){
```

```
	int i, iter;
	T zero;
	iter=(int) (floor( (tf-t0)/this->h) );
	vector<T> res(iter+1,zero);
	this->f = vector<T> (this->k,zero);

	res[0] = this->y0;
	this->f[0] = this->ptf(t0,res[0]);

	if(k==1)
	{
		for(i=1;i<iter;i++)
		{
			res[i] = res[i-1] + this->h * this->f[0];
			this->f[0] = this->ptf(t0 + i * this->h, res[i]);
		}
	}

	if(k==2)
	{
		res[1] = res[0] + this->h*this->f[0];
		f[1] = this->ptf(t0+this->h,res[1]);

		for(i=2;i<iter;i++)
		{
			res[i] = res[i-1] + this->h *(this->coef[2][0]*this->f[1] + this->coef[2][1]*this->f[0]);
			f[0]=f[1];
			this->f[1] = this->ptf(t0 + i * this->h, res[i]);
		}
	}

	if(k==3)
	{
		res[1] = res[0] + this->h*this->f[0];
		f[1] = this->ptf(t0+this->h,res[1]);
		res[2] = res[1] + this->h*(this->coef[2][0]*this->f[1] + this->coef[2][1] * this->f[0]);
		f[2] = this->ptf(t0 + 2*this->h,res[2]);

		for(i=3;i<iter;i++)
		{
			res[i] = res[i-1] + this->h *(this->coef[3][2]*this->f[2] + this->coef[3][1]*this->f[1] + this->coef[3][0]*this->f[0]);
			f[0]=f[1];
			f[1]=f[2];
			this->f[2] = this->ptf(t0 + i * this->h, res[i]);
		}
	}

```

```
        if(k==4)
        {
                res[1] = res[0] + this->h*this->f[0];
                f[1] = this->ptf(t0+this->h,res[1]);
                res[2] = res[1] + this->h*(this->coef[2][0]*this->f[1] + this->coef[2][1] * this->f[0]);
                f[2] = this->ptf(t0 + 2*this->h,res[2]);
                res[3] = res[2] + this->h*(this->coef[3][0]*this->f[2] + this->coef[3][1]*this->f[1] + this->coef[3][2]*this->f[0]);
                f[3] = this->ptf(t0 + 3*this->h, res[3]);

                for(i=1;i<iter;i++)
                {
                        res[i] = res[i-1] + this->h *(this->coef[4][0]*this->f[3] + this->coef[4][1]*this->f[2] + this->coef[4][2]*this->f[1] + this->coef[4][3]*this->f[0]);
                        f[0]=f[1];
                        f[1]=f[2];
                        f[2]=f[3];
                        this->f[3] = this->ptf(t0 + i * this->h, res[i]);
                }
        }
        return res;
}
```

Classe schemeAdamsImplicit

Cette classe permet la résolution d'un système par le schéma d'Adams implicite à differents ordres.Il contient les différentes méthodes canoniques de construction et de destruction. Comme pour Adams explicite, la méthode implémenté la plus intéressante est la méthode *IntegreTimeIntervalle*. En voici les détails :

```
template <class T> vector<T> schemeAdamsImplicit<T>::integreTimeInterval(double t0, double tf)
{
        int iter = (int) (floor( (tf-t0)/(this->h) ));
        int i = 0;
        T zero;
        T estimation;
        vector <T> res(iter, zero);

        res[0]= this->y0;
        this->f = vector<T>(this->k+1, zero);
        this->f[0] = this->ptf(t0, res[0]);

        if(k==0)
        {
                for(i=1;i<iter;i++)
                {
                        estimation = res[i-1] + this->h * this->f[0];
```

```
            estimation = newton(estimation, t0+ i*this->h, res[i-1],i);
            this->f[0] = this->ptf(t0 + i * this->h, res[i]);
        }
    }

    if(k==1)
    {
        for(i=1;i<iter;i++)
        {
            estimation = res[i-1] + this->h * this->f[0];
            estimation = newton(estimation, t0+ i*this->h, res[i-1],i);

            this->f[1] = this->ptf(t0 + i*this->h, res[i-1]);
            res[i] = res[i-1] + this->h *(this->coef[k][0]*this->f[1] + this->coef[k][1]*this->f[0]);
            this->f[1] = this->ptf(t0 + i*this->h, res[i]);
            this->f[0]=f[1];
        }
    }

    if(k==2)
    {
        estimation = res[0] + this-> h * this->f[0];
        this->f[1] = this->ptf(t0 + this->h, estimation);
        res[1] = res[0] + this->h*(this->coef[k][0]*this->f[1] + this->coef[k][1]*this->f[0]);
        this->f[1] = this->ptf(t0 + this->h, res[1]);

        for(i=2;i<iter;i++)
        {
            estimation = res[i-1] + this->h * this->f[1];
            estimation = newton(estimation, t0+ i*this->h, res[i-1],i);

            this->f[2] = this->ptf(t0 + i*this->h, estimation);
            res[i] = res[i-1] + this->h *(this->coef[k][0]*this->f[2] + this->coef[k][1]*this->f[1]+ this->coef[k][2]*this->f[0]);
            this->f[0] = this->f[2];
            this->f[2]=this->ptf(t0+i*this->h,res[i]);
            this->f[1]=f[2];
        }
    }

    if(k==3)
    {
```

```
                    estimation = res[0] + this-> h * this->f[0];
                    this->f[1] = this->ptf(t0 + this->h, estimation);
                    res[1] = res[0] + this->h*(this->coef[k][0]*this->f
                                    [1] + this->coef[k][1]*this->f[0]);
                    this->f[1] = this->ptf(t0 + this->h, res[1]);

                    estimation = res[1] + this-> h * this->f[0];
                    this->f[1] = this->ptf(t0 + 2*this->h, estimation);
                    res[2] = res[1] + this->h*(this->coef[k][2]*this->f
                                    [2] + this -> coef[k][1] * this->f[1]+
                                    this->coef[k][2]*this->f[0]);
                    this->f[2] = this->ptf(t0 + 2*this->h, res[2]);

                    for(i=3;i<iter;i++)
                    {
                              estimation = res[i-1] + this->h*f[2];
                              estimation = newton(estimation, t0 + i*this
                                                ->h, estimation, i);
                              this->f[3] = this->ptf(t0 + i*this->h,
                                                estimation);
                              res[i] = res[i - 1] + this->h*(this->coef[k
                                                ][0] * this -> f[3] + this->
                                                coef[k][1]*this->f[2]+ this->
                                                coef[k][2]*this->f[1] + this->
                                                coef[k][3]*this->f[0]);
                              this->f[0] = this->f[1];
                              this->f[1] = this->f[2];
                              this->f[3] = this->ptf(t0 + i*this->h, res[i
                                                ]);
                              this->f[2] = this->f[3];
                    }
          }
          return res;

}
```

Test unitaire sur les problèmes donnés

Concernant le shéma d'Adams Explicite, j'ai pu effectuer des tests sur trois systèmes différents:

Le premier est implémenter comme ci-dessous :

```
double f(double t, double x){
          return x;
}
```

De solution $u(x) = e^x$, sur l'interval $[0, 1]$, nous devrions donc pouvoir approximé e au final.

La dernière valeurs obtenue lorsque nous testons pour ce problème à l'ordre 3 avec un pas $h = 0.001$ est de 2.70749 ce qui est une bonne approximation de e.

Le second est implémenter comme ceci :

```
double f2(double t, double x){
        return t;
}
```

De solution $u(x) = \frac{x^2}{2}$, sur l'interval $[0, 1]$, nous devrions donc avoir une convergence vers $\frac{1}{2}$ au final.

La dernière valeurs obtenue lorsque nous testons pour ce problème à l'ordre 2 avec un pas $h = 0.001$ est de 0.499 ce qui est une bonne approximation de $\frac{1}{2}$.

Les deux systèmes précedante sont des cas particuliers du système n°3 de la forme :

$$\left\{ \begin{array}{l} \text{u'(x)}=\lambda u(x) + \alpha cos(\omega t) \\ \text{u(t}_0\text{)} \;=\; u_0 \;\;, t_0 \;=\; 0 \end{array} \right. \tag{1}$$

Implémenter comme ceci :

```
double f3(double t,double x){
        double a, om,lmd;
        a=1.;
        om=1.;
        lmd=0;

        return lmd*x+a*cos(om*t);
}
```

Avec les valeurs de λ, α, ω données ainsi que les conditions initials, la solution du système est $u(x) = sin(x)$. Sur $[0, 1]$, à l'ordre 4 et pour un pas $h = 0.001$ nous obtenons 0.84093 qui est une bonne approximation de $sin(1)$.

Pour le schéma d'Adams Implicite, j'ai testé le systeme n°3 avec un pas de $h = 0.001$ à l'ordre 2. Nous obtenons finalement : 0.841052 qui comme précédament est une bonne approximation de $sin(1)$.

Axes d'amélioration

Je n'ai malhereusement pas reussi à implementer ces méthodes pour des vecteurs, seulement pour des scalaires, ce qui limites grandement l'application et les tests possibles. Le premier points d'amélioration serait donc de reussir ceci pour pouvoir tester les schéma de Lotka-Volterra et de Bratu. Un autre points d'amélioration dans la qualité du rapport serait d'utiliser les fichiers crée par notre programme principal dans un script Matlab/octave afin d'avoir des courbes ce qui serait plus visuels afin de confirmé nos tests. Enfin il faudrait implementer le BDF.
On pourrait aussi ajouter des comparaisons des differents schéma en terme de temps de calculs et de precisions en fonction des ordres ainsi que des pas utiliser.