Accélération des méthodes de parallélisation en temps par approches de type quasi-Newton

Hadrien Batmalle, Pierre-Elliott Bécue, Vincent Le Gallic
Stage encadré par Florian De Vuyst
[prenom].[nom]@ens-cachan.fr

30 juin 2011

Résumé

Pour répondre aux exigences croissantes en calcul numérique, nous étudions des méthodes parallèles en temps de résolution d'équations différentielles. Au cours de ce stage, nous avons essayé de mettre en place une variante de l'algorithme pararéel, inspirée des méthodes de type Quasi-Newton dans le but d'accélérer cette parallélisation. D'autre part, nous avons tenté de déployer une structure de calcul parallèle au département de mathématiques.

Introduction

Pour répondre aux problèmes posés par la biologie, la physique, ou d'autres sciences, nous sommes régulièrement amenés à faire de la simulation numérique sur des modèles associés à des situations précises qui nous sont imposées par ces sciences, par exemple, l'étude de la trajectoire d'astéroïdes, de l'évolution du climat, ou des cycles stellaires...

En analyse numérique, nous sommes souvent confrontés au besoin d'effectuer des résolutions d'équations plus ou moins complexes par des méthodes approchées convergeant le plus rapidement possible, et dont l'exécution prenne un temps relativement court. Le développement des technologies informatiques connaissant actuellement ses limites en termes de miniaturisation, la taille des processeurs ne peut plus diminuer drastiquement dans un futur proche, et leur rapidité risque donc de stagner. Il devient dès lors impossible pour une unité de calcul seule de résoudre dans un délai court une équation, dès que celle-ci fait appel à des méthodes coûteuses. Des procédés d'optimisation peuvent permettre un gain de temps, mais celui-ci devient rapidement négligeable. Il faut dès lors penser à de nouvelles méthodes pour « gagner du temps ». Une méthode simple dans le cas de programmes exécutant des tâches indépendantes est de distribuer ces tâches sur plusieurs unités de calcul, ce qui est facilement fait à l'aide d'ordinateurs à processeurs multi-cœurs.

Il arrive cependant qu'un programme soit à « mémoire temporelle », c'est-à-dire que pour s'exécuter, une tâche ait besoin que la précédente ait été exécutée, et ainsi de suite. Ces processus ne peuvent pas être découpés temporellement, car alors il faudrait attendre qu'une unité de calcul ait exécuté une portion de tâche pour que la suivante puisse effectuer la portion suivante, et aucun gain de temps ne pourrait être constaté.

Cependant, il existe des méthodes de prédiction-correction, qui permettent à moindre coût de donner suffisamment d'informations aux unités pour qu'elles effectuent leurs tâches sans se préoccuper des autres machines, quitte à itérer plusieurs fois lesdites tâches en tenant compte des résultats des itérations précédentes pour affiner leurs calculs.

Notre stage a porté sur une de ces méthodes, appelée méthode pararéelle, et effectuant ce genre d'approximations de prédiction, qui si elles font perdre un peu de précision par rapport à un algorithme de résolution type RK4, peuvent diviser par deux le temps de calcul total (il faut cependant tenir compte du fait que le calcul a impliqué plus d'une machine).

Chapitre 1

L'algorithme pararéel

1.1 Cadre

On se donne un système différentiel de la forme

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t} = f(t,u) \\ u(0) = u_0 \end{cases}$$

où $u : \mathbb{R} \longrightarrow \mathbb{R}^p$, f vérifiant les hypothèses du théorème de Cauchy-Lipschitz, et éventuellement d'autres hypothèses qui seront fournies si nécessaire.

On cherche à le résoudre sur l'intervalle $[0,T]$ en parallélisant le calcul. Pour cela on découpe l'intervalle en $[0,T_1],[T_1,T_2],\ldots,[T_{N-1},T_N]$ ($T_N = T$) et on cherche à résoudre le problème en parallèle sur chacun des intervalles.

1.2 1ère approche

On désire réaliser une méthode de résolution de type Runge-Kutta, précise et efficace. Cependant, ces méthodes sont coûteuses en temps de calcul.

On se donne un solveur fin $\mathcal{F}$ et un solveur grossier $\mathcal{G}$. $u_1,\ldots,u_N$ ont pour but d'être les valeurs de U en $T_1,\ldots,T_N$.

On cherche à avoir

$$R(u_1,\ldots,u_N) = \begin{pmatrix} u_1 - \mathcal{F}(u_0) \\ u_2 - \mathcal{F}(u_1) \\ \vdots \\ u_N - \mathcal{F}(u_{N-1}) \end{pmatrix} = 0$$

La méthode de Newton nous donne (en notant U le vecteur $\begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}$) :

$R(U^{k+1}) = R(U^k) + \mathrm{D}R(U^k)(U^{k+1} - U^k) = 0$. Or,

$$\mathrm{D}R(U^k) = \begin{pmatrix} I_p & 0 & \ldots & \ldots & 0 \\ -\mathrm{D}\mathcal{F}(u_1^k) & I_p & & & \vdots \\ 0 & -\mathrm{D}\mathcal{F}(u_2^k) & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & -\mathrm{D}\mathcal{F}(u_{N-1}^k) & I_p \end{pmatrix}$$

En regardant la ligne $n+1$, on obtient : $u_{n+1}^k - \mathcal{F}(u_n^k) - \mathrm{D}\mathcal{F}(u_n^k)(u_n^{k+1} - u_n^k) + (u_{n+1}^{k+1} - u_{n+1}^k) = 0$

En simplifiant les u_{n+1}^k et en approximant $\mathrm{D}\mathcal{F}$ par $\mathrm{D}\mathcal{G}$: $u_{n+1}^{k+1} = \mathcal{F}(u_n^k) - \mathrm{D}\mathcal{G}(u_n^k)(u_n^{k+1} - u_n^k)$ et d'après la formule de Taylor au premier ordre, on obtient

$$u_{n+1}^{k+1} = \mathcal{G}(u_n^{k+1}) + \mathcal{F}(u_n^k) - \mathcal{G}(u_n^k)$$

L'idée de l'algorithme est donc la suivante :

– Utiliser une méthode rapide pour approcher les valeurs u_i de u aux points T_i.

– Utiliser une méthode plus précise (RK) pour déterminer u sur chacun des intervalles en utilisant u_i comme valeur initiale.
– Déterminer l'écart entre u_i et la « valeur réelle » et itérer le calcul.

L'intérêt de cette métode est notable : le calcul des u_{n+1}^{k+1} (pour $n \in [\![1, N]\!]$, k fixé) nécessite de connaître certains éléments de la même itération (les $\mathcal{G}(u_n^{k+1})$), en revanche le calcul des $\mathcal{F}(u_n^k)$ ne nécessite que la connaissance des informations à l'itération précédente (au rang k). On peut donc calculer les $\mathcal{F}(u_n^k)$ *en parallèle.* De plus, la partie qui est obligatoirement séquentielle porte sur le solveur grossier, donc plus rapide à calculer.

1.3 Le solveur grossier

On cherche à avoir un solveur grossier ($\mathcal{G}$). Une première approche est de considérer qu'une convergence d'ordre un ou deux suffit, et d'utiliser par exemple la méthode d'Euler ou de Heun.

1.3.1 Méthode d'Euler

La méthode d'Euler consiste à utiliser la formule de Taylor au rang 1.

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t} = f(t, u) \\ u(0) = u_0 \end{cases}$$

On écrit $u(t + h) \approx u(t) + h.f(t, u(t))$.
Pour l'exemple, on considèrera que les intervalles $[T_n, T_{n+1}]$ ont tous la même taille $\frac{T}{N}$. On a alors :

$$\begin{cases} u_0 = u(T_0 = 0) \\ u(T_1 = T_0 + \frac{T}{N}) = u(T_0) + \frac{T}{N}.f(T_1, u(T_0)) = u_0 + \frac{T}{N}.f(T_0, u_0) \\ \quad \vdots \\ u_N = u(T_N) = u_{N-1} + \frac{T}{N}.f(T_{N-1}, u_{N-1}) \end{cases}$$

La méthode d'Euler est d'ordre de convergence 1 en $h = \frac{T}{N}$.

1.3.2 Méthode de Heun

Elle consiste à calculer plusieurs points et à en faire une moyenne.
$h = \frac{T}{N}$

$$\begin{cases} u_0 = u(T_0) \\ \quad \vdots \\ p_1^{n+1} = f(T_n, u_n), \quad p_2^{n+1} = f(T_n + h, u_n + h \cdots p_1^{n+1}) \\ u_{n+1} = u_n + h\frac{p_1^{n+1} + p_2^{n+1}}{2} \\ \quad \vdots \end{cases}$$

Les méthodes d'Euler et de Heun sont rapides, et sont les premières méthodes qui ont été utilisées durant ce stage, mais leur précision laisse à désirer sur des systèmes plus oscillants, c'est pour cela que par la suite, nous utiliserons une méthode de Runge-Kutta, celle d'ordre 4.

1.4 Le solveur fin

Le solveur fin $\mathcal{F}$ doit être au moins aussi précis, car il doit calculer une solution le plus approchant possible de la valeur réelle. On utilise donc des méthodes avec des convergence d'ordre 4 typiquement.

1.4.1 La méthode de Runge-Kutta

Nous utiliserons la méthode de Runge-Kutta d'ordre 4 comme solveur fin, avec un pas défini sur chaque sous-intervalle de travail. Les méthodes de Runge-Kutta sont rappelées en annexe.

1.5 L'algorithme pararéel proprement dit

Nous avons vu dans la présentation rapide de l'algorithme les étapes clés de son fonctionnement. Nous allons ici présenter l'algorithme dans son ensemble.

1.5.1 Première étape

Une fois l'équation différentielle ordinaire posée, et la condition initiale donnée, on applique le solveur grossier avec un pas $\Delta T = T/N$, pour obtenir les points u_n^0. Le graphique ci-après présente les points obtenus avec le solveur grossier dans le cas de l'équation (sin) : $f(t,(x,y)) = (y,-2x)$, avec comme condition initiale $(0,1)$.

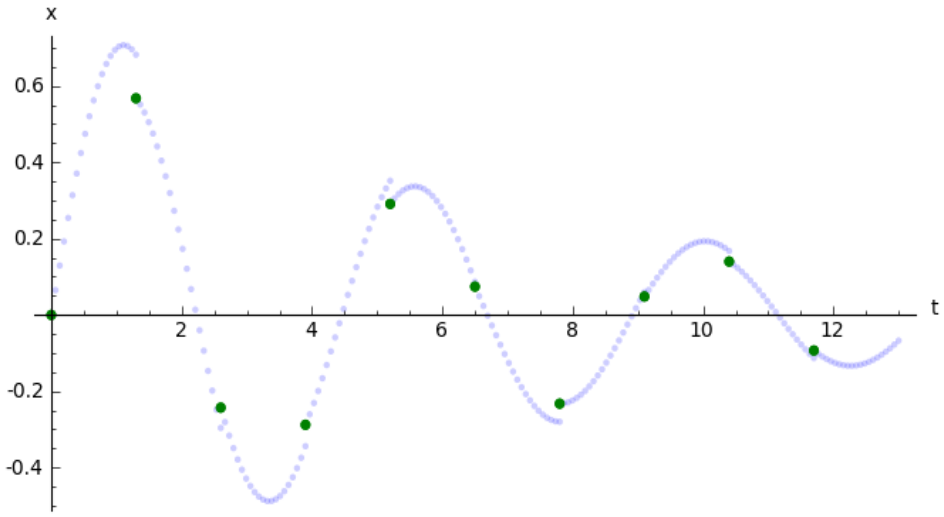

FIGURE 1.1 – La première étape, avec les points du solveur grossier (vert), les points bleu-pâle servent à avoir un aperçu de la version fine pour situer les points grossiers.

Une fois ces premiers points obtenus, on applique le solveur fin sur chaque intervalle, avec un pas $T/(Np)$, où $p \in \mathbb{N}\backslash\{0,1\}$, on obtient ainsi la première itération de notre algorithme pararéel. Voici un graphique présentant cette résolution fine.

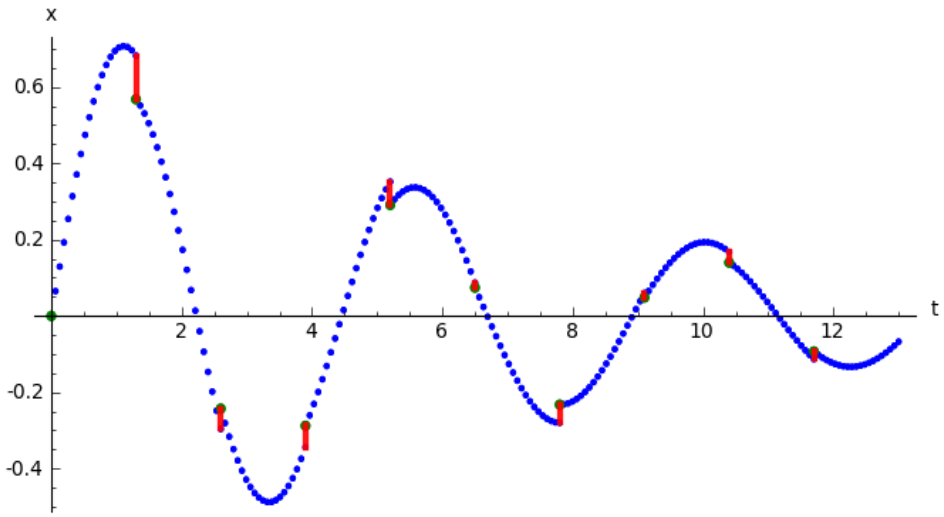

FIGURE 1.2 – La première étape.

1.5.2 Les étapes suivantes

Soit k un entier supérieur ou égal à 1, on suppose les u_n^{k-1} calculés, ainsi que les résultats de l'algorithme fin et grossier sur ces points, que nous noterons respectivement $\mathcal{F}(u_n^{k-1})$, et $\mathcal{G}(u_n^{k-1})$. On initialise u_0^k à u_0 (la condition initiale), et on calcule alors u_{n+1}^k à l'aide de la formule : $u_{n+1}^k = \mathcal{G}(u_n^k) + \mathcal{F}(u_n^{k-1}) - \mathcal{G}(u_n^{k-1})$. Cela nécessite le calcul des $\mathcal{G}(u_n^k)$ à chaque étape. Il ne reste qu'à calculer le solveur fin sur les u_n^k, $\mathcal{F}(u_n^k)$, et on obtient les résultats de l'itération k, qui sont également les données nécessaires à l'itération $k+1$.

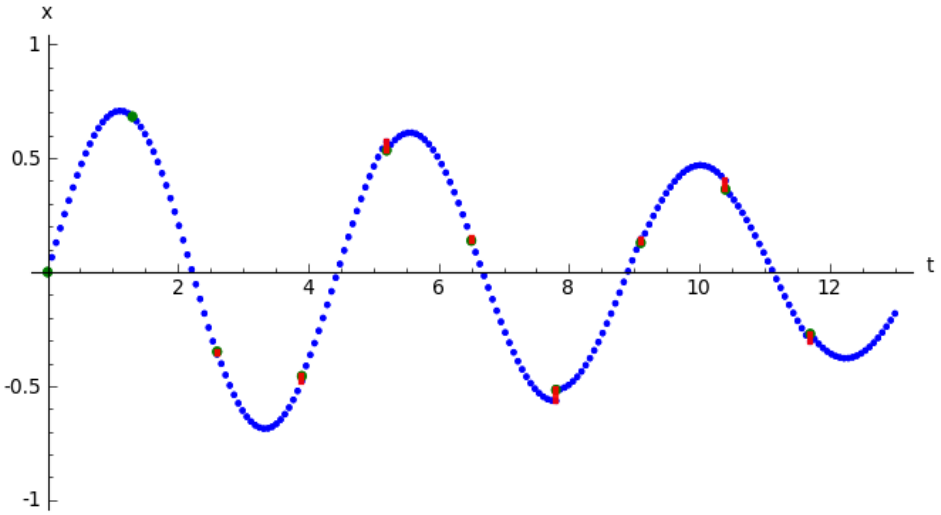

FIGURE 1.3 – Étape 2

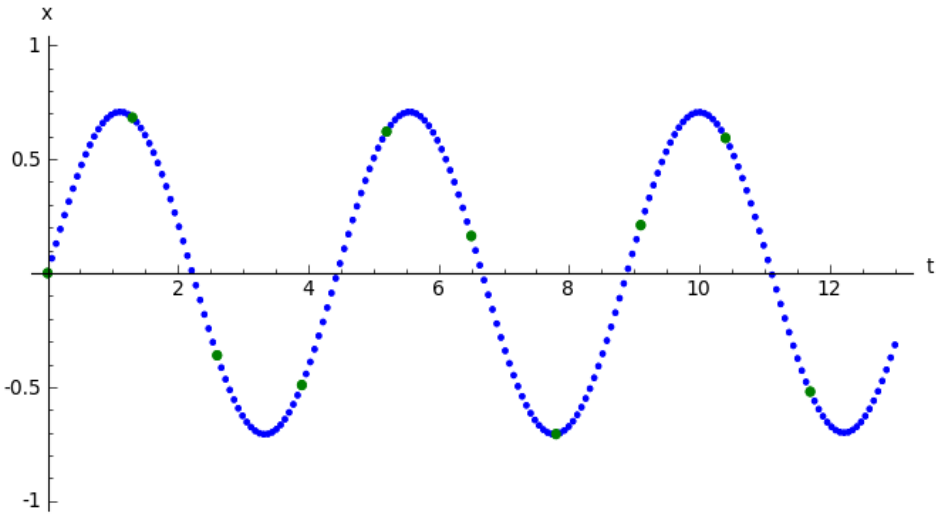

FIGURE 1.4 – Étape 5

1.5.3 L'algorithme en pseudo-code

On suppose les résolutions à l'aide du solveur fin et du solveur grossier connues.

```
Uold=matrice[N][dim]
Uold=0
Unew=Uold
U=Uold
Ufin=matrice[N][pas+1][dim]
H=T/N
h=H/pas
Uold=Grossier(u0,H)
Ufin=Fin(Uold,h,N)
Afficher(Ufin)
continuer=1
iteration=0
tant que continuer=1 faire
    iteration=iteration+1
    pour j allant de 2 à N faire
        U[j] = Unew[j-1]-Uold[j]+Ufin[j]
        si j < N faire
            Unew[j]=Grossierunpas(U[j],H)
        fin si
    fait
    Ufin=Fin(Unew,h,N)
    Uold=Unew
    Afficher(Ufin)
    demander(continuer)
fait
```

1.6 Convergence de l'algorithme pararéel

Il existe plusieurs théorèmes de convergence pour l'algorithme pararéel, les hypothèses variant d'un à l'autre. Nous traiterons ici d'un théorème qui figure dans l'article de Gander & Hairer (2006).

Considérons un algorithme fin et un algorithme grossier satisfaisant les conditions suviantes :

$$\mathcal{F}(T_n, T_{n-1}, x) - \mathcal{G}(T_n, T_{n-1}, x) = c_{p+1}(x)\Delta T^{p+1} + c_{p+2}(x)\Delta T^{p+2} + ...$$

où $\mathcal{G}(T_n, T_{n-1}, x)$ est la résolution fine sur l'intervalle $[T_{n-1}, T_n]$ partant du point x, et $\mathcal{F}(T_n, T_{n-1}, x)$ la même chose pour l'algorithme fin, avec les c_k continûment différentiables. Si le solveur grossier est une méthode de Runge Kutta, cette condition peut être vérifiée. On suppose de plus que $\mathcal{G}$ vérifie la condition suivante :

$$|\mathcal{G}(t + \Delta T, t, x) - \mathcal{G}(t + \Delta T, t, y)| \leqslant (1 + C_2\Delta T)||x - y||$$

enfin, on suppose que $\mathcal{G}(T_n, T_{n-1}, u_n^k)$ a une erreur bornée par $C_3\Delta T^{p+1}$. Alors à l'itération k de l'algorithme pararéel, on aura la majoration suivante :

$$||u(T_n) - u_n^{k+1}|| \leqslant \frac{C_3}{C_1}\frac{(C_1\Delta T^{p+1})^{k+1}}{(k+1)!}(1 + C_2\Delta T)^{n-k-1}\prod_{j=0}^{k}(n-j) \leqslant \frac{C_3}{C_1}\frac{(C_1 T_n)^{k+1}}{(k+1)!}\mathrm{e}^{C_2(T_n - T_{k+1})}\Delta T^{p(k+1)}$$

Voici une preuve de ce théorème. Comme $u(T_n) = \mathcal{F}(T_n, T_{n-1}, u(T_{n-1}))$, et comme $u_n^{k+1} = \mathcal{G}(T_n, T_{n-1}, u_{n-1}^{k+1}) + \mathcal{F}(T_n, T_{n-1}, u_{n-1}^k) - \mathcal{G}(T_n, T_{n-1}, u_{n-1}^k)$,

$$\begin{aligned} u(T_n) - u_n^{k+1} &= \mathcal{F}(T_n, T_{n-1}, u(T_{n-1})) - (\mathcal{F}(T_n, T_{n-1}, u_{n-1}^k) - \mathcal{G}(T_n, T_{n-1}, u_{n-1}^k)) - \\ &\quad \mathcal{G}(T_n, T_{n-1}, u_{n-1}^{k+1}) + \mathcal{G}(T_n, T_{n-1}, u(T_{n-1})) - \mathcal{G}(T_n, T_{n-1}, u(T_{n-1})) \\ &\quad = \mathcal{F}(T_n, T_{n-1}, u(T_{n-1})) - \mathcal{G}(T_n, T_{n-1}, u(T_{n-1})) - (\mathcal{F}(T_n, T_{n-1}, u_{n-1}^k) - \mathcal{G}(T_n, T_{n-1}, u_{n-1}^k)) + \\ &\quad \mathcal{G}(T_n, T_{n-1}, u(T_{n-1})) - \mathcal{G}(T_n, T_{n-1}, u_{n-1}^{k+1}) \end{aligned}$$

On peut appliquer la première hypothèse sur les quatre premiers termes, et la seconde hypothèse sur les deux derniers termes, on tire

$$||u(T_n) - u_n^{k+1}|| \leqslant C_1 \Delta T^{p+1} ||u(T_{n-1}) - u_{n-1}^k|| + (1 + C_2 \Delta T) ||u(T_{n-1}) - U_{n-1}^{k+1}||$$

on a ainsi une suite récurrente de la forme $e_n^{k+1} = \alpha e_{n-1}^k + \beta e_{n-1}^{k+1}$, avec $e_n^0 = \gamma + \beta e_{n-1}^0$, avec $\alpha = C_1 \Delta T^{p+1}$, $\beta = 1 + C_2 \Delta T$ et $\gamma = C_3 \Delta T^{p+1}$, et $e_n^k = ||u(T_n) - u_n^k||$. En multipliant la formule de récurrence précédente et en la multipliant par x^n, puis en sommant suivant n, on obtient, en notant $p_k(x) = \sum_{n \geqslant 1} x^n e_n^k$:

$$p_{k+1}(x) = \alpha x p_k(x) + \beta x p_{k+1}(x), p_0(x) = \gamma \frac{x}{1-x} + \beta x p_0(x)$$

en résolvant suivant p_k par récurrence, on a la formule

$$p_k(x) = \gamma \alpha^k \frac{x^{k+1}}{1-x} \frac{1}{(1-\beta x)^{k+1}}$$

on remplace $(1-x)$ par $(1-\beta x)$, cela fait croître les coefficients de p_k. Or comme

$$\frac{1}{(1-\beta x)^{k+2}} = \sum_{n \geqslant 0} \mathcal{C}_{k+1+n}^n \beta^n x^n$$

on obtient donc la majoration suivante pour le coefficient e_n^k :

$$e_n^k \leqslant \gamma \alpha^k \beta^{n-k-1} \mathcal{C}_n^{k+1}$$

cela conclut la preuve.

Chapitre 2

L'implémentation de l'algorithme pararéel classique

Une fois cet algorithme en place, nous avons décidé de l'implémenter en utilisant la structure informatique du département de mathématiques de l'E.N.S. de Cachan.

2.1 Les outils

Il a été question d'interfacer Matlab et C, mais nous avons abandonné rapidement cette idée, faute de connaissances et d'envie, pour privilégier l'interface entre Python et C, qui ne nécessite que des outils présents pour la plupart nativement sous Linux, et pour les autres installables facilement. Les algorithmes ont donc été programmés en Python, pour la souplesse, et le besoin de ne pas avoir à recompiler les programmes en permanence, et éviter les débordements de mémoire intempestifs, mais les calculs ont été laissés au C, pour la rapidité d'exécution. Le moteur d'exécution est Sage (`http://sagemath.org`), qui utilise Weave pour compiler la partie C de nos programmes.

Dans un premier temps, les algorithmes ont été programmés en séquentiel, et exécutés sur nos ordinateurs, pour effectuer les repérages d'erreurs, et permettre de faire le moins de nettoyage une fois l'interface pararéelle mise en place. Cette partie, bien que fastidieuse nous a permis de faire connaissance avec les différentes erreurs inhérentes à la mise en communication de plusieurs langages de programmation.

Pour la mise en pratique de l'algorithme pararéel, nous avions besoin d'utiliser openmpi, qui permet la distribution de tâches à plusieurs unités de calcul, et il fallait donc que nos autres outils soient compatibles avec celui-ci. D'autre part, nous avons dû prendre connaissance de la documentation, et faire des tests basiques (« hello world ») avant de pouvoir commencer à espérer faire tourner notre algorithme.

Une fois cette étape franchie, nous avons tenté d'obtenir les accès suffisants aux ordinateurs du département de mathématiques pour exécuter nos programmes sans avoir besoin de nous y rendre, et pour pouvoir installer sage. Cette opération a été nettement compliquée par le caractère vieillissant du serveur d'opération du département de mathématiques (qui doit être changé l'an prochain). Il a ensuite fallu que nous trouvions comment permettre à openmpi d'établir des communications entre les différentes machines, un fichier « hosts » contenant une clé générique nous a permis de débloquer le problème, et d'obtenir vingt « Hello World ! » de vingt machines différentes en moins d'une seconde !

Cependant, pour l'heure, nos programmes pararéels classiques n'ont pas voulu fonctionner aussi bien que de simples messages textuels, nous avons passé un certain temps à chercher la cause de ces erreurs, sans succès. Si nous avions eu plus de temps, il est probable que nous aurions vraiment réussi à programmer du pararéel sur les machines a1, ..., a20. Les résultats que nous présentons ici sont des résultats portant sur des algorithmes programmés en séquentiel. Il est probable qu'ils puissent marcher en parallèle, mais comme dit précédemment, nous n'avons pas su venir à bout des pannes. Plusieurs pistes sont envisagées cependant. Une première piste serait une erreur de configuration de MPI que nous ne saurions envisager, il suffirait alors de reconfigurer openmpi, et les programmes s'exécuteraient sans problème. Ce cas nous paraît peu probable, vu qu'un hello world fonctionne. Une seconde théorie est que nous ayons mal implémenté les appels à MPI, car nos algorithmes fonctionnent quand le nombre d'ordinateurs mis en réseau pour le calcul est faible.

2.2 Le code

Sera fourni en annexe. Mais n'est à lire sous aucun prétexte.

2.3 Les résultats

2.3.1 Une exponentielle

Sur le cas de l'exponentielle, nous avons menés nos tests avec comme solveur grossier l'algorithme d'Euler, celui de Heun, puis RK4. Dans tous les cas, la résolution se passe très bien, et les résultats sont vite probants. Cela étant, la légitimité d'une utilisation de l'algorithme pararéel dans ce cas se pose.

2.3.2 Le Brusselator

L'équation fournie ici est la suivante :

$$f(t,(x,y)) = (A + x^2y - (B+1)x, Bx - x^2y)$$

avec $A = 1$, $B = 3$. Si l'on utilise une méthode de type Euler, Heun ou RK3 sur cet exemple comme solveur grossier, il faudra avoir réalisé quasiment toutes les itérations de l'algorithme pour arriver à un résultat potable. Nous utilisons donc ici RK4 comme solveur grossier. Ici, nous utilisons 32 unités de calcul, on cherche à résoudre l'équation sur $[0, 12]$, avec comme conditions initiales $[0, 1]$, le pas de résolution fine est ici $h = 12/(32 \times 20)$.

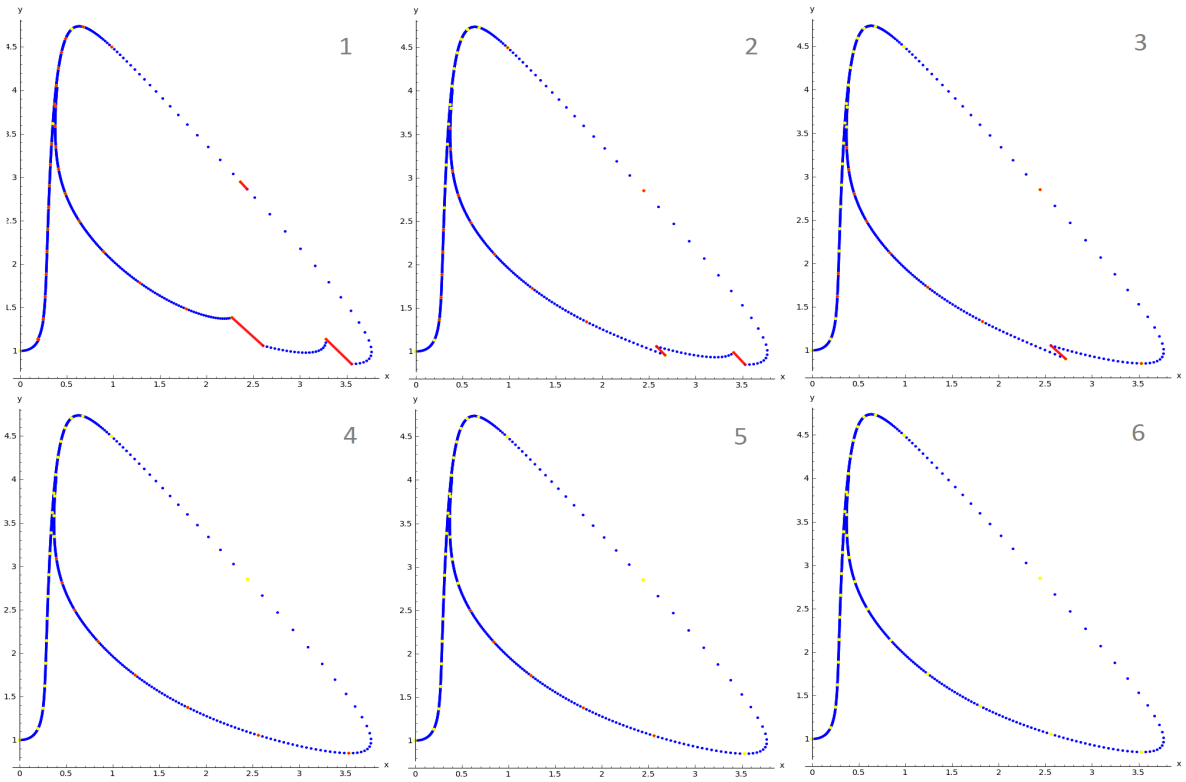

FIGURE 2.1 – Itérations 1 à 6

Nous avons ensuite voulu voir les écarts avec la résolution à l'aide de RK4, avec pour pas le pas h. Voici de même les erreurs, il faut faire attention à l'échelle, plus qu'à la forme de l'erreur.

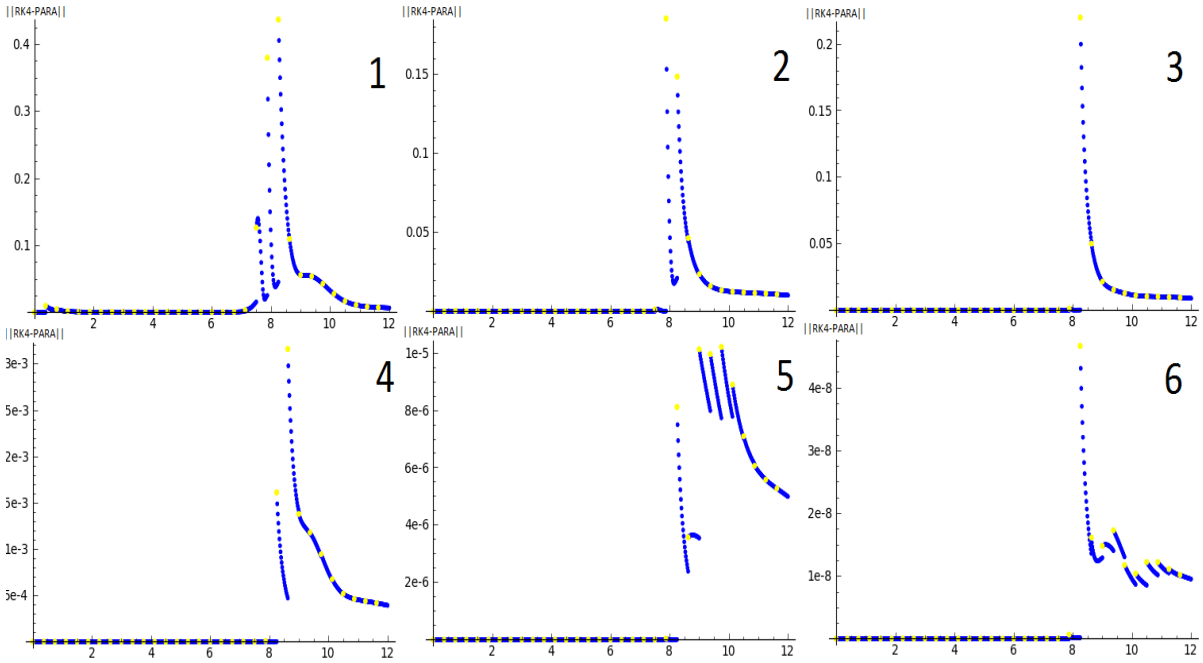

FIGURE 2.2 – Itérations 1 à 6

Ici, on remarque que l'erreur diminue rapidement pour atteindre 5×10^{-8} en 6 itérations. Nous n'avons pas été en mesure de vérifier l'ordre de grandeur des constantes citées dans le théorème de convergence de l'algorithme, mais on peut penser que leur ordre de grandeur devient rapidement négligeable devant le terme $\Delta T^{(k+1)p}$.

2.3.3 L'orbite d'Arenstorf

L'équation d'Arenstorf sert à modéliser le mouvement d'objets légers sous l'influence gravitationnelle d'objets bien plus lours, par exemple, un satellite sous l'influence de la terre et de la lune. L'équation est $f(t,(x,y,u,v)) = (u, v, x+2v-b(x+a)/D_1-a(x-b)/D_2, y-2u-by/D_1-ay/D_2)$ avec $D_1 = ((x+a)^2+y^2)^{\frac{3}{2}}$, et $D_2 = ((x-b)^2+y^2)^{\frac{3}{2}}$, $a = 0.012277471$ et $b = 1-a$, les conditions initiales étant $(0.994, 0, 0, -2.00158510637908)$. Nous résoudrons le système sur l'intervalle $[0, T = 17.06521656015796]$. Sur ce système, il n'est pas envisageable d'utiliser un solveur grossier plus faible que RK4, on travaille avec 250 unités de calcul, et le pas de résolution fine est $h = T/(250 \times 320)$.

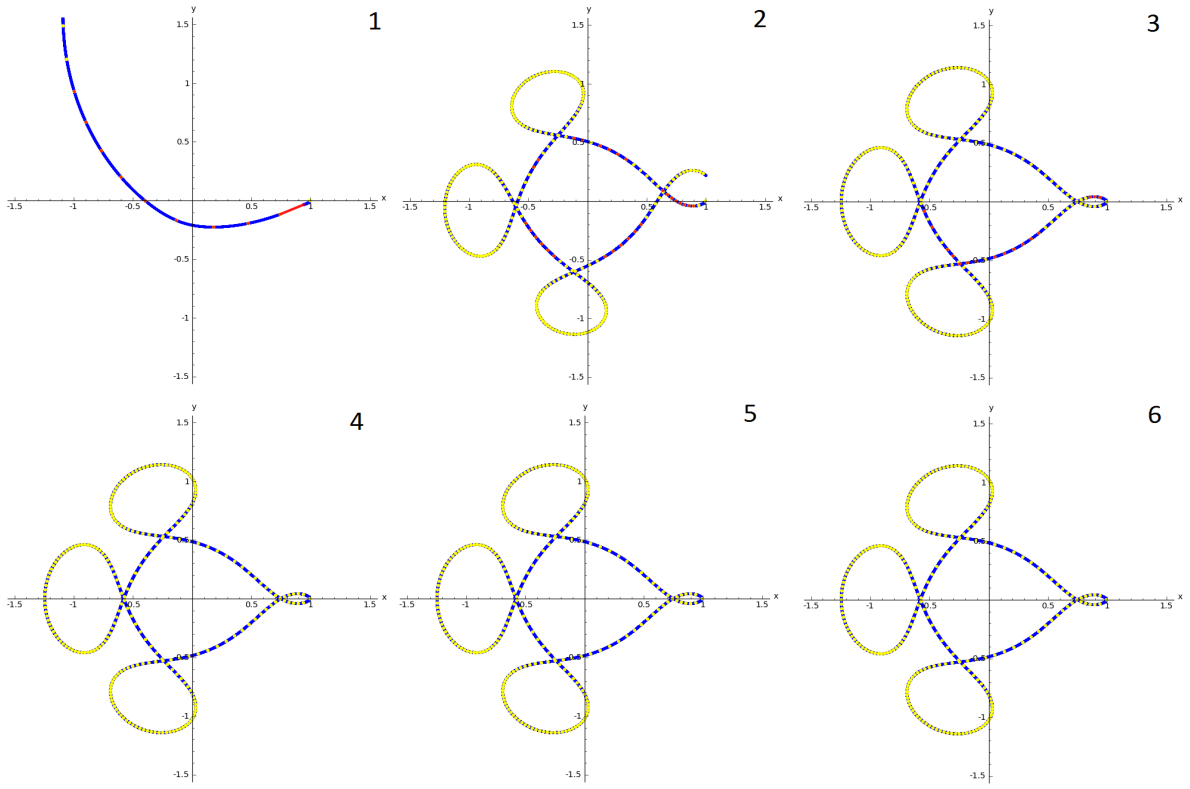

FIGURE 2.3 – Itérations 1 à 6

On constate que la première étape donne quelque chose de sensiblement différent du reste, l'échelle des graphes ayant été fixée manuellement, on ne voit pas toute la figure, mais en fait, on a ici une spirale. Voici comme précédemment l'erreur entre RK4 et l'algorithme pararéel.

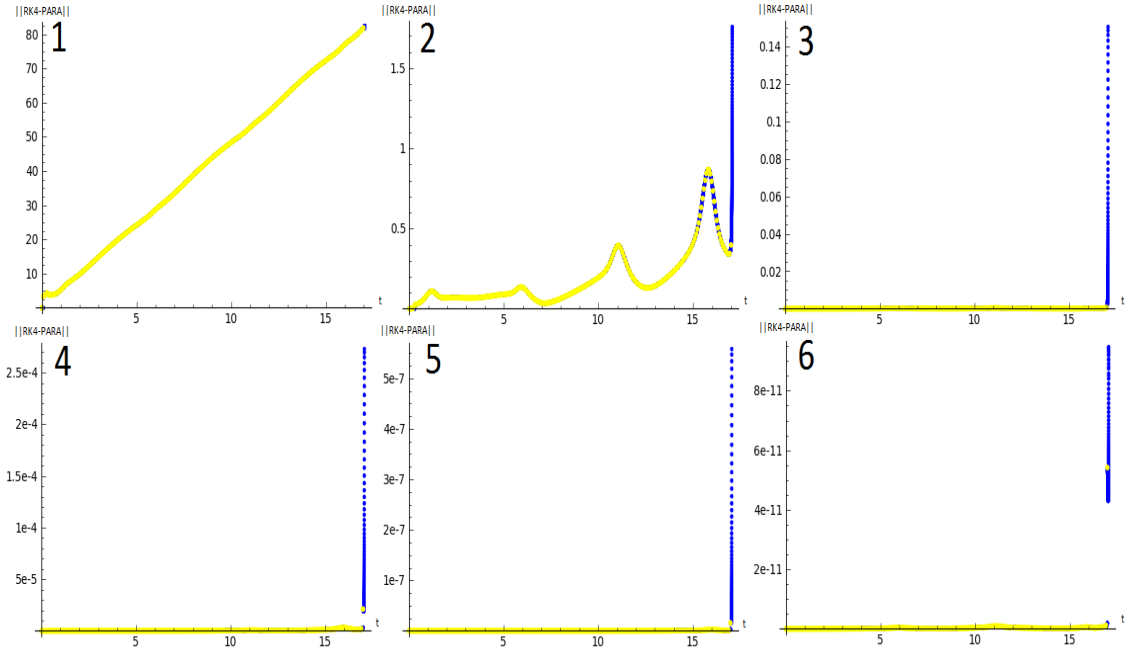

FIGURE 2.4 – Itérations 1 à 6

On constate ici comme précédemment que l'erreur décroît rapidement, puisqu'en six itérations, on arrive à une erreur majorée par 10^{-10}. Au regard du nombre d'itérations qu'aurait pris une résolution séquentielle avec RK4, le gain de temps est évident.

2.3.4 Équations de Lorenz

Les équations de Lorenz rencontrent plusieurs applications en climatologie. L'équation ici est $f(t,(x,y,z)) = (-\sigma x + \sigma y, -xz + rx - y, xy - bz)$, avec $\sigma = 10$, $r = 28$, $b = 8/3$, et avec comme conditions initiales $(5,-5,20)$ [il y a une erreur ici dans les conditions proposées par Gander & Hairer (2006), qui proposait les conditions initiales $(20,5,-5)$], on travaille sur $[0,10]$, avec 180 unités de calcul, et un pas fin de $h = 10/(180 \times 80)$.

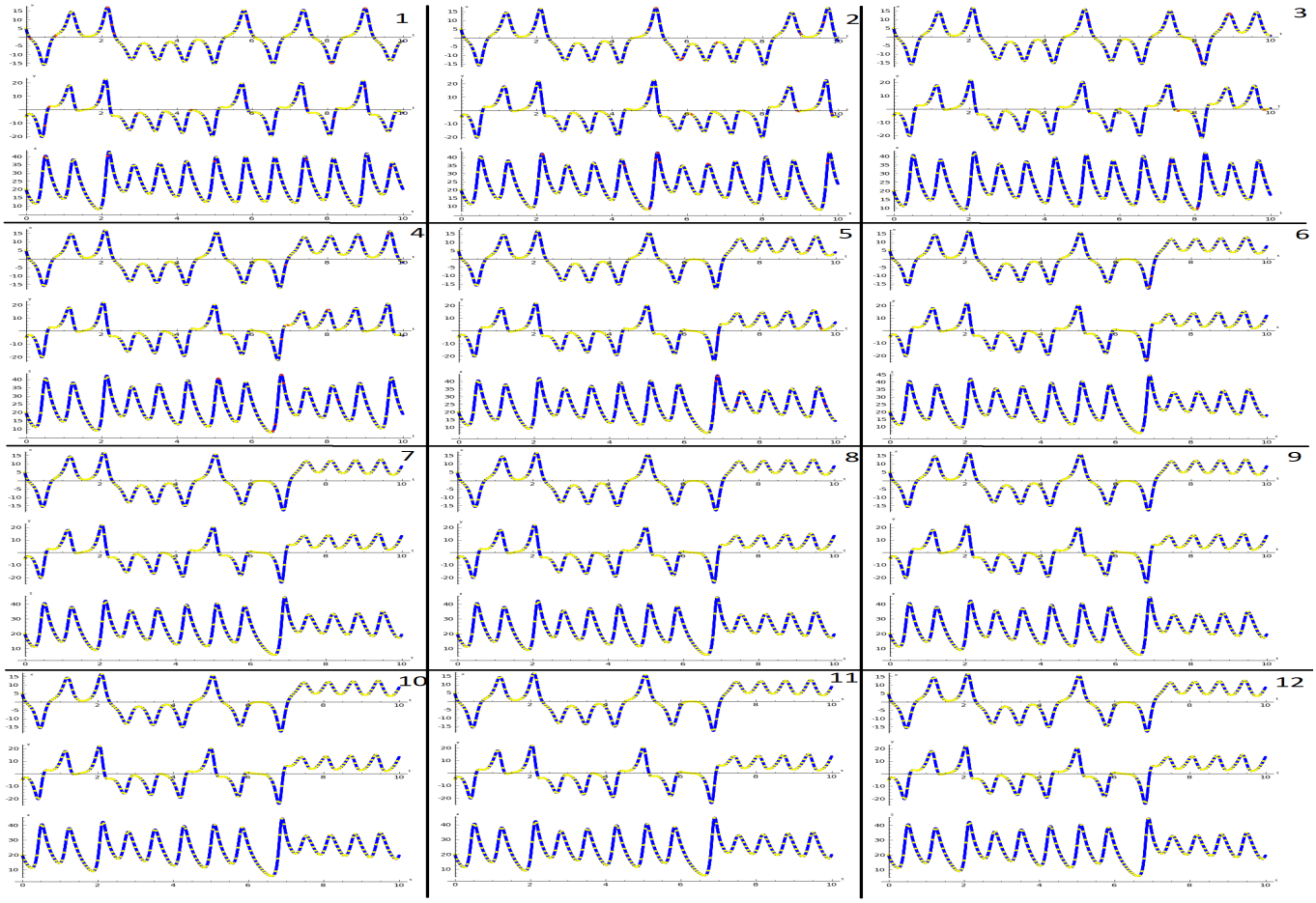

FIGURE 2.5 – Itérations 1 à 12

Sans l'évaluation des erreurs ici, on ne peut se rendre compte de la nécessité d'effectuer douze itérations. Mais comme le graphique suivant va le montrer, celle-ci reste forte jusqu'à la onzième, ce qui néecesite de continuer. Ce résultat est explicable par le fait qu'on utilise moins d'unités de calcul, avec des pas plus grands, donc la précision est moindre.

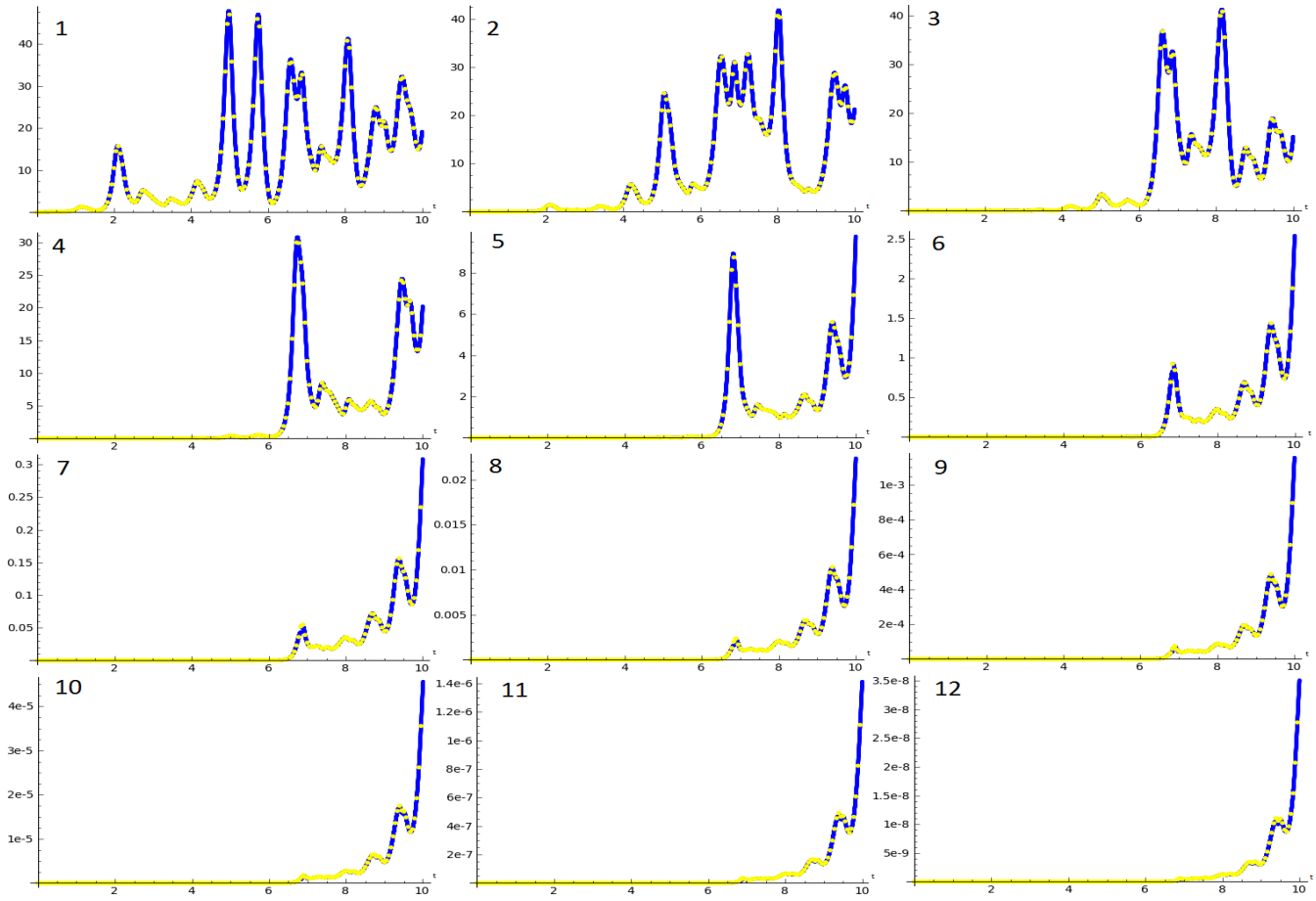

FIGURE 2.6 – Itérations 1 à 12

Les équations de Lorenz montre que l'algorithme peut nécessiter plus d'itérations pour converger suffisamment, elles sont un bon exemple où il pourrait être utile d'essayer de déterminer les constantes qui vont bien dans le théorème de convergence.

2.4 De l'importance d'un solveur grossier adapté au problème

Nous avons énoncé précédemment le fait que nous utiliserions l'algorithme RK4 comme solveur grossier, pour cause d'imprécision des algorithmes plus faibles. Voici ici un schéma de ce que donne une résolution sur le Brusselator avec comme solveur grossier le RK3.

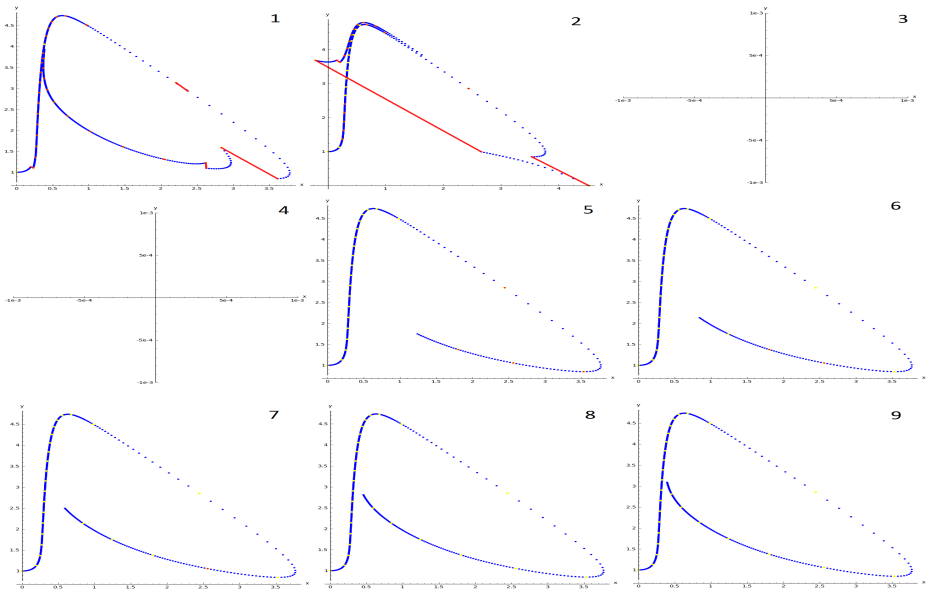

FIGURE 2.7 – Itérations 1 à 9

Ici, le schéma est grossièrement divergent au départ, et finalement se stabilise, mais on peut constater le décalage avec la résolution ayant RK4 comme grossier. Le schéma des erreurs permet de mieux voir ce point.

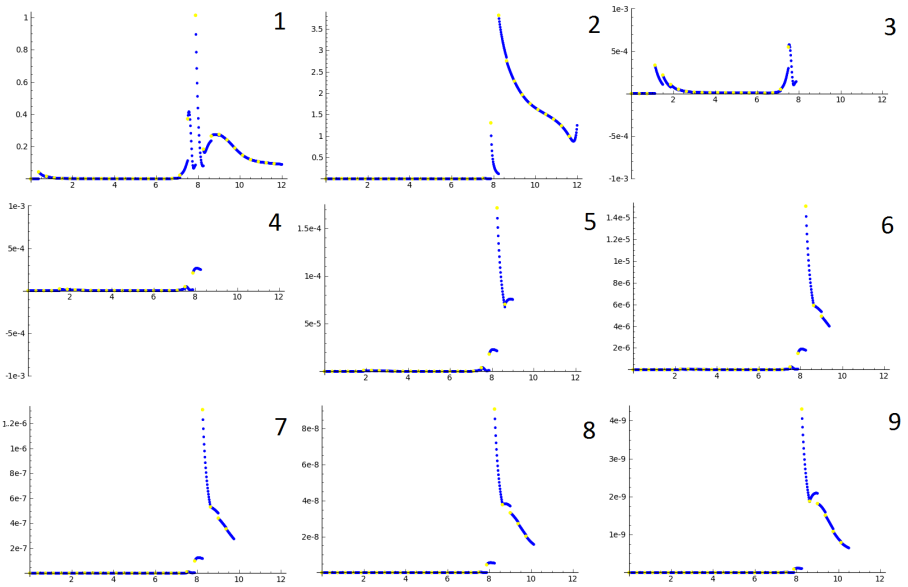

FIGURE 2.8 – Itérations 1 à 9

Chapitre 3

Une variante de l'algorithme avec méthode de type quasi-Newton

3.1 L'algorithme modifié

3.1.1 Motivation de l'algorithme

On reprend du développement de l'algorithme pararéel classique l'équation $-[u_{n+1}^k - \mathcal{F}(u_n^k)] = \mathrm{D}\mathcal{F}(u_n^k)(u_n^{k+1} - u_n^k) + (u_{n+1}^{k+1} - u_{n+1}^k)$ (obtenue « à la ligne $n+1$ »).

L'algorithme pararéel faisait l'approximation $\mathrm{D}\mathcal{F}(u_n^k)(u_n^{k+1} - u_n^k) \simeq \mathcal{G}(u_n^{k+1}) - \mathcal{G}(u_n^k)$. Ici, nous allons utiliser une méthode d'optimisation, en cherchant une approximation sous la forme : $\mathrm{D}\mathcal{F}(u_n^k)(u_n^{k+1} - u_n^k) \simeq \mathcal{F}(u_n^{k+1}) - \mathcal{F}(u_n^k) \simeq A_n^{k+1}[\mathcal{G}(u_n^k + 1) - \mathcal{G}(u_n^k)]$ où A_n^{k+1} est une matrice carrée.

On obtient A_n^{k+1} en l'incrémentant à chaque itération, donc on la cherche sous la forme $A_n^{k+1} = A_n^k + C_n^k$, le but étant que, à convergence, C_n^k tende vers 0, et donc que A_n^k tende vers une matrice A telle que $\mathrm{D}\mathcal{F} = A.\mathrm{D}\mathcal{G}$.

On note $\Delta\mathcal{F} = \mathcal{F}(u_n^{k+1}) - \mathcal{F}(u_n^k)$ et $\Delta\mathcal{G}$ son analogue. On cherche donc A_n^{k+1} de la forme $A_n^k + C_n^k$ et telle que $\Delta\mathcal{F} = A_n^{k+1}.\Delta\mathcal{G}$, donc $\Delta\mathcal{F} = A_n^k.\Delta\mathcal{G} + C_n^k.\Delta\mathcal{G}$. D'où $C_n^k.\Delta\mathcal{G} = \Delta\mathcal{F} - A_n^k.\Delta\mathcal{G}$ qu'on note d_n^k.
On prend comme candidat pour C_n^k une matrice de rang 1, donc de la forme $v_n^k.{}^t(w_n^k)$. Alors $d_n^k = v_n^k.{}^t(w_n^k) = v_n^k \langle w_n^k, \Delta\mathcal{G}\rangle$. Il suffit de choisir $v_n^k = d_n^k$ et de trouver w_n^k tel que $\langle w_n^k, \Delta\mathcal{G}\rangle = 1$. $w_n^k = \frac{\Delta\mathcal{G}}{\|\Delta\mathcal{G}\|^2}$ convient.

Pour initialiser l'algorithme, il nous faut les matrices A_n^0. On les prendra égales à l'identité, on aura ainsi pour la première itération celle du pararéel. Dans le cadre de cet algorithme modifié, la formule de récurrence sera la suivante :

$$u_{n+1}^{k+1} = \mathcal{F}(u_n^k) + A_n^k\Big(\mathcal{G}(u_n^{k+1}) - \mathcal{G}(u_n^k)\Big)$$

3.1.2 Les bases sur lesquelles l'algorithme repose

L'algorithme est basé sur le principe des méthodes de type Quasi-Newton, il a été adapté à partir de la méthode de Broyden, tout en conservant les idées du calcul parallèle. Plus de détails concernant la méthode de Broyden sont fournis en annexe.

3.1.3 Le pseudo-code

Précision : quand on déclare $UGold = matrice[N][dim]$, les matrices sont indexées de $0,0$ à $N-1, dim-1$.

```
UGold=matrice[N][dim]
UGold=0
UGnew=UGold
UG=UGold
UFold=matrice[N][pas+1][dim]
UFold=0
UFnew=UFold
C=matrice[N][dim][dim]
C=0
V=matrice[N][dim]
```

```
V=0
W=matrice[N][dim]
W=0
A=matrice[N][dim][dim]
dFin=matrice[N][dim]
dFin=0
dGrossier=matrice[N][dim]
dGrossier=0
H=T/N
h=H/pas
pour i allant de 0 à N-1
    A[i] = identité(dim)
fait
UGold=Grossier(u0,H)
UGnew=UGold
UG=UGold
UFold=Fin(UGold,h,N)
UFnew=UFol
Afficher(UFnew)
continuer=1
iteration=0
tant que continuer=1 faire
    iteration=iteration+1
    pour j allant de 1 à N-1 faire
        UG[j]=UFold[j-1]+A[j-1]*[UGnew[j-1]-UGold[j-1]]
        UGnew[j]=Grossierunpas(UG[j],H)
    fait
    UFnew=Fin(UG,h,N)
    pour j allant de 0 à N-1 faire
        dFin[j] = UFnew[j][pas]-UFold[j][pas]
    fait
    UFold = UFnew
    dGrossier = UGnew-UGold
    UGold=UGnew
    pour j allant de 1 à N-1 faire
        V[j] = dFin[j]-A[j]dGrossier[j]
        W[j] = dGrossier[j]/(<dGrossier[j],dGrossier[j]>)
    fait
    pour j allant de 1 à N-1 faire
        C[j]=V[j]*transpose(W[j])
    fait
    pour j allant de 1 à N-1 faire
        A[j]=A[j]+C[j]
    fait
    Afficher(UFnew)
    demander(continuer)
fait
```

3.2 Son implémentation

L'implémentation de l'algorithme modifié nous a posé plusieurs problèmes. Il nous a fallu conditionner sur la valeur de la norme de ΔG_n^k, qui était rapidement proche de 0, nous avons donc rajouté un ε, et travaillé avec les ΔG_n^k de norme supérieur à ε (quand elles sont plus petites, on ne modifie pas la matrice associée). Le problème est que si ce epsilon est suffisamment petit, l'algorithme rencontre des problèmes de division par zéro, si on le fixe grand, la résolution se rapproche de la méthode pararéelle classique, et si on choisit une valeur moyenne finement, on a un algorithme convergeant, mais plus faible que l'algorithme pararéel.

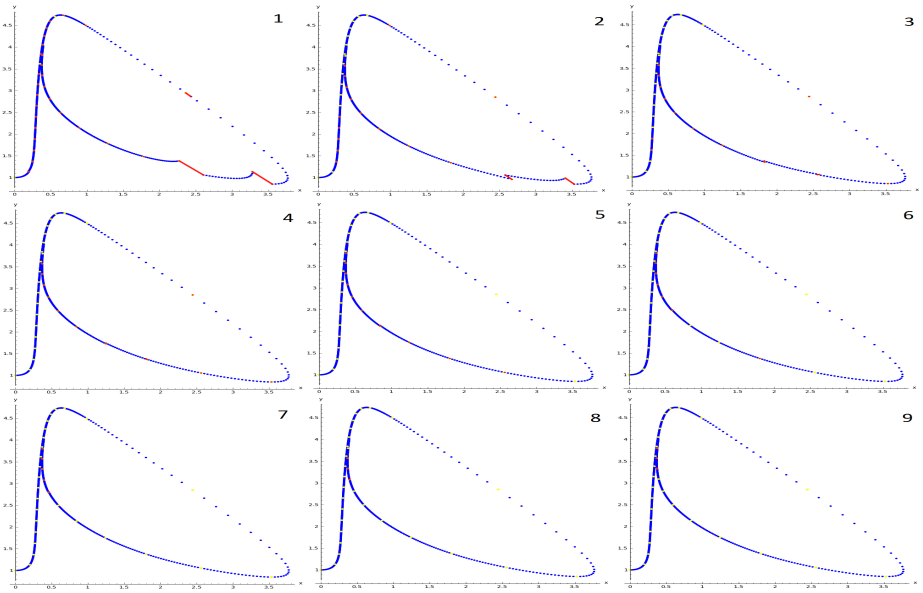

FIGURE 3.1 – Itérations 1 à 9

Voici les erreurs associées.

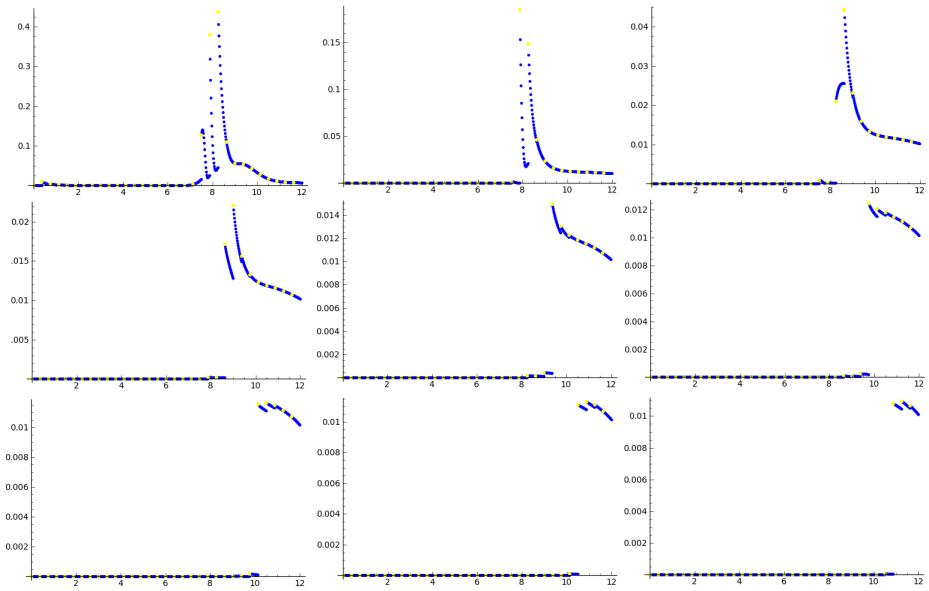

FIGURE 3.2 – Itérations 1 à 9

Nous avons plusieurs explications pouvant justifier ces résultats. La première est que notre programme souffre d'erreurs d'implémentation que nous n'avons pas repérées faute de temps. Le seconde est que l'algorithme nécessite d'être affiné pour pouvoir en tirer parti de la meilleure façon. Une idée serait peut-être de tenter une prédiction des ε qui vont bien à chaque itération.

Conclusion

Au cours de ce stage, nous avons pu nous familiariser avec, les outils de calcul numérique que sont `Sage` avec `Weave`, et ainsi expérimenter des méthodes parallèles de résolution d'équations différentielles ordinaires. Nous avons cependant été confrontés à beaucoup de problèmes techniques inintéressants et chronovores, ce qui ne nous a pas permis de faire tout ce que nous espérions accomplir. Nous avons ainsi manqué de temps, ce qui est dommage.

Plusieurs voies mériteraient d'être creusées, et pourraient apporter des résultats supplémentaires. Parmi lesquelles :

- L'influence de ε dans le calcul s'inspirant des méthodes de type Quasi-Newton.
- La vérification, et l'amélioration de l'algorithme, en vue d'obtenir des résultats de convergence plus probants.
- Nous avons entamé une installation des outils sur les ordinateurs du département de mathématiques. Notre travail a suscité l'attention de Nicolas Pajor, qui souhaite installer ces outils de manière globale. D'éventuels tests futurs pourraient s'avérer concluants. D'autre part, le non-déterminisme du « bug » nous empêchant d'utiliser notre algorithme sur un grand nombre de machines serait peut-être levé.
- Avec des valeurs de ε moyennes, nous avons constaté que l'algorithme modifié avait de meilleurs résultats que l'algorithme pararéel à la troisième étape (itération 2) étape, puis s'essouflait par la suite, nous aurions voulu chercher le motif de cette baisse de productivité.

En tout état de cause, ce stage nous a permis de mieux comprendre les méthodes de recherche de solution numérique, ainsi que l'optimisation à l'aide de méthodes de type Quasi-Newton. Nous tenons à remercier Florian De Vuyst pour son aide et son soutien durant ce stage.

Bibliographie

Bal, G. (2002). Parallelization in time of (stochastic) ordinary differential equations.

Bonnans, J. (2006). *Optimisation continue : cours et problèmes corrigés*, Dunod, Paris.

Brezinski, C. & Redivo-Zaglia, M. (2006). *Méthodes numériques itératives*, Ellipses, Paris.

Demailly, J.-P. (2006). *Analyse numérique et équations différentielles*, EDP Sciences, Les Ulis, France.

Gander, M. J. & Hairer, E. (2006). Nonlinear convergence analysis for the parareal algorithm, *DD17 Proceedings*.

Gander, M. J. & Vandewalle, S. (2005). Analysis of the parareal time-parallel time-integration method, *Technical Report TW 443*, KU Leuven.

Lions, J.-L., Maday, Y. & Turinici, G. (2001). Résolution d'EDP par un schéma en temps « pararéel », *C.R. Acad. Sci. Paris, Série I* **332** : 661–668.

Staff, G. A. & Rønquist, E. (2003). The parareal algorithm : A survey of present work.

Annexes

Annexe A

La méthode de Runge-Kutta

A.1 Le principe

On s'intéressera pour exposer la méthode générale de Runge-Kutta au problème de Cauchy suivant :

$$\begin{cases} y' = f(t,y), & t \in [t_0, t_0+T] \\ y(t_0) = y_0 \end{cases}$$

On subdivise l'intervalle en $t_0 < t_1 < \ldots < t_N = t0 + T$.
On cherche à calculer les points (t_n, y_n) en utilisant les points intermédiaires :

$$t_{n,i} = t_n + c_i.h_n, \quad 1 \leqslant i \leqslant q, \quad c_i \in [0,1]$$

Et on pose $p_{n,i} = f(t_{n,i}, y_{n,i})$.
Si z est une solution exacte du problème de Cauchy, on a :

$$\begin{aligned} z(t_{n,i}) &= z(t_n) + \int_{t_n}^{t_{n,i}} f(t, z(t))dt \\ &\underset{\text{ChdV } t=t_n+uh_n}{=} z(t_n) + h_n \int_0^{c_i} \end{aligned}$$

De même

$$z(t_{n+1}) = z(t_n) + h_n \int_0^1 f(t_n + uh_n, z(t_n + uh_n))du$$

On utilise différentes méthodes d'intégration approchée, une pour chaque intégrale

$$\int_0^{c_i} g(t)dt \approx \sum_{1 \leqslant j < i} a_{i,j} g(c_j)$$

Par ailleurs, on calcule aussi des valeurs approchées des intégrales sur $[0,1]$

$$\int_0^1 g(t)dt \approx \sum_{j=1}^{q} b_j g(c_j)$$

D'où

$$z(t_{n,i}) \approx z(t_n) + h_n \sum_{1 \leqslant j < i} a_{i,j} f(t_{n,j}, z(t_{n,j}))z(t_{n+1}) \qquad \approx z(t_n) + h_n \sum_{j=1}^{q} b_j f(t_{n,j}, z(t_{n,j}))$$

L'algorithme de Runge-Kutta est donc le suivant :

$$\begin{cases} \left[\begin{array}{l} t_{n,i} = t_n + c_i h_n \\ y_{n,i} = y_n + h_n \sum_{1 \leqslant j < i} a_{i,j} p_{n,j} \\ p_{n,i} = f(t_{n,i}, y_{n,i}) \end{array} \right] 1 \leqslant i \leqslant q \\ t_{n+1} = t_n + h_n \\ y_{n+1} = y_n + h_n \sum_{j=1}^{q} b_j p_{n,j} \end{cases}$$

Par convention, on prend $a_{i,j} = 0$ pour $j \geqslant i$. De plus on suppose que ces méthodes sont d'ordre 0 au moins, c'est-à-dire que $c_i = \sum_{1 \leqslant j < i} a_{i,j}$ et $\sum_{j=1}^{q} b_j = 1$.

A.2 Exemples

Pour $q = 1$, le seul choix possible est $c_1 = 0, \quad b_1 = 1$. L'algorithme devient alors :

$$\begin{cases} p_{n,1} = f(t_n, y_n) \\ t_{n+1} = t_n + h_n \\ y_{n+1} = y_n + h_n p_{n,1} \end{cases}$$

C'est la méthode d'Euler.

Pour $q = 2$, les choix possibles sont $c_1 = 0, \quad c_2 = a_{2,1} = \alpha, \quad b_2 = 1 - b_1 = \frac{1}{2\alpha}$ avec $\alpha \in]0, 1]$.
Pour $\alpha = \frac{1}{2}$, on obtient la méthode du point milieu.
Pour $\alpha = 1$, on obtient la méthode de Heun.

La méthode Runge-Kutta classique (RK4) s'obtient avec q=4. La seule solution possible est $c_1 = 0, c_2 = c_3 = \frac{1}{2}, \quad a_{2,1} = a_{3,2} = \frac{1}{2}, a_{4,3} = 1, \quad b_1 = b_4 = \frac{1}{6}, b_2 = b_3 = \frac{2}{6}$, tous les autres coefficients nuls.
L'algorithme obtenu est le suivant :

$$\begin{cases} p_{n,1} = f(t_n, y_n) \\ t_{n,2} = t_n + \frac{1}{2} h_n \\ y_{n,2} = y_n + \frac{1}{2} h_n p_{n,1} \\ p_{n,1} = f(t_{n,2}, y_{n,2}) \\ y_{n,3} = y_n + \frac{1}{2} h_n p_{n,2} \qquad (t_{n,3} = t_{n,2}) \\ p_{n,3} = f(t_{n,3}, y_{n,3}) \\ t_{n+1} = t_n + h_n \qquad (t_{n,4} = t_{n+1}) \\ y_{n,4} = y_n + h_n p_{n,3} \\ p_{n,4} = f(t_{n,4}, y_{n,4}) \\ y_{n+1} = y_n + \frac{h_n}{6}(p_{n,1} + 2p_{n,2} + 2p_{n,3} + p_{n,4}) \end{cases}$$

A.3 Ordre des méthodes de Runge-Kutta

Les méthodes RK2, RK3 et RK4 sont d'ordre respectif 2, 3 et 4. Au delà de ce terme, les ordres de convergence sont plus difficiles à déterminer. On pourra se référer au Demailly (2006) pour une preuve des ordres dans le cas de RK2, 3 et 4. Ces preuves étant très calculatoires, plus que difficiles théoriquement parlant, elles ne seront pas abordées ici.

Annexe B

La méthode de Broyden

B.1 La méthode de la sécante en dimension 1

Lorsqu'on recherche un point d'annulation d'une fonction, il existe une méthode pouvant nous donner rapidement ce point, il s'agit de la méthode de Newton. Le principe est de considérer un réel x_0, et de trouver une suite x_n convergeant vers le point d'annulation en question, à l'aide de la formule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On utilise ici la méthode des différences finies pour approximer f'

$$f'(x_n) \simeq \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

ainsi,

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1}) \times f(x_n)}{f(x_n) - f(x_{n-1})}$$

B.2 Généralisation à la dimension n

L'idée générale est d'utiliser la matrice Jacobienne à la place de la dérivée, et d'écrire alors

$$J_n(x_n - x_{n-1}) \simeq F(x_n) - F(x_{n-1})$$

cependant, le caractère multi-dimensionnel du problème implique l'existence de multiples direction pour calculer x_{n+1}. Broyden suggère alors de prendre la direction qui provoque un changement minimal sur J_n au sens de la norme matricielle usuelle. On a alors, après s'être tapé plein de calculs chiants :

$$J_{n+1} = J_n + \frac{(F(x_{n+1}) - F(x_n)) - J_n(x_{n+1} - x_n)}{||x_{n+1} - x_n||^2} {}^t(x_{n+1} - x_n)$$

on aura alors

$$x_{n+1} = x_n - J_n^{-1} F(x_n)$$

B.3 Méthode pour éviter l'inversion matricielle

Une méthode pour contourner le calcul de l'inverse de la jacobienne est de faire les modifications directement sur les inverses. Broyden fournit une formule de la forme :

$$J_{n+1}^{-1} = J_n^{-1} + \frac{\Delta x_{n+1} - J_n^{-1}\Delta F_{n+1}}{{}^t\Delta x_{n+1} J_n^{-1} \Delta F_{n+1}} ({}^t\Delta x_{n+1} J_n^{-1})$$

avec $\Delta x_n = x_n - x_{n-1}$, $\Delta F_n = F(x_n) - F(x_{n-1})$ la formule que nous utilisons ressemble à une autre forme proposée par Broyden :

$$J_{n+1}^{-1} = J_n^{-1} + \frac{\Delta x_{n+1} - J_n^{-1}\Delta F_{n+1}}{{}^t\Delta F_{n+1} \Delta F_{n+1}} ({}^t\Delta F_{n+1})$$

On pourra se référer à Brezinski & Redivo-Zaglia (2006) pour plus de détails.

Annexe C

Les codes sources

Voici les codes source, colorés syntaxiquement, prévoir de l'aspirine.