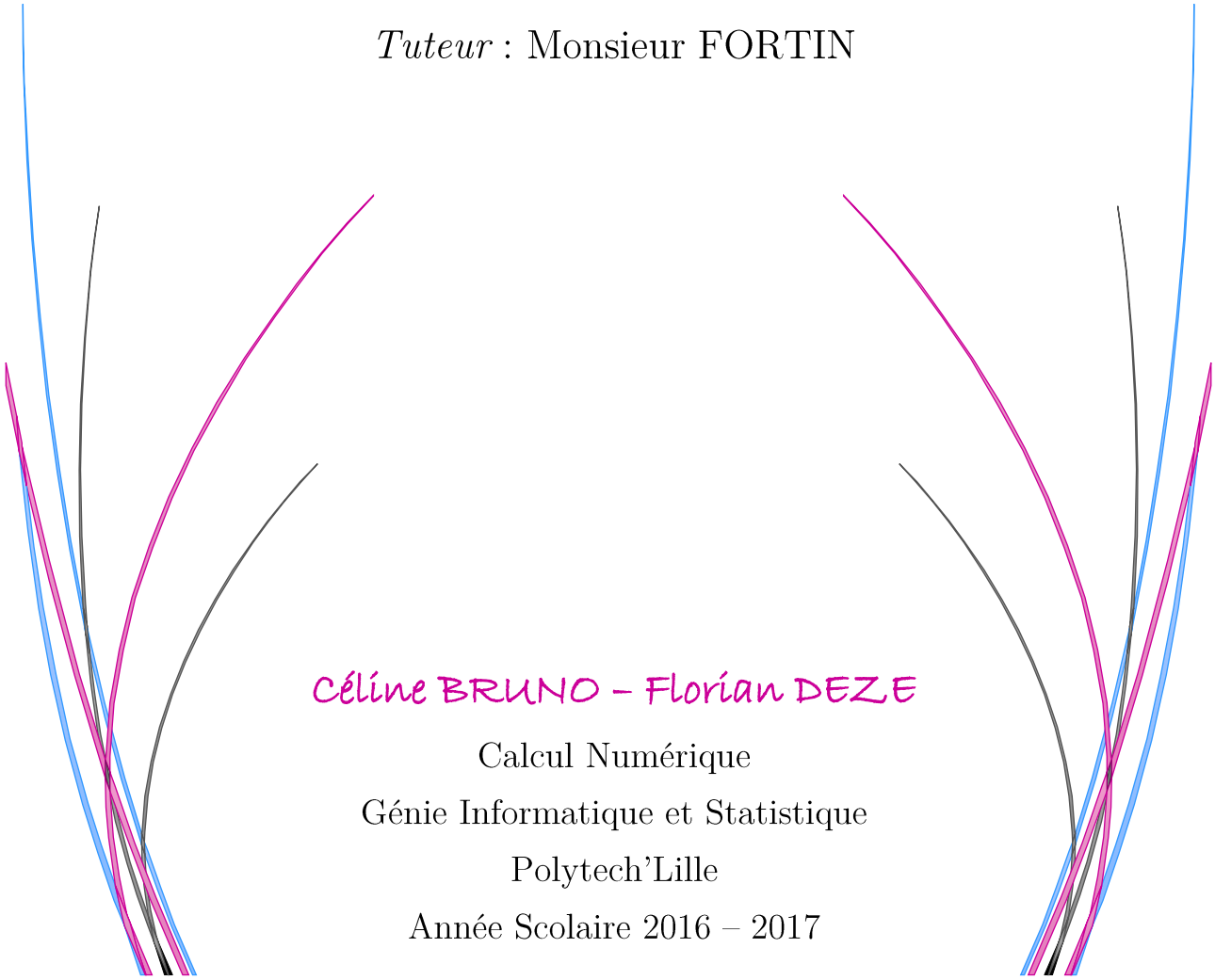
Intégration Numérique, Equations Différentielles

Rapport de projet

Tuteur : Monsieur FORTIN



Céline BRUNO – Florian DEZE

Calcul Numérique

Génie Informatique et Statistique

Polytech'Lille

Année Scolaire 2016 – 2017

Table des matières

Introduction 3

1 Intégration Numérique 4
1.1 Description des fichiers 4
1.2 Fonctions simples, fonctions complexes 5
1.3 Formule des Trapèzes Composites 5
1.4 Formule de Simpson 7
1.5 Formule de Weddle 8
1.6 Programme Principal 9

2 Equations Différentielles 10
2.1 Description des fichiers 10
2.2 Questions préliminaires 11
2.2.1 Utilisation des variables `C2` et `A12` 11
2.2.2 Implantation du schéma de Heun 11
2.2.3 Schéma quelconque 13
2.2.4 Observation expérimentale de l'ordre du schéma 13
2.2.5 Problème d'Arenstorf 15
2.3 Nouveau schéma, problème et questions optionnelles 16
2.3.1 Schéma RK4 16
2.3.2 Le problème Hodgkin-Huxley 18
2.3.3 Question Optionnelle : pas adaptatif 19

Conclusion 21

Introduction

Dans le cadre de notre formation en Génie Informatique et Statistique, en quatrième année, à Polytech Lille, nous avons eu un cours de Calcul Numérique suivi d'un projet dont l'objectif étant d'appliquer les notions vues lors de ce cours.

Ce projet est partagé en deux parties. La première consiste à l'implantation de schémas numériques d'intégration en se basant sur les formules de Newton-Cotes. La seconde se consacre à l'intégration numérique des systèmes d'équations différentielles. Les programmes sont réalisés en FORTRAN 77.

Dans ce rapport, nous commencerons dans un premier temps par détailler les fonctions prises comme exemples et la pertinence dans leur choix. Nous passerons ensuite aux schémas d'implantations numériques tels que la formule des Trapèzes composites et Simpson, en justifiant la justesse de nos programmes. Enfin, nous étudierons le schéma de Weddle, avec les contraintes qu'il impose. Dans un second temps nous expliquerons comment nous avons résolu les problèmes tels que l'orbite d'Arenstorf ou de Hodgkin-Huxley.

1 Intégration Numérique

Dans cette première partie, il s'agit de réaliser un programme qui implante certains schémas numériques d'intégration basés sur les formules de Newton-Cotes et qui vérifie que les calculs sont corrects en observant expérimentalement l'ordre des formules c'est-à-dire en regardant le nombre de bits exacts. Elle est réalisée dans son intégralité.

Nous avons dû dans un premier temps, choisir des fonctions $p(x)$ pas trop compliquées mais pas trop simples non plus. Puis nous avons réalisé un programme principal qui fonctionne pour n'importe quel schéma de Newton-Cotes et enfin nous avons programmé la formule des trapèzes composites, la formule de Simpson et nous avons choisi de programmer aussi la formule de Weddle.

1.1 Description des fichiers

Une fois l'archive décompressée, un premier dossier nommée *Part 1* nous intéresse dans cette première partie du projet. A l'intérieur de ce dossier, 14 fichiers :

- Le programme principal : `MAIN.f`
- La formule des trapèzes composites : `trapeze.f`
- La formule de Simpson : `simpson.f`
- La formule de Weddle : `weddle.f`
- Différent exemples composés d'une fonction et d'une primitive :
  - Exemples de degré 1, 2, 4 et 6
  - Exemple de fonction exponentielle

Pour compiler, il faut impérativement mettre le programme principal, un schéma de Newton-Cotes et un exemple (fonction et primitive). Par exemple, la ligne de commande pour la formule des trapèzes composites sur l'exemple de degré 2 est la suivante :

```
gfortran -Wall -fimplicit-none MAIN.f trapeze.f exemple_function_degre2.f
exemple_primitive_degre2.f
```

Afin de pousser les calculs plus loin, on peut transformer toutes les variables et les constantes double précision en quadruple précision en ajoutant `-fdefaut -real-8` :

```
gfortran -Wall -fimplicit-none -fdefaut -real-8 MAIN.f trapeze.f
exemple_function_degre2.f exemple_primitive_degre2.f
```

1.2 Fonctions simples, fonctions complexes

Une fonction « simple » est un polynôme de degré supérieur à deux. Par exemple $f(x) = 2x + 1$ est une fonction « trop simple » à intégrer numériquement. De plus c'est une fonction facile à intégrer mathématiquement. Une fonction compliquée est par exemple $f(x) = \frac{exp^{(x^2)}}{x+1}$ c'est-à-dire toute fonction à base d'exponentielle ou de logarithme. Peu importe l'ordre de la méthode utilisée, l'approximation de l'aire calculée ne sera identique à l'aire réelle[1].

Afin de construire nos exemples, nous n'avons pas pris une fonction afin d'en déduire sa primitive mais l'inverse car il est plus facile de dériver une fonction que de l'intégrer. Nous sommes partis d'une primitive et nous avons demandé au logiciel MAPLE de la dériver afin d'obtenir la fonction correspondante. Nous avons construit cinq exemples : une fonction compliquée, une fonction très simple (polynôme de degré un) et trois autres fonctions. Le tableau suivant répertorie tous les exemples utilisés.

	Fonction	Primitive
Exemple ordre 1	$f(x) = 2x + 2$	$F(x) = x^2 + 2x + 1$
Exemple ordre 2	$f(x) = 6x^2 + 4x + 42$	$F(x) = 2x^3 + 2x^2 + 42x$
Exemple ordre 4	$f(x) = 5x^4 + 4x$	$F(x) = x^5 + 2x^2$
Exemple ordre 6	$f(x) = \frac{3}{200}x^6 + 9x^4$	$F(x) = \frac{x^7}{600} + 3x^3$
Exemple exponentielle	$f(x) = \frac{2\,x\,e^{(x^2)}}{x+1} - \frac{e^{(x^2)}}{(x+1)^2}$	$F(x) = \frac{e^{(x^2)}}{x+1}$

1.3 Formule des Trapèzes Composites

Une formule de quadrature est dite d'ordre p si elle est exacte pour tous les polynômes de degré strictement inférieur à p. Par conséquent, la formule des Trapèzes Composites est d'ordre 2[2] donc elle est exacte pour tous les polynômes de degré strictement inférieur à 2, soit des polynômes de degré 1.

[1] L'aire réelle *reel* est calculée grâce à la primitive de la fonction : *reel* = P(B) – P(A).

[2] L'ordre de la formule des trapèzes composites est admis. De manière plus générale, tous les ordres des schémas de Newton-Cotes sont admis.

Avant de vérifier l'exactitude de nos résultats, nous allons détailler rapidement le code de la formule des Trapèzes.

Nous calculons dans un premier temps l'aire réelle RESP. Ceci nous permettra de comparer les résultats à la fin de l'exécution du programme. Puis à l'intérieur d'une boucle DO, nous remplissons le vecteur RES qui contient la valeur de la fonction pour tous les points x_i. Ensuite, nous calculons l'aire à partir des valeurs de RES. Enfin, nous calculons le nombre de bits exacts en utilisant la formule $nb\ bits = \dfrac{-log\left|\dfrac{calculee - reel}{reel}\right|}{log\,2}$ où *calculee* est l'aire calculée dans notre programme et *reel* est l'aire réelle. Cette formule est la même peu importe le schéma de Newton-Cotes, c'est la formule générale du calcul du nombre de bits exacts.

Vérifions maintenant sur un exemple l'ordre de la formule des Trapèzes Composites.

```
$ gfortran -Wall -fimplicit-none MAIN.f trapeze.f exemple_function_degre1.f
exemple_primitive_degre1.f
$ ./a.out
      NBSTEPS        NB BITS EXACT
            2                  Infinity
            4                  Infinity
            8                  Infinity
           16                  Infinity
           32                  Infinity
           64                  Infinity
          128                  Infinity
          256                  Infinity
          512                  Infinity

 ---- Resultat du calcul de la primitive ----
 Resultat :    32.000000000000000

 -------------- Aire calculee ---------------
 Aire :        32.000000000000000
```

Dès les premiers pas, le nombre de bits exacts est « infini » c'est-à-dire que la précision est très grande. On obtient bien une approximation exacte de l'aire : l'aire calculée est identique au résultat du calcul de la primitive c'est-à-dire identique à l'aire réelle. Sur cet exemple, on ne peut pas démontrer que la formule des trapèzes composites est d'ordre 2 car la fonction est trop simple, on obtient immédiatement le bon résultat.

```
$ gfortran -Wall -fimplicit-none MAIN.f trapeze.f exemple_function_degre4.f
exemple_primitive_degre4.f
$ ./a.out
```

```
      NBSTEPS        NB BITS EXACT
            2   1.9587517137324828
            4   3.9446765285207603
            8   5.9411790842845331
           16   7.9403060464116644
           32   9.9400878694674475
           64   11.940033330386413
          128   13.940019695938300
          256   15.940016287346406
          512   17.940015435199690

 ---- Resultat du calcul de la primitive ----
 Resultat :    3172.0000000000000

 -------------- Aire calculee ---------------
 Aire :        3172.0126139298081
```

Sur ce second exemple, plusieurs remarques. L'aire calculée avoisine l'aire réelle (résultat du calcul de la primitive). Ce constat est normal puisque la fonction est de degré 4 et que la formule des trapèzes composites est exacte pour des polynômes de degré strictement inférieur à 2. De plus, le nombre de bits exacts augmente de deux en deux lorsque l'on multiplie le pas par deux. Ceci démontre l'ordre 2 de la formule des trapèzes composites. Enfin, le nombre de bits exacts est une première vérification car il permet de détecter une erreur dans le code. Si le nombre de bits exacts ne passe pas de deux en deux lorsque l'on multiplie le pas par deux, nous savons qu'il y a une erreur dans le programme.

1.4 Formule de Simpson

La formule de Simpson est d'ordre 4 par conséquent elle est exacte pour tous les polynômes de degré strictement inférieur à 4. Reprenons notre exemple de degré 4, l'aire calculée par la formule de Simpson n'est pas identique à l'aire réelle mais nous obtenons une approximation plus précise qu'avec la formule des Trapèzes Composites.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 MAIN.f simpson.f
exemple_function_degre4.f exemple_primitive_degre4.f
$ ./a.out
      NBSTEPS        NB BITS EXACT
            2   6.2161395564251346183227538211897518 8
            4   10.216139556425134618322753821189816 6
            8   14.216139556425134618322753821190884 3
           16   18.216139556425134618322753821207957 3
           32   22.216139556425134618322753821481097 3
           64   26.216139556425134618322753825851346 7
          128   30.216139556425134618322753895775333 3
          256   34.216139556425134618322755014559112 8
          512   38.216139556425134618322772915099629 0
```

```
---- Resultat du calcul de la primitive ----
 Resultat :    3172.000000000000000000000000000000

 -------------- Aire calculee ---------------
 Aire :        3172.000000009934107462565104166666654
```

Nous vérifions que notre schéma et d'ordre 4 par le biais du nombre de bits exacts, ce qui est le cas. Pour nos exemples de degré 1 et 2, nous obtenons un nombre de bits exacts infini dès les premières itérations c'est-à-dire que la formule de Simpson approche exactement la valeur réelle.

1.5 Formule de Weddle

La formule de Weddle est d'ordre 8 par conséquent elle est exacte pour tous les polynômes de degré strictement inférieur à 8. Regardons un exemple de degré 6. Nous avons effectué les calculs en quadruple précision afin d'obtenir une meilleure précision.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 MAIN.f weddle.f
exemple_function_degre6.f exemple_primitive_degre6.f
$ ./a.out
      NBSTEPS         NB BITS EXACT
            6   15.130277520020696767914273868332104 0
           12   23.130277520020696767914273875455279 1
           24   31.130277520020696767914275312964237 4
           48   39.130277520020696767914863821640274 5
           96   47.130277520020696768050229666884406 1
          192   55.130277520020696790886844228571899 6
          384   63.130277520020701801892553764578769 3

 ---- Resultat du calcul de la primitive ----
 Resultat :    3627.206666666666666666666666666690

 -------------- Aire calculee ---------------
 Aire :        3627.206666666666670259732155152298 9
```

Le nombre de bits augmente de 8 à chaque ligne donc l'ordre de la formule ce quadrature est 8, ce qui correspond à la formule de Weddle. Remarquons ici, une contrainte supplémentaire sur les nombre de pas NBSTEPS : ils doivent être un multiple de 6 et mieux encore, le nombre d'intervalle N doit lui aussi être un multiple de 6 !

Souvenez-vous maintenant de notre exemple compliqué :

$$f(x) = \frac{2\,x\,e^{(x^2)}}{x+1} - \frac{e^{(x^2)}}{(x+1)^2} \quad \text{et} \quad F(x) = \frac{e^{(x^2)}}{x+1}.$$

Nous avions dit que peu importe l'ordre de la méthode de Newton-Cotes utilisée, l'aire calculée ne sera jamais exacte à l'aire réelle. C'est ce que nous allons regarder dans l'exemple suivant.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 MAIN.f weddle.f
exemple_function_exponentielle.f exemple_primitive_exponentielle.f
$ ./a.out
      NBSTEPS         NB BITS EXACT
            6   9.02988082162600532623048953286545400E-0002
           12   2.63773755232151193899677566479416512
           24   6.79271347924569988722663462739452039
           48   12.66795034447854244289548126125937790
           96   19.78987358695706524649832365793249 51
          192   27.51355495562057178337956192982782 73
          384   35.43903119364442917358514797309360 78

 ---- Resultat du calcul de la primitive ----
 Resultat :    12000816554.87183783979736929334088 25

 -------------- Aire calculee ---------------
 Aire :        12000816555.12946959398375388864665 46
```

Nous obtenons une approximation de l'aire réelle. Nous remarquons que le nombre de bits exacts ne nous donne pas clairement l'information sur l'ordre de la formule utilisée. Ce n'est qu'à partir d'un nombre de pas élevé que nous pouvons déduire l'ordre de la formule utilisée (à partir de 96 pas).

1.6 Programme Principal

Le programme principal nommé `MAIN.f` n'a pas besoin d'être modifié en fonction d'un schéma ou en fonction d'un autre. En effet, à l'intérieur du programme principal, nous définissons les bornes d'intégration : `A` et `B`. Ensuite nous définissons le nombre d'intervalle `N`. Puis nous faisons appel à une subroutine `INTERPOLATION` avec comme paramètre les bornes d'intégration (`A` et `B`), `RES`, le nombre d'intervalle d'intégration et l'aire calculée à l'intérieur de la subroutine. Cette subroutine nous permet d'appliquer les schémas de Newton-Cotes. Nous avons fait attention à ce que tous les schémas possède la même signature de fonction ainsi que le même nom de sous fonction. Et c'est grâce à cela, que nous ne devons pas changer le programme principal. Enfin, le programme principal affiche l'aire calculée.

2 Equations Différentielles

Cette seconde partie est consacrée à l'intégration numérique de systèmes d'équations différentielles. Notre but est de créer des intégrateurs de systèmes d'équations différentielles permettant de résoudre différents problèmes avec les schémas de Runge-Kutta.

Cette partie est décomposée en trois sous parties. La première explique comment compiler et utiliser les fichiers du dossier. La seconde consiste à prendre le code préalablement fournie en main en créant le schéma de Heun et en améliorant le programme principale pour qu'il puisse être utilisé avec n'importe quel schéma. Cette partie explique comment on peut vérifier expérimentalement l'ordre d'un schéma. La troisième partie consiste à programmer le schéma RK4, le problème Hodgkin-Huxley et de transformer l'intégrateur à pas fixe en pas adaptatif.

2.1 Description des fichiers

Le second dossier *Part2* nous intéresse dans cette deuxième partie du projet. La compilation des fichiers s'effectue toujours avec trois fichiers, un programme principal, dont nom de fichier commence par `main`, un schéma avec le nom commençant par `sch` et enfin le problème à résoudre, dont le nom de fichier commence par `pb`.

En fonction des étapes du projet et des tests, les commandes de compilation et le nom des fichiers seront précisés.

Les fichiers commençant par `plot_2D` permettent de créer une courbe dans un fichier `png` à partir de `gnuplot`.

Exemple.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_pas_fixe.f pb_Arenstorf.f
sch_Heun.f
$ ./a.out > Arenstorf.txt
$ gnuplot plot_2D_Arenstorf.gp
```

Remarque. Si on souhaite tracer un graphique pour un problème donné, il faut utiliser le fichier `plot_2D_nomDuProblème.gp`. Le fichier pris en entré doit être un fichier `.txt` avec un nom particulier.

Problème	Exécution 1	Exécution 2	Résultat
Problème exponentielle	`./a.out > expo.txt`	`gnuplot plot_2D_exponentielle.gp`	`courbe_expo.png`
Problème Arenstorf	`./a.out > Arenstorf.txt`	`gnuplot plot_2D_Arenstorf.gp`	`courbe_Arenstorf.png`
Problème Oscillateur Harmonique	`./a.out > oscillateur.txt`	`gnuplot plot_2D_oscillateur.gp`	`courbe_oscillateur.png`

2.2 Questions préliminaires

2.2.1 Utilisation des variables C2 et A12

Les constantes c_2 et a_{12} du schéma de Runge sont remplacés dans le sous-programme `CALCUL_Y1_Y1HAT`. Nous remplaçons les lignes `YTEMP(I) = Y0(I) + H/2D0*K(I,1)` par `YTEMP(I) = Y0(I) + A(2,1)*H*K(I,1)`

`CALL PROBLEM_F(N, X0+H/2D0, YTEMP, K(1,2))` par `CALL PROBLEM_F(N, X0+C(2)*H, YTEMP, K(1,2))`

Nous avons vérifié nos résultats en exécutant le programme avec le schéma de Runge et le problème exponentielle et en observant l'erreur globale obtenue à la fin de l'exécution. L'erreur globale étant faible et la même que précédemment nous en déduisons que la modification est correcte.

Le fichier est par la suite modifié pour qu'il s'adapte au schéma de Heun. Par conséquent il sera impossible de le compiler à nouveau avec Runge et d'obtenir ce résultat.

2.2.2 Implantation du schéma de Heun

Pour réaliser le schéma de Heun (dans le fichier `sch_Heun.f`) nous avons repris la structure du programme du schéma de Runge (`sch_Runge.f`) et nous avons ajouté et modifié les constantes.

Nous avons ensuite ajouté les lignes de codes nécessaire dans le fichier principal (`main_pas_fixe.f`), c'est-à-dire une nouvelle boucle « pour » et un troisième appel au sous-programme `Probleme_F` permettant de prendre en compte l'ordre 3 du schéma de Heun.

Pour vérifier si notre schéma est correct, nous avons compilé le programme `sch_Heun.f` avec les fichiers `main_pas_fixe.f` et `pb_Arenstorf.f`. Les résultats obtenus sont :

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_pas_fixe.f pb_Arenstorf.f
sch_Heun.f

# Erreur relative globale = 3.77576020880981489E-003
```

Nous l'avons aussi testé avec le `pb_exponentielle.f` :

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_pas_fixe.f
pb_exponentielle.f sch_Heun.f

1.0000000000010041 2.7182818284590775
# Erreur relative globale = 1.19261041955433148E-014
```

On remarque que pour $x_{end} = 1$ la valeur obtenue est très proche de e, avec une erreur relative de l'ordre de 10^{-14}, qui est plus faible qu'avec le schéma de Runge :

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_pas_fixe.f
pb_exponentielle.f sch_Runge.f

1.0000000000010041 2.7182818281758792
# Erreur relative globale = 1.04170925979683157E-010
```

Cela s'explique par l'ordre du schéma de Heun qui est plus élevé que celui de Runge. Donc l'erreur globale est plus petite avec le schéma de Heun qu'avec celui de Runge.

Preuve. Si on intègre de x_0 à x_{end} avec un pas de h, au moyen d'un schéma de Runge-Kutta d'ordre p alors, il existe une constante $K > 0$ tel que l'erreur globale vérifie :

$$|y(x_{end}) - y_N| \leq K h^p$$

On remarque que pour un pas $h[0,1]$, la méthode est d'autant plus précise que l'ordre est élevé.

Le moyen de vérifier le résultat concrètement est d'observer la vitesse de convergence du résultat. Cependant cette fonctionnalité sera implémentée dans les parties suivantes qui consiste à adapter le programme principale à l'utilisation d'un schéma quelconque et au calcul et l'affichage du nombre de bits exactes de la solution calculé en fonction du nombre de pas.

2.2.3 Schéma quelconque

L'adaptation de programme principale à un schéma de Runge-Kutta quelconque se décompose en trois parties : le calcul des k_i, le calcul des y_{1i} et enfin le calcul de $\hat{y}_{1i}$ avec $i[0,n]$. La difficulté était de trouver des algorithmes généraux pour ces trois parties.
La modification a été réalisée dans le fichier `main_pas_fixe2.f`.

Afin de nous assurer que les algorithmes fonctionnent il faut à la fois retrouver le bon résultat final avec une erreur faible équivalente à celle obtenue avec le programme `main_pas_fixe`. Nous devons ensuite vérifier l'ordre du schéma expérimentalement, car des algorithmes faux peuvent converger vers un résultat correct, seule la vitesse et/ou l'ordre de grandeur de l'erreur change. La partie suivante concerne l'observation expérimentale de l'ordre d'un schéma quelconque et garde l'algorithme du fichier `main_pas_fixe2.f`. Par conséquent, tous les tests sont effectués dans la sous-partie suivante.

2.2.4 Observation expérimentale de l'ordre du schéma

Nous avons écrit un nouveau programme principale dans le fichier `main_nbbits_pas_fixe.f` afin d'observer expérimentalement l'ordre de n'importe quel schéma de Runge-Kutta.

Les résultats attendus dépendent de l'ordre du schéma utilisé. Si un schéma est d'ordre 2 (comme celui de Runge), alors l'évolution du nombre de bits exacts est de 2 lorsque le nombre de pas double. Pour un schéma d'ordre 3 (celui de Heun) l'évolution du nombre de bits exacts augmente de 3 lorsque le nombre de pas double.

Explication. Pour un schéma d'ordre p inconnu, on observe deux valeurs :

NBSTEPS	Erreur globale	Longueur du pas
4	2^{-12}	$h_1 = \frac{1}{4}$
8	2^{-14}	$h_2 = \frac{1}{8} = \frac{h_1}{2}$

On pose que l'erreur est h^p.
On sait que $h_1^p = 2^{-12}$ et que $h_2^p = 2^{-14}$. Or $2^{-14} = h_2 = \frac{h_1}{2}$ donc $2^{-14} = \frac{2^{-12}}{2^p}$, ainsi $p = 2$.
On remarque bien que l'évolution de l'erreur dépend de l'ordre du schéma imposé.

Les expérimentations que nous faisons ne donnent pas l'ordre exactement. On tient un raisonnement simplifié où on fait l'hypothèse qu'on peut négliger des constantes, qu'elles ont la même valeur d'une itération sur l'autre.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_nbbits_pas_fixe.f
pb_exponentielle.f sch_Heun.f

# NBSTEPS   NB BITS EXACTS (APPROX.)
#       4   10.872570594961713
#       8   13.728975776340011
#      16   16.657028387770914
#      32   19.621011924870196
#      64   22.602991595294135
#     128   25.593978174875300
#     256   28.589470211275099
#     512   31.587216676844790
#    1024   34.586026360047242
#    2048   37.586269512263435
#    4096   40.594072100459627
#    8192   43.482693108820882
#   16384   47.398300921530513
#   32768   45.888106189211328
```

Avec le schéma de Heun on remarque que l'évolution du nombre de bits exacts est de 3. Cela correspond à l'ordre du schéma et vérifie la théorie. Notre programme principal fonctionne avec un schéma d'ordre 3.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_nbbits_pas_fixe.f
pb_exponentielle.f sch_Runge.f

# NBSTEPS   NB BITS EXACTS (APPROX.)
#       4   6.8584320694052145
#       8   8.7212864917120907
#      16   10.652902927432585
#      32   12.618860307260315
#      64   14.601891029858672
#     128   16.593421378239558
#     256   18.589190555576408
#     512   20.587076175334616
#    1024   22.586019232613165
#    2048   24.585490815809575
#    4096   26.585226740035896
#    8192   28.585093074143931
#   16384   30.585051503325406
#   32768   32.584972161435047
```

On observe une évolution de 2 bits lorsque le nombre de pas double. Ainsi notre programme principal fonctionne pour un schéma d'ordre 2.

Les tests sont réalisés avec le problème de l'exponentielle. Les tests avec le problème de l'oscillateur harmonique nous ont donné des résultats similaires. Dans le cas du problème d'Arenstorf, les observations sont quelques peu différentes.

```
$ ~/cnum/gfor -Wall -fimplicit-none -Wno-unused-dummy-argument -fdefault-real-8
main_nbbits_pas_fixe.f pb_Arenstorf.f sch_Runge.f

#     NBSTEPS   NB BITS EXACTS (APPROX.)
#           4   -23.899264230768381833504266936447419 5
#           8   -22.586216892916411032646533125040054 1
#          16   -14.887337589325848834519538227154412 6
#          32   -10.130663095060550828108746160470529 3
#          64   -7.552745939940215752677848021782299 99
#         128   -4.881791972966446339827128090555078 44
#         256   -10.232957029363402717708678994862474 9
#         512   -4.997106567483815155713295389194476 93
#        1024   -7.436820270415866596075251365127569 19
#        2048   -2.648579124066103865140808495757865 66
#        4096    1.403543251305157155673662416973694 88
#        8192    0.434776718139774552273436699196960 40
#       16384    2.126004949620538876733084999871042 19
#       32768    4.644885538801566747970510320002969 14
#       65536    7.122904858750338975174211311185433 08
#      131072    9.532844117278822904826399727244450 44
#      262144    11.816343168277278774501472678435010 8
#      524288    13.978447825709930232119255766531232 2
#     1048576    16.064791517840544017930618780028953 9
#     2097152    18.109336877684934369111342781191185 1
#     4194304    20.131960853785445735013727436031187 5
#     8388608    22.143361751310954554258024731272385 5
```

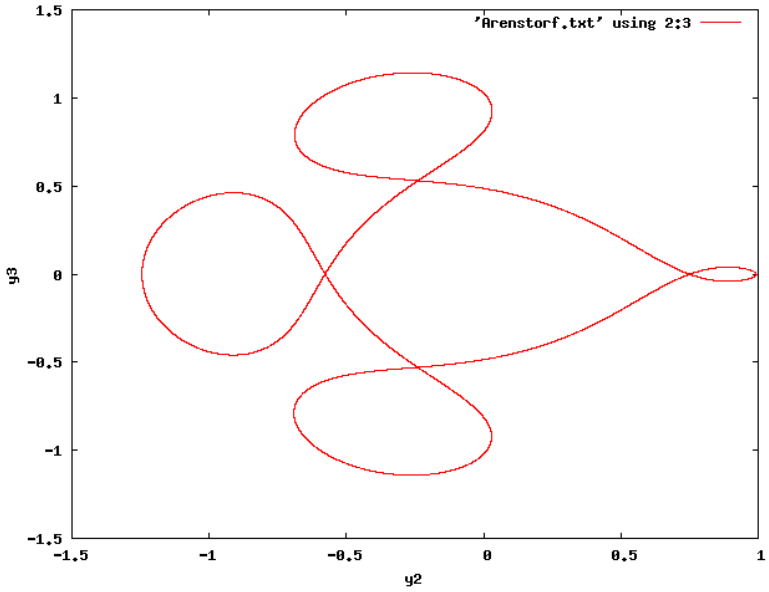
Sur un problème difficile comme Arenstorf, avec un schéma rudimentaire comme Runge, l'hypothèse d'une évolution du nombre de bits exacts égal à l'ordre du schéma n'est pas forcément réalisée pour des pas de grande taille. Il faut donc prendre en compte l'hypothèse à partir d'un certain seuil de bits exacts. On observe qu'à partir de 9 bits exacts, l'hypothèse est vraie.

2.2.5 Problème d'Arenstorf

Nous avons tracé le graphique correspondant aux valeurs calculées par un intégrateur sur le problème d'Arenstorf. Nous avons utilisé l'outil Gnuplot et le fichier `plot_2D_Arenstorf.gp`.

```
$ gfortran -Wall -fimplicit-none -fdefault-real-8 main_nbbits_pas_fixe.f
pb_Arenstorf.f sch_Heun.f
$ ./a.out > Arenstorf.txt
$ gnuplot plot_2D_Arenstorf.gp
```

Le résultat s'affiche dans le fichier `courbe_Arenstorf.png`. Voici le résultat obtenu :



2.3 Nouveau schéma, problème et questions optionnelles

2.3.1 Schéma RK4

Avec le même principe que le schéma de Heun, nous avons implémenté le schéma RK4 en ajoutant et en modifiant les coefficients. Ce schéma est d'ordre 4. Nous avons effectué les mêmes tests qu'avec le schéma de Heun, c'est-à-dire, utiliser le programme principal `main_nbbits_pas_fixe.f` pour observer l'évolution du nombre de bits exacts lorsque l'on double de nombre de pas.

```
$ ~/cnum/gfor -Wall -fimplicit-none -Wno-unused-dummy-argument -fdefault-real-8
main_nbbits_pas_fixe.f pb_exponentielle.f sch_RK4.f

#     NBSTEPS   NB BITS EXACTS (APPROX.)
#           4   15.206559307803412
#           8   19.056947294248221
#          16   22.981974929210772
#          32   26.944446645043122
#          64   30.925675941575093
#         128   34.916340995529630
#         256   38.910826265711414
#         512   42.857732540167810
#        1024   46.511957703326075
#        2048   45.770269698917467
#        4096   46.067655609542044
#        8192   45.872839432558017
#       16384   47.742255322747873
#       32768   50.442695040888964
#       65536   46.334170584110794
#      131072   48.050377618110211
```

Avec le fichier `pb_exponentielle.f` on remarque bien une évolution de 4 bits à chaque tour. Pour l'exponentielle le schéma permet bien de converger vers le bon résultat avec une bonne précision.

Concernant le problème de l'oscillateur harmonique:

```
$ ~/cnum/gfor -Wall -fimplicit-none -Wno-unused-dummy-argument -fdefault-real-8
main_nbbits_pas_fixe.f pb_oscillateur_harmonique.f sch_RK4.f

#     NBSTEPS   NB BITS EXACTS (APPROX.)
#           4   -9.05023684662768640E-002
#           8   2.4283758395827038
#          16   6.3071031407634965
#          32   10.298790443958143
#          64   14.297604107861492
#         128   18.297337620681112
#         256   22.297271962682693
#         512   26.297255502371023
#        1024   30.297256909709329
#        2048   34.297241048969468
#        4096   38.299859775994989
#        8192   42.255669993338564
#       16384   45.370183530402407
```

On remarque encore une fois une augmentation du nombre de bits exacts de 4. A partir de 42 bits, il est normal d'obtenir une augmentation plus faible, car on ne peut pas augmenter la précision à l'infini.

A présent, nous passons au test de notre programme avec le problème d'Arenstorf.

```
$ ~/cnum/gfor -Wall -Wno-unused-dummy-argument -fimplicit-none -fdefault-real-8
main_nbbits_pas_fixe.f pb_Arenstorf.f sch_RK4.f

#     NBSTEPS   NB BITS EXACTS (APPROX.)
#           4   -26.204461405916792
#           8   -8.0026760851528973
#          16   -9.4979385469914543
#          32   -8.8749122348308358
#          64   -7.8972608258056107
#         128   -6.8359348062201137
#         256   -5.8065500818928042
#         512   -4.5462908528056483
#        1024   -1.2067763339675113
#        2048    1.5703049791722556
#        4096    6.66996442212437202E-002
#        8192    3.6693631165285243
#       16384    7.3232091260867307
#       32768    11.539833097680454
#       65536    15.661761338679993
```

Ici nous observons le même problème d'évolution que tout à l'heure, cependant on remarque qu'en poussant les calculs plus loin, le nombre de bits exactes est bien de 4. L'erreur est également très faible.

Après avoir testé avec différents problèmes de dimensions différentes, nous observons une évolution du nombre de bits exacts correcte. On peut conclure que notre programme du schéma RK4 est correct.

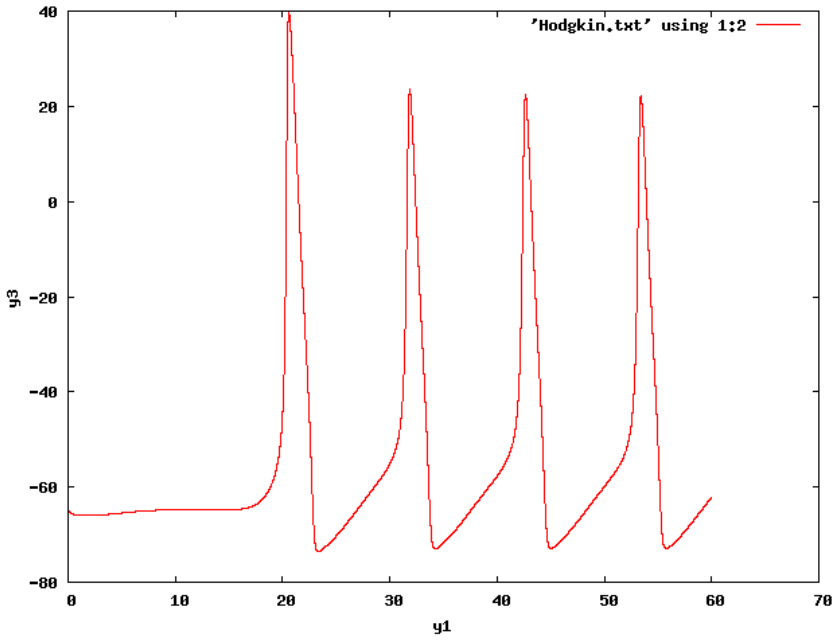
2.3.2 Le problème Hodgkin-Huxley

Le problème de Hodgkin-Huxley est une modélisation mathématique permettant d'interpréter les potentiels d'actions d'une migration d'ions sodium Na+ et d'ions potassium K+ à travers la membrane d'un neurone. Ce problème est de dimension $N = 4$.

Contrairement aux problèmes précédents, celui de Hodgkin-Huxley n'a pas de solution exacte connue. Un moyen d'estimer la justesse de notre résultat est de simuler le même modèle avec un autre logiciel et de vérifier s'ils donnent la même valeur.

Pour savoir si notre programme est potentiellement correct, nous avons observé nos résultats graphiquement avec `gnuplot` dans le fichier `courbe_hodgkin.png`.

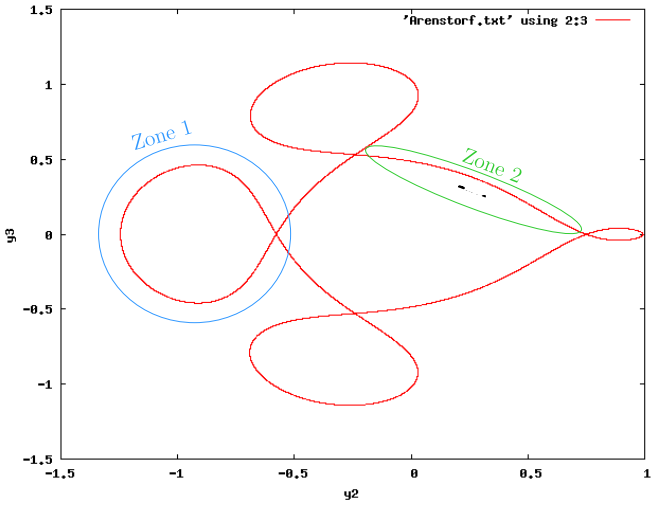
```
$ ~/cnum/gfor -Wall -Wno-unused-dummy-argument -fimplicit-none -fdefault-real-8
main_pas_fixe2.f pb_Hodgkin_Huxley.f sch_RK4.f
$ ./a.out > hodgkin.txt
$ gnuplot plot_2D_Hodgkin.gp
$ display courbe_hodgkin.png
```



Le résultat obtenu est similaire au graphique observé dans le sujet `Hodgkin-Huxley.pdf` donner pour résoudre ce problème.

2.3.3 Question Optionnelle : pas adaptatif

Dans cette question optionnelle, nous avons transformé notre programme principal pour ne plus avoir un nombre de pas fixe, mais un nombre de pas qui dépend de la précision du résultat obtenu à chacun d'entre eux. Si pour un pas donner, l'erreur (relative) est jugé trop grande, alors celui-ci est réduit et on calcul à nouveau l'erreur. Ce processus se répète jusqu'à ce que l'erreur soit suffisamment petite. A l'inverse, on augmente le pas si on estime que l'erreur reste suffisamment petite. Ces changements de pas peuvent se justifier graphiquement, en prenant l'exemple du problème d'Arenstorf.



Lorsque nous sommes dans une zone de courbe (zone 1) le pas aura tendance à diminuer pour faire un arrondi correct et obtenir une erreur faible. Dans une zone où le tracé ressemble plus linéaire (zone 2), le pas peut augmenter sans avoir une erreur trop grande.

Quels avantages peut-on en tirer par rapport au pas fixe?
Si on fixe le pas on peut se retrouver dans deux cas :

- Il est trop grand et les résultats ne sont pas assez précis
- Il est trop petit et malgré la précision du résultat, il y a trop de calcul

On peut l'observer aussi graphiquement avec le problème d'Arenstorf. En effet, avec le programme `main_pas_fixe2.f`, il y a une zone du graphique où il manque un tracé si le nombre de pas est fixé à 25 000. Dans le cas où nous le fixons à 40 000, cette zone est bien tracée.

Si nous prenons le schéma RK4 pour résoudre le problème d'Arenstorf avec le fichier `main_pas_fixe2.f` et un nombre de pas de 25 000, nous obtenons $4 \times 25\,000 = 100\,000$ appels à la fonction de calcul de y_1 et $\hat{y}_1$. Dans notre problème de pas adaptatif nous avons calculé le nombre de pas utilisés grâce à l'ajout d'une variable. Le résultat obtenu avec Arenstorf et RK4 est de 4 386, soit $4 \times 4\,386 = 17\,544$ appels à la fonction. Cela représente 5,7 fois moins d'appel à la fonction et le résultat est plus précis. Nous passons d'une erreur de l'ordre de 10^{-4} à un ordre de 10^{-9}. Bien sûr, il faut prendre en compte les calculs d'erreur absolue qui permettent de déterminer le pas h et les calculs du nouveau pas. Pour les mêmes conditions que précédemment, le nombre de calculs de l'erreur et du nouveau pas est de 8 778. Cela reste très faible en termes de calcul par rapport au pas fixe.

Le pas adaptatif permet donc d'obtenir un résultat plus précis avec moins de d'appel à la fonction, ce qui rend le temps d'exécution plus court.

Vérification de nos résultats. Pour vérifier que notre programme est correct nous avons regardé l'erreur obtenue après exécution.

```
$ ~/cnum/gfor -Wall -Wno-unused-dummy-argument -fimplicit-none -fdefault-real-8
main_pas_adaptatif.f pb_Arenstorf.f sch_RK4.f

Nombre de pas                       4386
Nombre de calculs erreur+Hnew       8778
# Erreur relative globale =         6.51869469459231482E-009
```

Conclusion

Le but de ce projet est de programmer quelques méthodes d'intégrations numériques en FORTRAN. Ce projet permet de mettre en pratique des notions mathématiques assez complexes. Nous pouvons voir qu'en fonction du problème d'intégration, certains schémas sont plus adaptés que d'autres.

Nous avons réussi la première partie dans son intégralité. La deuxième partie n'a pas été réalisée entièrement, la question facultative concernant la sortie dense n'a pas été programmé car l'intégrateur à pas adaptatif et le modèle de Hodgkin-Huxley nous a pris beaucoup de temps. Chaque programme donne les résultats attendus.

Pour conclure, ce projet n'a pas été sans difficultés mais le rendu a été possible grâce à la forte implication des deux membres du binôme durant les heures de cours à Polytech'Lille et en dehors des heures prévues. Le travail en groupe est important sur ce type de projet car chacun apporte sa propre vision des choses et cela permet d'avancer plus vite dans la détection d'erreur par exemple.