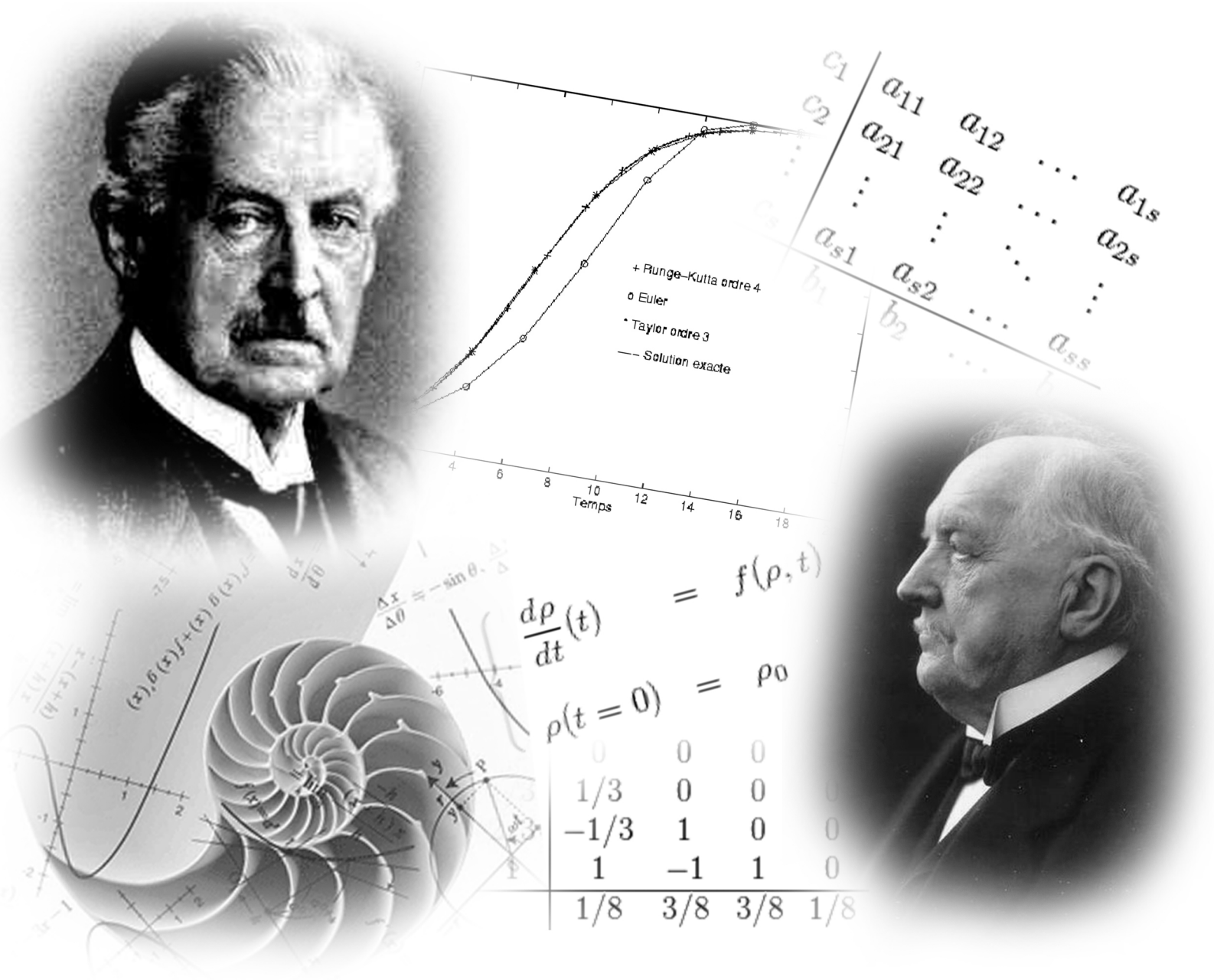
Projet GM3
Étude et application d'une méthode Runge-Kutta (RK34)



Cécile Poulard
Elliot Sisteron

À l'attention de M. Forcadel

INSA ROUEN
À taille humaine, à l'échelle du monde

Introduction

Lorsque l'on étudie des phénomènes physiques, biologiques, économiques... apparaissent des relations que l'on peut alors décrire par un modèle mathématique. Les équations différentielles sont utilisées pour construire ces modèles. Dans le cas particulier des équations différentielles ordinaires (EDO), cela s'exprime mathématiquement par :

- une variable scalaire $x \in [a, b] \subset \mathbb{R}$;
- une fonction $y : [a, b] \to \mathbb{R}^m$ dépendant de cette variable scalaire (i.e. $y(x)$) et caractérisant le phénomène étudié ;
- une relation entre cette fonction y et ses dérivées y', y'', etc. Par exemple, si y' est liée à y, on peut exprimer cette relation par une fonction $f : [a, b] \times \mathbb{R}^m \to \mathbb{R}^m$ dépendant de x et y (i.e. $f(x, y(x))$) telle que $y' = f(x, y)$;
- une condition initiale ou condition de Cauchy : on connait l'état du phénomène à x_0 fixé, on parle alors de problème de Cauchy. On supposera dans la suite de ce projet que $x_0 = a$, et donc que $y(a) = y_0 \in \mathbb{R}^m$ fixé.

On peut alors toujours se ramener à un système différentiel d'ordre 1 :

$$(S) \begin{cases} \forall x \in [a, b],\ y'(x) &= f(x, y(x)) \\ y(a) &= y_0 \end{cases}$$

Le théorème de Cauchy-Lipschitz-Lindelöf-Picard affirme que ce type de système possède toujours une unique solution globale lorsque la fonction f est continue sur son ensemble de définition et lipschitzienne par rapport à sa seconde variable (i.e. y). On peut ainsi se rendre compte que la plupart de ces systèmes admettent une solution unique.

Vu qu'il existe une solution mais qu'il n'est pas toujours possible de la trouver, on comprend rapidement le besoin de méthodes numériques pour approcher cette solution. C'est pour cette raison qu'au fil des siècles sont apparues plusieurs méthodes de résolution des EDO : Euler, la méthode du point-milieu... et plus récemment les méthodes de Runge-Kutta.

Dans ce projet, on se propose d'étudier la méthode de Runge-Kutta à formules emboîtées RK_{34}. Dans un premier temps, nous présenterons le cadre d'apparition des méthodes de Runge et Kutta puis nous présenterons en détails la méthode RK_{34}. Ensuite, nous nous attarderons sur l'implémentation de cette méthode ; d'abord en concevant un algorithme, puis en discutant du coût des calculs impliqués et des erreurs à prévoir. Tout cela dans le but de développer un programme C chargé d'appliquer la méthode sur une fonction f. Enfin, nous utiliserons divers exemples pour mettre en évidence les avantages et désavantages de cette méthode numérique, ce qui nous permettra alors de faire un petit bilan concernant son efficacité et son intérêt.

Table des matières

Présentation de la méthode Runge-Kutta RK_{34}

Les méthodes numériques de résolution d'EDO cherchent à approximer la solution y d'un problème de Cauchy en subdivisionnant d'abord l'intervalle $[a, b]$ par une suite $(x_k)_{k\in[\![0,n]\!]}$ telle que $a = x_0 < x_1 < \ldots < x_n = b$. On pose la suite $(h_k)_{k\in[\![0,n-1]\!]} = (x_{k+1} - x_k)_{k\in[\![0,n-1]\!]}$. Selon la méthode utilisée, en découle alors une suite de points $(x_k, y_k)_{k\in[\![0,n]\!]}$ relativement proche de la suite de points de la solution discrétisée $(x_k, y(x_k))_{k\in[\![0,n]\!]}$.

1.1 Méthodes de résolution des EDO à un pas : de Euler à Runge-Kutta

Dans ce projet, nous nous intéresserons à une méthode à un pas : chaque terme y_{k+1} sera calculé à partir du terme précédent y_k. De plus, cette méthode sera explicite, i.e. y_{k+1} sera donné explicitement par rapport à y_k. Il existera donc une application ϕ et $h^* \in [0, b-a]$ telle que $\phi : [a, b] \times \mathbb{R}^m \times [0, h^*] \to \mathbb{R}^m$ et :

$$(y_k)_{k\in[\![0,n]\!]} \begin{cases} y_0 &= y(a) \\ y_{k+1} &= y_k + h_k\phi(x_k, y_k, h_k) \end{cases}$$

1.1.1 Notions théoriques

Consistance

Pour une méthode à un pas donné, on définit l'erreur de consistance en $k \in [\![0, n-1]\!]$ par :

$$\epsilon_k = y(x_{k+1}) - y(x_k) - h_k\phi(x_k, y(x_k), h_k)$$

On dira alors que la méthode est consistante lorsque, pour $h = \max_{k\in[\![0,n-1]\!]}(h_k)$:

$$\sum_{k=0}^{n-1} |\epsilon_k| \underset{h\to 0}{\longrightarrow} 0$$

On peut alors montrer que la consistance d'une méthode est équivalente à :

$$\phi(x, y(x), 0) = f(x, y(x))$$

Stabilité théorique

Si l'on définit les suites $(\hat{\epsilon}_k)_{k\in[\![0,n]\!]}$ et $(z_k)_{k\in[\![0,n]\!]}$ par :

$$\forall k \in [\![0, n-1]\!],\ z_{k+1} = z_k + h_k\phi(x_k, y_k, h_k) + \hat{\epsilon}_k$$

Alors on dira que la méthode est stable théoriquement ssi :

$$\exists M > 0\ /\ \max_{k\in[\![0,n]\!]}(y_k - z_k) \leq M(|y_0 - z_0| + \sum_{k=0}^{n-1} |\hat{\epsilon}_k|)$$

On peut alors montrer qu'une condition suffisante pour la stabilité théorique est que la fonction ϕ soit lipschitzienne par rapport à sa seconde variable.

Convergence

On dira que la méthode est convergente dès lors que :

$$\max_{k\in[\![0,n]\!]} |y(x_k) - y_k| \underset{h\to 0}{\longrightarrow} 0$$

La consistance et la stabilité théorique entraînent alors la convergence de la méthode.

Ordre

L'ordre nous donne une idée de la vitesse de convergence de la méthode. On dira que la méthode est d'ordre p ssi il existe $K > 0$ ne dépendant que de y et de ϕ tel que pour tout $y \in \mathcal{C}^{p+1}([a,b])$ vérifiant (S) :

$$\sum_{k=0}^{n-1} |\epsilon_k| \leq Kh^p$$

Si de plus la méthode est stable théoriquement, on a :

$$\max_{k\in[\![0,n]\!]} |y(x_k) - y_k| \leq MKh^p$$

Si f est p fois continuement dérivable sur son domaine de définition, et que les fonctions $\left(\frac{\partial^j \phi}{\partial h^j}(x, y(x), h)\right)_{j\in[\![0,p-1]\!]}$ existent et sont continues sur le domaine de définition de ϕ, dire qu'une méthode est d'ordre p est équivalent à :

$$\forall j \in [\![0, p-1]\!],\ \frac{\partial^j \phi}{\partial h^j}(x, y(x), 0) = \frac{1}{j+1} f^{(j)}(x, y(x))$$

1.1.2 Euler

Une des première méthode numérique de résolution d'EDO nous vient du célèbre mathématicien Leonhard Euler au XVIII$^{\text{e}}$ siècle. Il fit des découvertes remarquables malgré un handicap très pénalisant : il devient aveugle en 1771 mais continue ses travaux en dictant ses textes scientifiques à ses fils ou à son valet.

FIGURE 1.1 – Leonhard Euler (1707 - 1783)

Sa méthode repose sur la définition de la dérivée d'une fonction, pour une application $y : [a, b] \to \mathbb{R}^m$ dérivable en $x \in [a, b]$, on a l'expression de sa dérivée en x :

$$y'(x) = \lim_{t \to x} \frac{y(t) - y(x)}{t - x} = \lim_{h \to 0} \frac{y(x + h) - y(x)}{h}$$

Pour h proche de 0, on a donc l'approximation suivante :

$$y(x + h) \approx y(x) + hy'(x)$$

Dans le cas du système (S) à résoudre, on a $y'(x) = f(x, y(x))$. Ainsi,

$$y(x + h) \approx y(x) + hf(x, y(x))$$

Et donc, pour un pas maximal h proche de 0 :

$$y(x_k + h_k) \approx y(x_k) + h_k f(x_k, y(x_k))$$

On en déduit donc une méthode de calcul de y_{k+1} relativement correcte en posant $\forall k \in [\![0, n - 1]\!]$:

$$y_{k+1} = y_k + h_k f(x_k, y_k)$$

i.e.

$$\phi(x_k, y_k, h_k) = f(x_k, y_k)$$

Cette méthode est clairement consistante et stable théoriquement, elle est donc convergente.

1.1.3 Point-milieu

Le problème de la méthode d'Euler est qu'elle est d'ordre 1. Une solution serait de passer par un développement de Taylor, pour une application $y : [a, b] \to \mathbb{R}^m$ dérivable $p + 1$ fois en $x \in [a, b]$ et $y^{(p+1)}$ bornée :

$$\begin{aligned} y(x + h) &= \sum_{k=0}^{p} \frac{h^k}{k!} y^{(k)}(x) + O(h^{p+1}) \\ &= y(x) + hy'(x) + \frac{h^2}{2} y''(x) + \ldots + \frac{h^p}{p!} y^{(p)}(x) + O(h^{p+1}) \\ &= y(x) + hf(x, y(x)) + \frac{h^2}{2} \frac{df}{dx}(x, y(x)) + \ldots + O(h^{p+1}) \end{aligned}$$

Il devient nécessaire de calculer les dérivées partielles de f, qui sont difficiles à évaluer... par exemple, pour une méthode d'ordre 2 :

$$y(x + h) = y(x) + hf(x, y(x)) + \frac{h^2}{2} \frac{df}{dx}(x, y(x)) + O(h^3)$$

Et alors on doit calculer :

$$\begin{aligned} \frac{df}{dx}(x, y(x)) &= \frac{\partial f}{\partial x}(x, y(x)) + \frac{\partial f}{\partial y}(x, y(x)) \frac{dy}{dx}(x) \\ &= \frac{\partial f}{\partial x}(x, y(x)) + \frac{\partial f}{\partial y}(x, y(x)) f(x, y(x)) \end{aligned}$$

La méthode du point-milieu nous évite ce calcul fastidieux en centrant l'évaluation de la dérivée au point-milieu $x_m = \frac{x_k+x_{k+1}}{2}$:

$$\begin{aligned} y(x_k + h_k) &= y(x_{k+1}) \\ &= y(x_m + \frac{h_k}{2}) \\ &= y(x_m) + \frac{h_k}{2}y'(x_m) + \frac{h_k^2}{8}y''(x_m) + O(h_k^3) \\ y(x_k) &= y(x_m - \frac{h_k}{2}) \\ &= y(x_m) - \frac{h_k}{2}y'(x_m) + \frac{h_k^2}{8}y''(x_m) + O(h_k^3) \end{aligned}$$

On en déduit :

$$\begin{aligned} & y(x_{k+1}) - y(x_k) = h_k y'(x_m) + O(h_k^3) \\ \Leftrightarrow\ & y(x_{k+1}) = y(x_k) + h_k f(x_m, y(x_m)) + O(h_k^3) \end{aligned}$$

Or,

$$\begin{aligned} y(x_m) &= y(x_k + \frac{h_k}{2}) \\ &= y(x_k) + \frac{h_k}{2}y'(x_k) + O(h_k^2) \\ &= y(x_k) + \frac{h_k}{2}f(x_k, y(x_k)) + O(h_k^2) \end{aligned}$$

D'où l'expression de la méthode du point-milieu en posant $\forall k \in [\![0, n-1]\!]$:

$$y_{k+1} = y_k + h_k f\left(x_k + \frac{h_k}{2}, y_k + \frac{h_k}{2}f(x_k, y_k)\right)$$

i.e.

$$\phi(x_k, y_k, h_k) = f\left(x_k + \frac{h_k}{2}, y_k + \frac{h_k}{2}f(x_k, y_k)\right)$$

On peut montrer que cette méthode est d'ordre 2, ce qui est déjà plus intéressant. D'autres méthodes furent développées à la fin du XIX^{e} siècle, notamment la méthode de Heun, la méthode d'Euler modifiée. Mais c'est à l'aube du XX^{e} siècle que des méthodes bien plus puissantes furent énoncées...

1.1.4 Méthodes de Runge-Kutta

En 1901, deux scientifiques allemands, Carl Runge et Martin Kutta, développèrent des méthodes de résolution numériques des EDO : les méthodes Runge-Kutta. Carl Runge était un physicien, mais c'est en faisant ses recherches sur les spectres atomiques qu'il développa ces méthodes pour résoudre des systèmes différentiels. Il pratiquait tellement les mathématiques que les physiciens le méprenait pour un mathématicien, et inversement. Kutta, lui, travaillait en mathématiques appliquées et apporta dans sa thèse une contribution très importante à la construction de ces méthodes encore utilisées aujourd'hui.

FIGURE 1.2 – Carl David Tolmé Runge (1856 - 1927)

FIGURE 1.3 – Wilhelm Martin Kutta (1867 - 1944)

Le principe des méthodes de Runge et Kutta repose sur la création de points intermédiaires $(x_{k,i})_{i\in[\![1,r]\!]}$ pour chaque intervalle $[x_k, x_{k+1}]$ définis par : $x_{k,i} = x_k + \theta_i h_k$ avec $\theta_i \in [0,1]$. On se donne des coefficients réels $(a_{i,j})_{i,j\in[\![1,r]\!]}$ et $(c_i)_{i\in[\![1,r]\!]}$. On définit alors :

$$y_{k,i} = y_k + h_k \sum_{j=1}^{r} a_{i,j} f(x_{k,j}, y_{k,j})$$

Et la suite $(y_k)_{k\in[\![0,n]\!]}$ par y_0 et :

$$y_{k+1} = y_k + h_k \sum_{i=1}^{r} c_i f(x_{k,i}, y_{k,i})$$

Depuis le travail de John Butcher, un mathématicien Néo-Zélandais reconnu pour son apport exceptionnel en analyse numérique, on visualise les coefficients d'une méthode de Runge-Kutta par le tableau suivant, aussi appelé tableau de Butcher :

$$\begin{array}{c|ccc} \theta_1 & a_{11} & \dots & a_{1r} \\ \vdots & \vdots & & \vdots \\ \theta_r & a_{r1} & \dots & a_{rr} \\ \hline & c_1 & \dots & c_r \end{array}$$

FIGURE 1.4 – Tableau de Butcher

FIGURE 1.5 – John Charles Butcher (1933)

Selon la valeur des coefficients de A, on peut avoir une méthode implicite ou explicite. En particulier, si A est triangulaire inférieure à diagonale nulle, on retrouve bien une méthode explicite et :

$$\phi(x_k, y_k, h_k) = \sum_{i=1}^{r} c_i f\left(x_k + \theta_i h_k, y_k + h_k \sum_{j=1}^{i-1} a_{i,j} f(x_{k,j}, y_{k,j})\right)$$

On peut montrer que toutes les méthodes de Runge-Kutta explicites sont stables théoriquement, il suffit alors de trouver les bons coefficients pour avoir la convergence.

En fait, les méthodes de Runge-Kutta généralisent des méthodes déjà connues. Par exemple, pour les méthodes de Euler et du point-milieu on a les tableaux de Butcher suivants :

0	0
	1

FIGURE 1.6 – Méthode d'Euler

0	0	0
1/2	1/2	0
	0	1

FIGURE 1.7 – Méthode du point-milieu

Le grand intérêt est que l'on peut alors construire des méthodes d'ordre p en trouvant les coefficients adéquats, il suffit d'avoir :

$$\forall j \in [\![0, p-1]\!],\ \frac{\partial^j \phi}{\partial h^j}(x, y(x), 0) = \frac{1}{j+1} f^{(j)}(x, y(x))$$

i.e. pour une méthode d'ordre 1 (et donc consistante) :

$$\sum_{i=1}^{r} c_i = 1$$

Pour une méthode d'ordre 2, il nous suffit d'avoir la condition de la méthode d'ordre 1 et :

$$\frac{\partial \phi}{\partial h}(x, y(x), 0) = \frac{1}{2} \frac{df}{dx}(x, y(x))$$

i.e.

$$\sum_{i=1}^{r} c_i \theta_i \frac{\partial f}{\partial x}(x, y(x)) + \sum_{i=1}^{r} c_i \sum_{j=1}^{i-1} a_{i,j} f(x, y(x)) \frac{\partial f}{\partial y}(x, y(x)) = \frac{1}{2} \left(\frac{\partial f}{\partial x}(x, y(x)) + \frac{\partial f}{\partial y}(x, y(x)) f(x, y(x)) \right)$$

En ayant le même raisonnement et avec beaucoup de calculs, on peut trouver des conditions pour une méthode d'ordre p :

p	conditions
1	$\sum_{i=1}^{r} c_i = 1$
2	$\begin{cases} \sum_{i=1}^{r} c_i \theta_i = \frac{1}{2} \\ \sum_{i=1}^{r} \sum_{j=1}^{i-1} c_i a_{i,j} = \frac{1}{2} \end{cases}$
3	...

FIGURE 1.8 – Les conditions à satisfaire pour une méthode d'ordre p

À mesure que l'ordre augmente, il devient de plus en plus difficile de trouver de bons coefficients étant donné que le nombre de conditions devient très élevé (58 conditions pour $p = 5$). Au XX^{e} siècle, les mathématiciens ont pu assister à une véritable course à la recherche d'une méthode ayant le plus grand ordre possible : la méthode la plus efficace étant d'ordre 2 en 1895 (Runge), 5 en 1901 (Kutta) et 10 en 1978 (Hairer).

1.2 La méthode de Runge-Kutta RK_{34}

La méthode de Runge-Kutta RK_{34} est une méthode dite à « formules emboîtées ».

1.2.1 Les méthodes de Runge et Kutta à formules emboîtées

On dira qu'un couple $RK_{pp'}$ $(p' \geq p+1)$ est une méthode de Runge-Kutta à formules emboîtées si elle est formée par une méthode de Runge-Kutta d'ordre p complétée par une ligne et une colonne afin d'obtenir une méthode d'ordre p'.

$$\begin{array}{c|ccc|c} \theta_1 & a_{11} & \dots & a_{1r} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ \theta_r & a_{r1} & \dots & a_{rr} & 0 \\ 1 & c_1 & \dots & c_r & 0 \\ \hline & c'_1 & \dots & c'_r & c'_{r+1} \end{array}$$

FIGURE 1.9 – Tableau de Butcher d'une méthode $RK_{pp'}$

Dans le cas de la méthode RK_{34}, il s'agit donc d'une méthode de Runge-Kutta d'ordre 3 complétée par une méthode d'ordre 4.

1.2.2 Méthodes de contrôle du pas

Le grand intérêt des méthodes $RK_{pp'}$ est qu'elles permettent un contrôle de l'erreur de consistance pour l'empêcher de dépasser un certain seuil. Si l'on suppose que l'on utilise une méthode d'ordre p pour calculer $(y_k)_{k\in[\![0,n]\!]}$:

$$y_{k+1} = y_k + h_k\phi(x_k, y_k, h_k)$$

On a donc l'expression suivante de l'erreur de consistance :

$$\epsilon_k = y(x_{k+1}) - y(x_k) - h_k\phi(x_k, y(x_k), h_k)$$

On suppose que l'on utilise une autre méthode d'ordre p' telle que $p' \geq p+1$. On aura aussi une expression de l'erreur de consistance commise avec cette méthode :

$$\epsilon^*_k = y(x_{k+1}) - y(x_k) - h_k\phi^*(x_k, y(x_k), h_k) = O(h_k^{p+2})$$

D'où :

$$\epsilon_k = \epsilon^*_k + h_k\left(\phi^*(x_k, y(x_k), h_k) - \phi(x_k, y(x_k), h_k)\right)$$

On ne sait pas calculer $y(x_k)$, mais on peut utiliser un développement limité au premier ordre pour exprimer ϕ en fonction de y_k ; il existe $\bar{y_k}$ compris entre y_k et $y(x_k)$ tel que :

$$\begin{aligned}\epsilon_k &= \epsilon^*_k + h_k\left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) + (y(x_k) - y_k)\left(\frac{\partial\phi^*}{\partial y}(x_k, \bar{y_k}, h_k) - \frac{\partial\phi}{\partial y}(x_k, \bar{y_k}, h_k)\right)\right) \\ &= \epsilon^*_k + h_k\left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) + (y(x_k) - y_k)\left(\frac{\partial\phi^*}{\partial y}(x_k, \bar{y_k}, 0) - \frac{\partial\phi}{\partial y}(x_k, \bar{y_k}, 0) + O(h_k)\right)\right)\end{aligned}$$

Or, par consistance et comme $y(x_k) - y_k = O(h_k^p)$:

$$\begin{aligned}
\epsilon_k &= \epsilon_k^* + h_k \left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) + O(h_k^p) \left(\frac{\partial f}{\partial y}(x_k, \bar{y_k}) - \frac{\partial f}{\partial y}(x_k, \bar{y_k}) + O(h_k) \right) \right) \\
&= \epsilon_k^* + h_k \left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) + O(h_k^{p+1}) \right) \\
&= O(h_k^{p+2}) + h_k \left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) \right) \\
&\approx h_k \left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) \right)
\end{aligned}$$

Finalement, on a bien une bonne approximation calculable $\bar{\epsilon_k}$ de ϵ_k. Dans le cas d'une méthode de Runge-Kutta emboîtée, on a :

$$\begin{aligned}
\bar{\epsilon_k} &= h_k \left(\phi^*(x_k, y_k, h_k) - \phi(x_k, y_k, h_k) \right) \\
&= h_k \left(\sum_{i=1}^{r} (c_i' - c_i) f(x_{k,i}, y_{k,i}) + c_{r+1}' f(x_{k,r+1}, y_{k,r+1}) \right) \\
&= h_k \left(\sum_{i=1}^{r} (c_i' - c_i) f(x_{k,i}, y_{k,i}) + c_{r+1}' f \left(x_k + \theta_{r+1} h_k, y_k + \sum_{j=1}^{r+1} a_{r+1,j} f(x_{k,j}, y_{k,j}) \right) \right) \\
&= h_k \left(\sum_{i=1}^{r} (c_i' - c_i) f(x_{k,i}, y_{k,i}) + c_{r+1}' f(x_{k+1}, y_{k+1}) \right)
\end{aligned}$$

Maintenant, si l'on souhaite avoir une erreur de consistance en dessous d'un certain seuil δ, il suffit de comparer à 1 ce terme :

$$e = \frac{|\bar{\epsilon_k}|}{\delta}$$

— si $e > 1$ alors il faut réduire le pas ;
— si $e < 1$ alors il faut prendre un pas plus grand pour limiter les calculs ;
— si $e = 1$ alors on a un pas optimal.

On aimerait donc modifier le pas après chaque test de telle sorte qu'il soit optimal. Or, on sait que pour h_k au voisinage de 0 :

$$e \approx C h_k^{p+1}$$

Et donc

$$C \approx \frac{e}{h_k^{p+1}}$$

Et, pour un pas optimal :

$$e = 1 \approx C h_{k_{opt}}$$

Finalement, on pourrait prendre :

$$h_{k_{opt}} = h_k \left(\frac{1}{e} \right)^{\frac{1}{p+1}}$$

Mais on préfère se donner une petite marge de sécurité :

$$h_{k_{opt}} = 0.9 h_k \left(\frac{1}{e} \right)^{\frac{1}{p+1}}$$

1.2.3 La méthode de Runge-Kutta RK_{34}

Le tableau de Butcher de la méthode que nous étudierons dans ce projet est le suivant :

0	0	0	0	0	0
2/7	2/7	0	0	0	0
4/7	−8/35	4/5	0	0	0
6/7	29/42	−2/3	5/6	0	0
1	1/6	1/6	5/12	1/4	0
	11/96	7/24	35/96	7/48	1/12

FIGURE 1.10 – Tableau de Butcher de la méthode RK_{34}

Vérifions que la méthode d'ordre 3 non complétée est bien consistante :

$$\begin{aligned}\sum_{i=1}^{r} c_i &= \frac{1\times 2}{6\times 2} + \frac{1\times 2}{6\times 2} + \frac{5}{12} + \frac{1\times 3}{4\times 3}\\ &= 1\end{aligned}$$

Elle est aussi bien d'ordre 2 :

$$\begin{aligned}\sum_{i=1}^{r} c_i\theta_i &= 0 + \frac{2\times 1}{7\times 6} + \frac{4\times 5}{7\times 12} + \frac{6\times 1}{7\times 4}\\ &= \frac{2\times 2}{42\times 2} + \frac{10\times 2}{42\times 2} + \frac{6\times 3}{28\times 3}\\ &= \frac{1}{2}\end{aligned}$$

Et,

$$\begin{aligned}\sum_{i=1}^{r}\sum_{j=1}^{i-1} c_i a_{i,j} &= \frac{1\times 2}{6\times 7} + \frac{5}{12}(-\frac{8}{35} + \frac{4\times 7}{5\times 7}) + \frac{1}{4}(\frac{29}{42} - \frac{2\times 14}{3\times 14} + \frac{5\times 7}{6\times 7})\\ &= \frac{2}{42} + \frac{5}{12}\frac{20}{35} + \frac{1}{4}\frac{36}{42}\\ &= \frac{4}{84} + \frac{20}{84} + \frac{18}{84}\\ &= \frac{1}{2}\end{aligned}$$

On peut aussi vérifier qu'elle est bien d'ordre 3. Nous ne ferons pas ici ce calcul, car le nombre de conditions à vérifier est important.

Vérifions que la méthode emboîtée d'ordre 4 est bien consistante :

$$\begin{aligned}\sum_{i=1}^{r+1} c'_i &= \frac{11}{96} + \frac{7\times 4}{24\times 4} + \frac{35}{96} + \frac{7\times 2}{48\times 2} + \frac{1\times 8}{12\times 8}\\ &= 1\end{aligned}$$

De même, cette méthode est bien d'ordre 2 :

$$\begin{aligned}
\sum_{i=1}^{r+1} c'_i\theta_i &= 0 + \frac{2\times 7}{7\times 24} + \frac{4\times 35}{7\times 96} + \frac{6\times 7}{7\times 48} + \frac{1}{12} \\
&= \frac{2}{24} + \frac{5}{24} + \frac{3}{24} + \frac{2}{24} \\
&= \frac{1}{2} \\
\sum_{i=1}^{r+1}\sum_{j=1}^{i-1} c'_i a_{i,j} &= \frac{7\times 2}{24\times 7} + \frac{35}{96}(-\frac{8}{35} + \frac{4\times 7}{5\times 7}) + \frac{7}{48}(\frac{29}{42} - \frac{2\times 14}{3\times 14} + \frac{5\times 7}{6\times 7}) + \frac{1}{12}(\frac{1\times 2}{6\times 2} + \frac{1\times 2}{6\times 2} + \frac{5}{12} + \frac{1\times 3}{4\times 3}) \\
&= \frac{2}{24} + \frac{35}{96}\frac{20}{35} + \frac{7}{48}\frac{36}{42} + \frac{1}{12}\times 1 \\
&= \frac{8}{96} + \frac{20}{96} + \frac{12}{96} + \frac{8}{96} \\
&= \frac{1}{2}
\end{aligned}$$

On peut vérifier que la méthode est bien d'ordre 3, puis d'ordre 4...

Implémentation de la méthode RK_{34} en C

Dans cette partie, nous allons développer un programme C pour pouvoir calculer la suite de points approximant la solution de (S). On adoptera d'abord une démarche structurée pour écrire en pseudo-langage toutes les opérations qui seront nécessaires au bon fonctionement de la méthode. Ensuite, nous nous intéresserons au coût (en terme d'opérations) engendré par tous ces calculs. Puis, nous parlerons rapidement des erreurs d'approximation machine qui viennent s'ajouter aux erreurs générées par la méthode et qui seront à prendre en compte dans la suite de ce projet. Enfin, nous présenterons rapidement le programme réalisé.

2.1 L'algorithme

On se donne d'emblée un tableau constant $rk34$ de taille 6×6 qui contient tout les coefficients de la méthode (i.e. le tableau de Butcher présenté précédemment).

Nous avons vu que la méthode RK_{34} sert à utiliser un pas adaptatif pour contrôler l'erreur de consistance ; ainsi on peut affirmer que le nombre de points générés par la méthode ne peut pas être « prédit » et dépendra de la fonction f et du seuil choisi. On peut donc remarquer que trois boucles imbriquées seront nécessaires au bon fonctionnement de la méthode :

1. Une boucle principale indéterministe dont la condition d'arrêt sera de savoir si l'abscisse x_k calculée a dépassé la borne b de l'intervalle $[a, b]$.
2. Une boucle déterministe que l'on fera dépendre d'une variable i, dans laquelle les points intérmédiaires $(x_{k,i}, y_{k,i})$ seront calculés.
3. Une dernière boucle déterministe qui dépendra d'une variable j, à l'intérieur de laquelle se déroulera le calcul d'un $y_{k,i}$ (i fixé).

Notre première idée était de stocker les points calculés (i.e. les (x_k, y_k)) dans un tableau au fur et à mesure de l'exécution du programme. On a vu que, si l'on veut contrôler l'erreur, il est impossible de « prédire » la taille de ce tableau à l'avance. Nous avons donc, dans un premier temps, réalisé un programme en Fortran en allouant dynamiquement ce tableau. Le problème est que cette façon de concevoir la méthode est couteuse en mémoire et en complexité (il faut, à chaque fois que l'on dépasse la taille initiale du tableau, réallouer un espace mémoire plus grand et déplacer toutes les données). Nous nous sommes alors aperçus qu'il ne nous était pas nécéssaire de conserver les points calculés en mémoire, mais plutôt de les sauvegarder au fur et à mesure dans un fichier, ce qui est moins couteux.

Il nous a donc semblé logique d'utiliser des variables locales x et y chargées de contenir respectivement l'abscisse et l'ordonnée du point (x_k, y_k) en cours de traitement, puis de les sauvegarder une fois le calcul et l'affectation de x_{k+1} et y_{k+1} effectué. Sachant qu'il est inutile de conserver en mémoire les points intérmédiaires $(x_{k,i}, y_{k,i})$, nous avons aussi utilisé deux variables locales $xtmp$ et $ytmp$ pour les stocker au fur et à mesure. Pour limiter le nombre d'appels de la fonction f, nous avons utilisé un tableau fct chargé de conserver les valeurs des $f(x_{k,i}, y_{k,i})$ en mémoire (à chaque nouvelle itération dans la boucle principale, ces valeurs laissent place aux nouvelles).

Pour simplifier l'algorithme, nous avons affiché au fur et à mesure les points (x_k, y_k), mais dans notre programme C ces points sont sauvegardés dans un fichier (comme le pas et l'erreur).

```
Procédure rungeKutta34 (E a, b, y0, seuil : Réel, f : Fonction(x,y : Réel) : Réel )
   Var  x, y, h, xtmp, ytmp, phi3, phi4, erreur, e : Réel, fct : Tableau[1...4] de Réel, i, j :
        Entier
        Constante rk34 = Tableau[1...6][1...6] de Réel
        Constante NINIT = 10000
Début
   x ← a
   y ← y0
   h ← (b-a)/(NINIT)
   afficher(x,y)
   Tant que (x < b) faire
      xtmp ← 0
      ytmp ← 0
      phi3 ← 0
      phi4 ← 0
      Pour i ←1 à 4 faire
         xtmp ← x + h*rk34[i][1]
         ytmp ← y
         Pour j ←2 à i faire
            ytmp ← ytmp + rk34[i][j]*fct[j-1]
         FinPour
         ytmp ← y + h*ytmp
         fct[i] ← f(xtmp, ytmp)
         phi3 ← phi3 + rk34[5][i+1]*fct[i]
         phi4 ← phi4 + rk34[6][i+1]*fct[i]
      FinPour
      x ← x + h
      y ← y + h*phi3
      phi4 ← phi4 + rk34[6][6]*f(x,y)
      erreur ← h*(phi4 - phi3)
      e ← abs(erreur/seuil)
      Si (e > 1) et (e > 0) alors
         h ← 0.9*h*(1/e)^1/4
      Sinon
         Si (e < 0.9) et (e > 0) alors
            h ← 0.9*h*(1/e)^1/4
         FinSi
         Si (x ≤ b) alors
            afficher(x,y)
         FinSi
      FinSi
   FinTantQue
Fin
```

FIGURE 2.1 – L'algorithme de la méthode RK_{34}

2.2 La complexité

Comme nous l'avons spécifié précédemment, on ne peut pas prédire le nombre d'itérations réalisées dans la boucle principale sans connaitre la fonction f et le seuil rentré par l'utilisateur. On appellera donc $\alpha = \alpha_{f,seuil}$ ce nombre d'itérations. On néglige le cout engendrée par l'affectation et on considérera l'appel d'une fonction comme étant une opération (ce qui est, normalement, plus couteux).

Dans la boucle principale, on compte :

— un certain nombre d'opérations dans la boucle i : n_i ;
— une somme pour le calcul de x ;
— une somme et une multiplication pour le calcul de y ;
— une somme, une multiplication et un appel de fonction pour la fin du calcul de $phi4$, puis une multiplication et une somme pour le calcul de $erreur$;
— une divison pour le calcul de e ;
— on suppose que l'on change h à chaque fois et donc que l'on effectue deux multiplication et un appel de fonction (pour la puissance 1/4).

Donc $n_i + 12$ opérations.

Dans la boucle i, on a :

— $i - 1$ sommes et $i - 1$ multiplications dans la boucle j ;
— une somme et une multiplication pour le calcul de $xtmp$;
— une somme et une multiplication pour le calcul de $ytmp$;
— un appel de la fonction f ;
— une somme et une multiplication pour le calcul de $phi3$;
— une somme et une multiplication pour le calcul de $phi4$.

Donc $n_i = \sum_{i=1}^{4}(2i + 7) = 48$ opérations.

On a donc 60 opérations à chaque exécution de la boucle principale, donc 60α opérations en tout. Par exemple, dans le cas de la même méthode à pas constant, on a $60n$ opérations pour n points à calculer.

2.3 Présentation du programme C

Nous avons implémenté la méthode en langage C, en prenant soin de copier les valeurs des points, de l'erreur de consistance et du pas dans des fichiers dans un dossier *data* pour pouvoir les utiliser. De plus, nous avons aussi codé la même méthode à pas constant pour pouvoir comparer les résultats plus tard. Dans le programme suivant on a choisi $y' = y$, il suffit de modifier la valeur de retour de f pour pouvoir utiliser notre programme sur une autre équation différentielle. Si nous avions eu plus de temps, nous aurions souhaité passer la fonction f en paramètre du programme en utilisant une librairie d'évaluation d'expression mathématiques (comme muParser, par exemple).

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NINIT 10000

/* Fonction f
Entrées :
    x : la variable scalaire
    y : une application

```

```
11  Sortie :
12      Un réel */
13  double f(double x, double y){
14      return y;
15  }
16
17  /* Procédure rungeKutta34
18  Entrées :
19      a, b : l'intervalle [a,b]
20      y0 : la condition de Cauchy
21      seuil : le seuil fixé pour l'erreur de consistance
22      f : la fonction f(x,y)
23
24  Sortie :
25      Les x_k, y_k dans le fichier pointsRK34
26      Les erreurs de consistances dans le fichier erreurs
27      Le pas utilisé dans le fichier pas */
28  void rungeKutta34(double a, double b, double y0, double seuil, double ...
        (*f)(double,double)){
29      /* Déclarations
30      Variables de type simple
31          x : x_k que l'on affiche au fur et à mesure
32          y : y_k que l'on affiche au fur et à mesure
33          h : le pas h_k
34          xtmp : x_{k,i}
35          ytmp : y_{k,i}
36          phi3 : la fonction phi de la méthode d'ordre 3
37          phi4 : la fonction phi de la méthode d'ordre 4
38          erreur : l'erreur de consistance
39          e : erreur/seuil
40          i, j : pour les boucles
41          ficPoints, ficErreurs, ficPas : pour les fichiers
42
43      Tableaux
44          fct : un tableau pour limiter les appels de la fonction f (couteux)
45          rk34 : un tableau constant à double dimension contenant tous les ...
                coefficients de la méthode RK34 */
46      double x, y, h, xtmp, ytmp, phi3, phi4, erreur, e;
47      double fct[4];
48      double rk34[6][6];
49      int i, j;
50      FILE *ficPoints, *ficErreurs, *ficPas;
51
52      /* Initialisation du tableau constant de la méthode RK34
53      On initialise que les variables qui nous seront utiles */
54      rk34[0][0] = 0.;
55      rk34[1][0] = 2./7.;
56      rk34[2][0] = 4./7.;
57      rk34[3][0] = 6./7.;
58      rk34[1][1] = 2./7.;
59      rk34[2][1] = -8./35.;
60      rk34[3][1] = 29./42.;
61      rk34[4][1] = 1./6.;
62      rk34[5][1] = 11./96.;
63      rk34[2][2] = 4./5.;
64      rk34[3][2] = -2./3.;
65      rk34[4][2] = 1./6.;
66      rk34[5][2] = 7./24.;
67      rk34[3][3] = 5./6.;
68      rk34[4][3] = 5./12.;
69      rk34[5][3] = 35./96.;
70      rk34[4][4] = 1./4.;
```

```
71      rk34[5][4] = 7./48.;
72      rk34[5][5] = 1./12.;
73
74      /* Ouverture des fichiers */
75      ficPoints = fopen("../data/pointsRK34", "w");
76      ficErreurs = fopen("../data/erreursRK34", "w");
77      ficPas = fopen("../data/pas", "w");
78
79      /* Initialisation des variables */
80      x = a;
81      y = y0;
82      h = (b-a)/(NINIT);
83
84      /* Écriture du point initial et du pas initial */
85      fprintf(ficPoints, "%.50f %.50f\n", x, y);
86      fprintf(ficPas, "%.50f\n", h);
87
88      /* Boucle principale : parcours de l'intervalle [a,b] */
89      while(x<b){
90          /* Initialisation des variables */
91          xtmp = 0.;
92          ytmp = 0.;
93          phi3 = 0.;
94          phi4 = 0.;
95
96          /* Boucle pour le calcul points intermédiaires */
97          for(i = 0 ; i < 4 ; i++){
98              /* Calcul de x_{k,i} */
99              xtmp = x + h*rk34[i][0];
100
101             /* Calcul de y_{k,i} */
102             for(j = 1 ; j ≤ i ; j++){
103                 ytmp += rk34[i][j]*fct[j-1];
104             }
105             ytmp = y + h*ytmp;
106
107             /* Appel de la fonction f puis stockage dans un tableau */
108             fct[i] = (*f)(xtmp, ytmp);
109
110             /* Calcul de phi ordre 3 et 4 */
111             phi3 += rk34[4][i+1]*fct[i];
112             phi4 += rk34[5][i+1]*fct[i];
113         }
114
115         /* Calcul de x_{k+1} et y_{k+1} */
116         x += h;
117         y += h*phi3;
118
119         /* Calcul de l'erreur de consistance */
120         phi4 += rk34[5][5]*((*f)(x,y));
121         erreur = fabs(h*(phi4 - phi3));
122
123         /* Écriture de l'erreur de consistance */
124         fprintf(ficErreurs, "%.50f\n", erreur);
125
126         /* Calcul du coefficient pour calculer le h optimal */
127         e = erreur/seuil;
128
129         if((e > 1) && (e > 0)){
130             /* Calcul de h optimal */
131             h = 0.9*h*pow(1./e, 1./4.);
132         }
```

```
133         else {
134             if ((e < 0.9) && (e > 0)){
135                 /* Calcul de h optimal */
136                 h = 0.9*h*pow(1./e, 1./4.);
137             }
138             if(x ≤ b){
139                 /* Écriture du point si il respecte la condition du seuil */
140                 fprintf(ficPoints, "%.50f %.50f\n", x, y);
141             }
142         }
143 
144         /* Écriture du pas */
145         fprintf(ficPas, "%.50f\n", h);
146     }
147 
148     /* Fermeture des fichiers */
149     fclose(ficPoints);
150     fclose(ficErreurs);
151     fclose(ficPas);
152 }
153 
154 /* Procédure rungeKutta3
155     Même méthode à pas constant */
156 void rungeKutta3(double a, double b, double y0, int n, double (*f)(double,double)){
157     double x, y, h, xtmp, ytmp, phi3, phi4, erreur;
158     double fct[4];
159     double rk34[6][6];
160     int i, j, k;
161     FILE *ficPoints, *ficErreurs;
162 
163     /* Initialisation du tableau constant de la méthode RK34
164     On initialise que les variables qui nous seront utiles */
165     rk34[0][0] = 0.;
166     rk34[1][0] = 2./7.;
167     rk34[2][0] = 4./7.;
168     rk34[3][0] = 6./7.;
169     rk34[1][1] = 2./7.;
170     rk34[2][1] = -8./35.;
171     rk34[3][1] = 29./42.;
172     rk34[4][1] = 1./6.;
173     rk34[5][1] = 11./96.;
174     rk34[2][2] = 4./5.;
175     rk34[3][2] = -2./3.;
176     rk34[4][2] = 1./6.;
177     rk34[5][2] = 7./24.;
178     rk34[3][3] = 5./6.;
179     rk34[4][3] = 5./12.;
180     rk34[5][3] = 35./96.;
181     rk34[4][4] = 1./4.;
182     rk34[5][4] = 7./48.;
183     rk34[5][5] = 1./12.;
184 
185     /* Ouverture des fichiers */
186     ficPoints = fopen("../data/pointsRK3", "w");
187     ficErreurs = fopen("../data/erreursRK3", "w");
188 
189     /* Initialisation des variables */
190     x = a;
191     y = y0;
192     h = (b-a)/n;
193 
194     /* Écriture du point initial */
```

```
195     fprintf(ficPoints, "%.50f %.50f\n", x, y);
196
197     /* Boucle principale : parcours de l'intervalle [a,b] */
198     for (k = 0 ; k < n ; k++){
199         /* Initialisation des variables */
200         xtmp = 0.;
201         ytmp = 0.;
202         phi3 = 0.;
203         phi4 = 0.;
204
205         /* Boucle pour le calcul points intermédiaires */
206         for(i = 0 ; i < 4 ; i++){
207             /* Calcul de x_{k,i} */
208             xtmp = x + h*rk34[i][0];
209
210             /* Calcul de y_{k,i} */
211             for(j = 1 ; j ≤ i ; j++){
212                 ytmp += rk34[i][j]*fct[j-1];
213             }
214             ytmp = y + h*ytmp;
215
216             /* Appel de la fonction f puis stockage dans un tableau */
217             fct[i] = (*f)(xtmp, ytmp);
218
219             /* Calcul de phi ordre 3 et 4 */
220             phi3 += rk34[4][i+1]*fct[i];
221             phi4 += rk34[5][i+1]*fct[i];
222         }
223
224         /* Calcul de x_{k+1} et y_{k+1} */
225         x += h;
226         y += h*phi3;
227
228         /* Calcul de l'erreur de consistance */
229         phi4 += rk34[5][5]*((*f)(x,y));
230         erreur = fabs(h*(phi4 - phi3));
231
232         /* Écriture de l'erreur de consistance */
233         fprintf(ficErreurs, "%.50f\n", erreur);
234
235         /* Écriture du point */
236         fprintf(ficPoints, "%.50f %.50f\n", x, y);
237     }
238
239     /* Fermeture des fichiers */
240     fclose(ficPoints);
241     fclose(ficErreurs);
242 }
243
244
245 int main(int argc, char* argv[]){
246     double a, b, y0, seuil;
247     int n;
248
249     /* Récupération des variables */
250     printf("a : ");
251     scanf("%lf", &a);
252     printf("b : ");
253     scanf("%lf", &b);
254     printf("y0 : ");
255     scanf("%lf", &y0);
256     printf("seuil pas variable : ");
```

```
257       scanf("%lf", &seuil);
258       printf("nombre d'itérations pas constant : ");
259       scanf("%d", &n);
260
261       /* Appel de la procédure rungeKutta34 */
262       rungeKutta34(a, b, y0, seuil, f);
263
264       /* Appel de la procédure rungeKutta3 */
265       rungeKutta3(a, b, y0, n, f);
266
267       return EXIT_SUCCESS;
268   }
```

2.4 Précision machine et approximations

Il est nécessaire de souligner que les solutions obtenues ne sont pas toujours exactes mais approximées. En effet, outre les erreurs dûes aux methodes de résolutions, il existe également des erreurs dues à la machine. Nos ordinateurs disposent d'une capacité de représentation finie : d'une part, certains nombres sont trop petits pour être représentés, et de la même manière il existe des nombres qui dépassent cette capacité. Dans ce cas là, on parlera alors respectivement d'*underflow* et d'*overflow*. Un *overflow* provoquera la représentation du nombre comme *infinity* et arrêtera directement l'éxecution, tandis qu'un *underflow* sera arrondi à zéro.

D'autre part la représentation informatique des réels repose sur une expression binaire (i.e. en base 2) de ces nombres. Ainsi, des nombres ayant une valeur exacte dans une base peut avoir une valeur arrondie dans une autre. La machine effectue donc des approximations afin de pouvoir les représenter. On a l'exemple de 0.1 qui en décimal aura une expression finie tandis qu'en binaire aura une représentation périodique infini.

Enfin il est important de souligner que l'arrondi d'un réel dans un programme donné peut amener à une accumulation d'erreurs. Une erreur d'abord non significative aura de grandes répercutions du fait du nombre important de fois où elle est réalisée, on assistera alors à une dégradation progressive du résultat.

Application de la méthode RK_{34} sur des exemples

Nous allons ici appliquer la méthode RK_{34} et la méthode RK_3 sur des exemples concrets pour tester leur efficacité. On se donnera des EDO resolvables pour pouvoir calculer l'erreur commise par la méthode : $(e_k)_{k\in[\![0,n]\!]} = (|y_k - y(x_k|)_{k\in[\![0,n]\!]}$.

3.1 Quelques exemples intéressants

3.1.1 Exponentielle

On se donne $f(x, y) = y$.

Résolution directe

On a :

$$\begin{aligned} y' &= y \\ &\Leftrightarrow \frac{dy}{dt} = y \end{aligned}$$

On suppose $y \neq 0$,

$$\begin{aligned} \frac{dy}{y} = dt \Leftrightarrow \int_a^x \frac{dy}{y} &= \int_a^x dt \\ \Leftrightarrow ln(|y|) - ln(|y_0|) &= x - a \\ \Leftrightarrow ln(|\frac{y}{y_0}|) &= x - a \\ \Leftrightarrow \frac{y(x)}{y_0} &= e^{x-a} \\ \Leftrightarrow y(x) &= y_0 e^{x-a} \end{aligned}$$

Application de la méthode

Dans un premier temps , nous avons étudié les valeurs de la solution pour la valeur initiale $y_0 = 1$ puis celles pour $y_0 = 1000$.

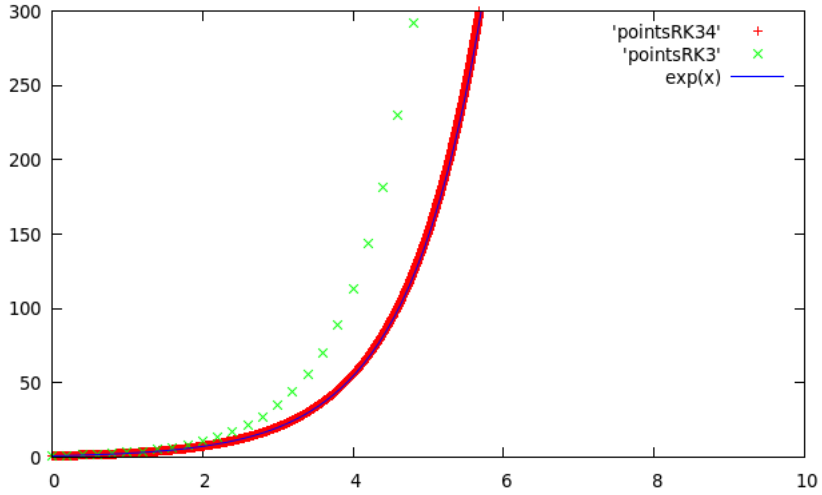


FIGURE 3.1 – Pour $y_0 = 1$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

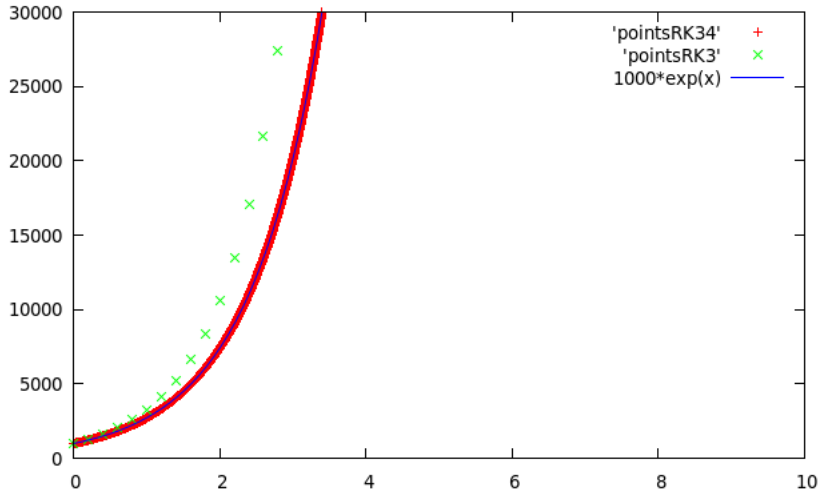


FIGURE 3.2 – Pour $y_0 = 1000$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

Puis, nous nous sommes intéressés à la comparaison de l'erreur de consistance selon la méthode utilisée :

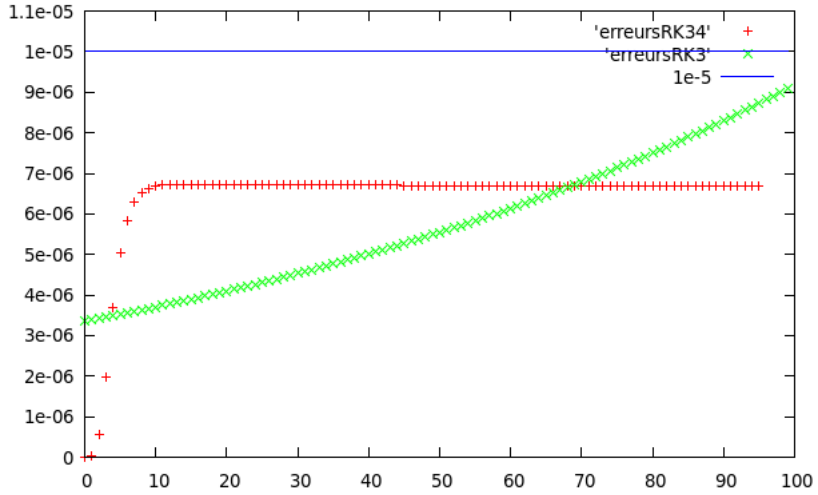


FIGURE 3.3 – Pour $y_0 = 1$ sur l'intervalle $[0, 1]$, seuil de 10^{-5}, pas de 100

On remarque que l'erreur de consistance croit pour la méthode à pas constant, alors qu'elle se stabilise pour la méthode à pas adaptatif. On remarque aussi que l'erreur de consistance est bien en dessous du seuil fixé, voir trop en dessous. Cela est du au facteur 0.9 que l'on s'est donné par sécurité dans le calcul du $h_{k_{opt}}$. En enlevant cette sécurité, on a l'erreur de consistance suivante :

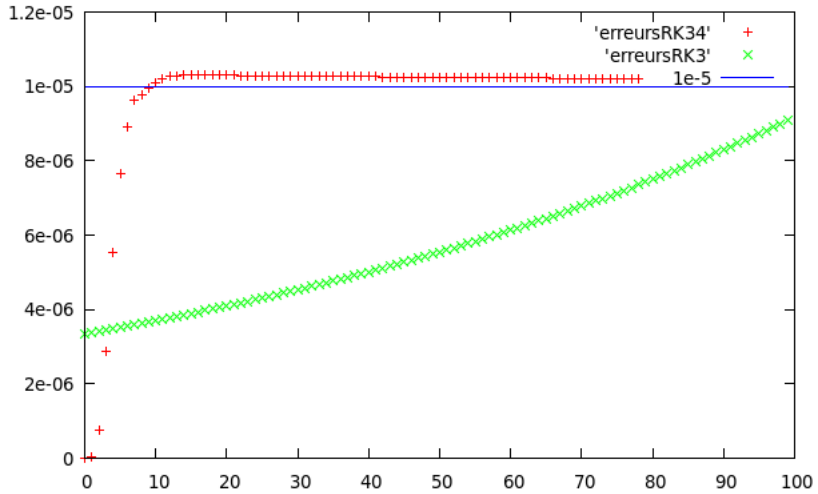


FIGURE 3.4 – Pour $y_0 = 1$ sur l'intervalle $[0, 1]$, seuil de 10^{-5}, pas de 100

Cette erreur est clairement au dessus du seuil fixé, on voit donc bien l'intérêt de ce facteur de sécurité.

Nous avons l'erreur réelle suivante pour la méthode RK_{34}, qui croit fortement et dépasse clairement le seuil de l'erreur de consistance :

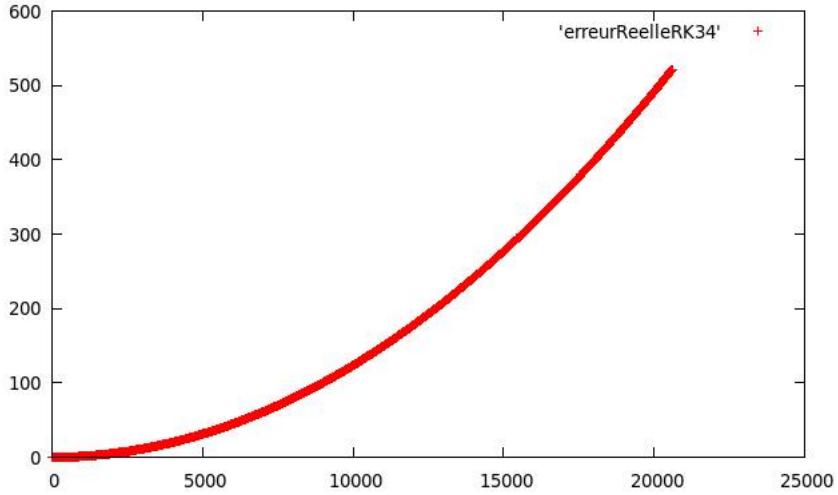


FIGURE 3.5 – Pour $y_0 = 1$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

Pour la méthode RK_3, l'erreur explose :

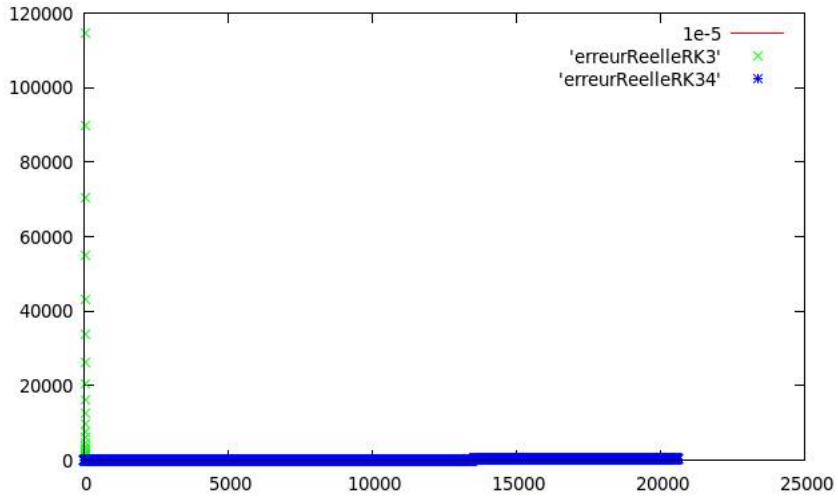


FIGURE 3.6 – Pour $y_0 = 1$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

Comparaison des résultats

On a pu montré ici que la méthode RK_{34} était bien plus efficace dans le sens où elle contient bien mieux l'erreur de consistance et l'erreur réelle. On remarque tout de même une perte d'efficacité de la méthode pour une fonction dont la croissance est aussi démusurée que celle de l'exponentielle.

3.1.2 Exponentielle décroissante

On s'intéresse ici à $f(x, y) = -y$.

Résolution directe

De la même manière que précédemment, nous retrouvons :

$$y(x) = y_0 e^{-(x-a)}$$

Application de la méthode

Ici, nous utilisons les mêmes valeurs initiales que précédemment :

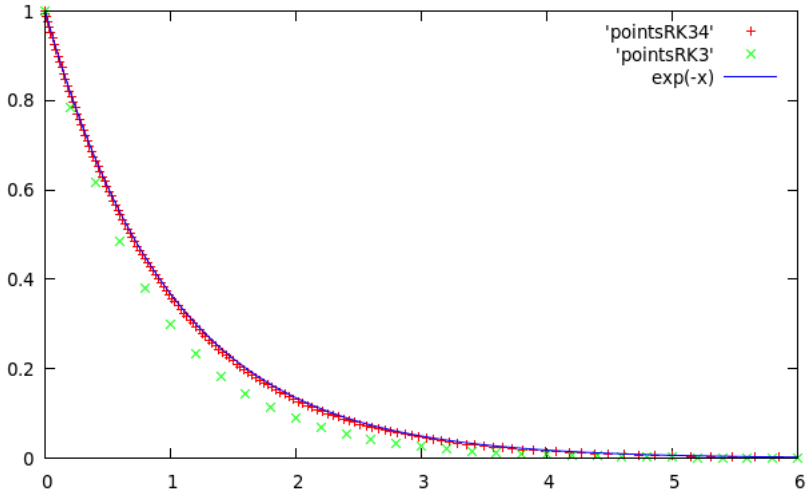


FIGURE 3.7 – Pour $y_0 = 1$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

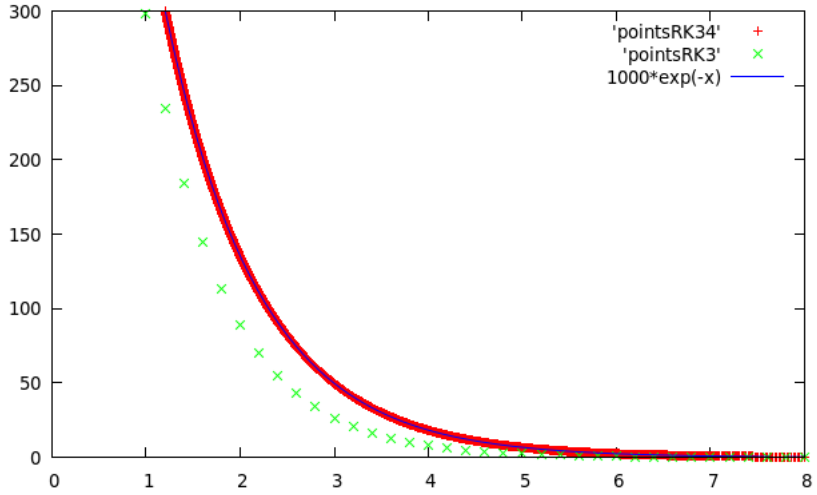


FIGURE 3.8 – Pour $y_0 = 1000$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

Ce qui donne, pour l'erreur de consistance :

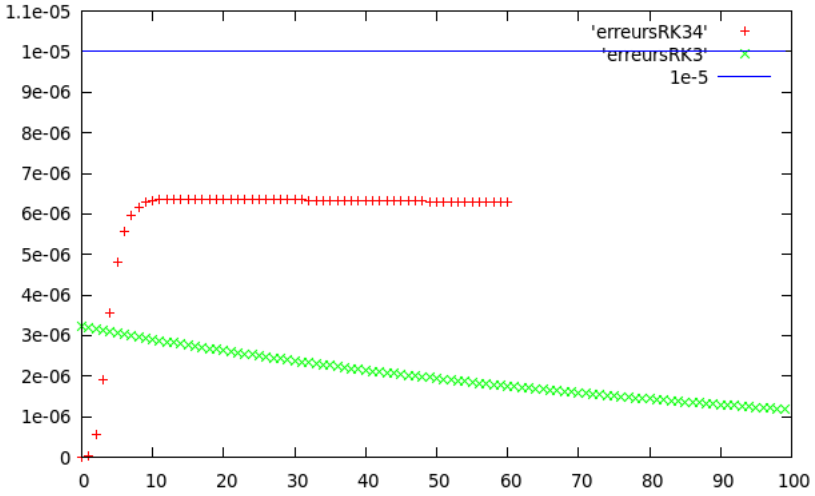


FIGURE 3.9 – Pour $y_0 = 1$ sur l'intervalle $[0, 1]$, seuil de 10^{-5}, pas de 100

On remarque que l'erreur de consistance décroit pour la méthode à pas constant, alors qu'elle se stabilise pour la méthode à pas adaptatif.

Concernant l'erreur réelle, on a :

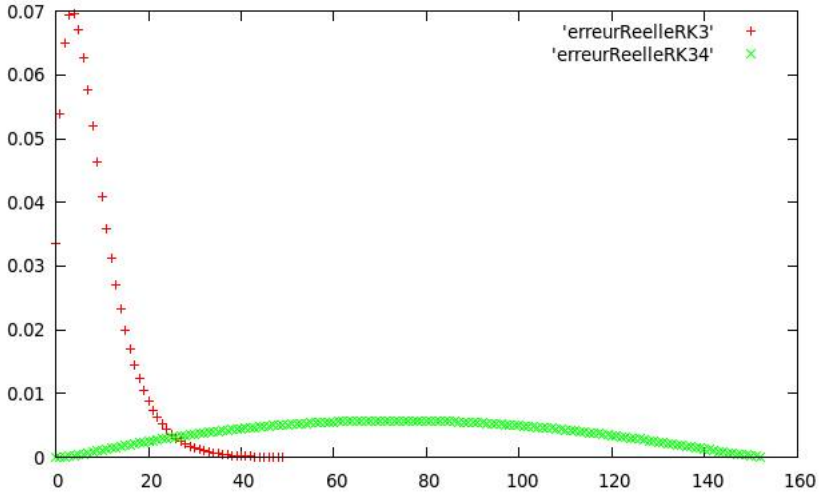


FIGURE 3.10 – Pour $y_0 = 1$ sur l'intervalle $[0, 10]$, seuil de 10^{-5}, pas de 50

Comparaison des résultats

On remarque que l'erreur réelle est ici mal maitrisée par la méthode adaptatif : par un souci d'optimisation, celle-ci met un certain temps à atteindre une erreur aussi faible que celle à pas constant. Mais, la méthode à pas constant n'est pas très intéressante car l'erreur est très élevée puis décroit, alors que RK_{34} nous offre un certain contrôle.

3.1.3 Ça oscille ici

On se propose d'étudier $f(x, y) = \frac{1}{x^2} sin\left(\frac{1}{x}\right)$.

Résolution directe

On commence par remarquer que :

$$\left(cos(\frac{1}{x})\right)' = \frac{1}{x^2} sin\left(\frac{1}{x}\right)$$

D'où :

$$y(x) = cos\left(\frac{1}{x}\right) - cos\left(\frac{1}{a}\right) + y_0$$

Application de la méthode

On choisit $y_0 = cos(\frac{1}{a})$. Nous avons étudié la résolution cette équation différentielle pour des seuils plus ou moins petit (i.e. 10^{-5} et 10^{-10}).

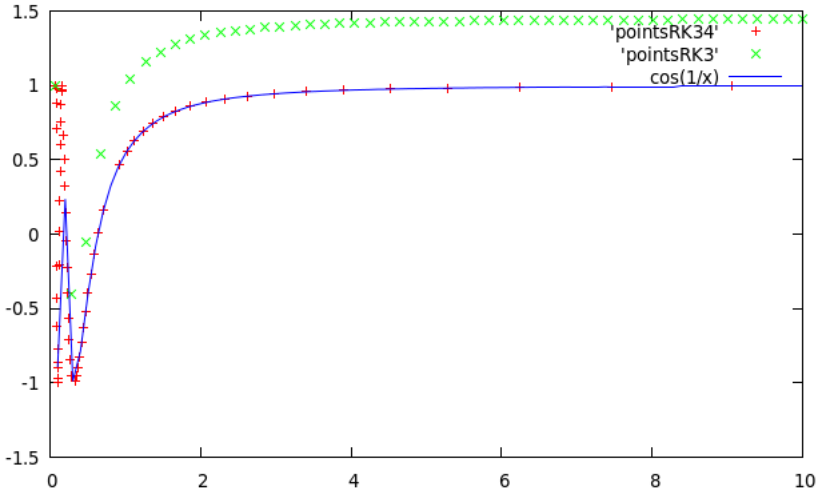


FIGURE 3.11 – Pour l'intervalle $[0.08, 10]$, seuil de 10^{-5}, pas de 50

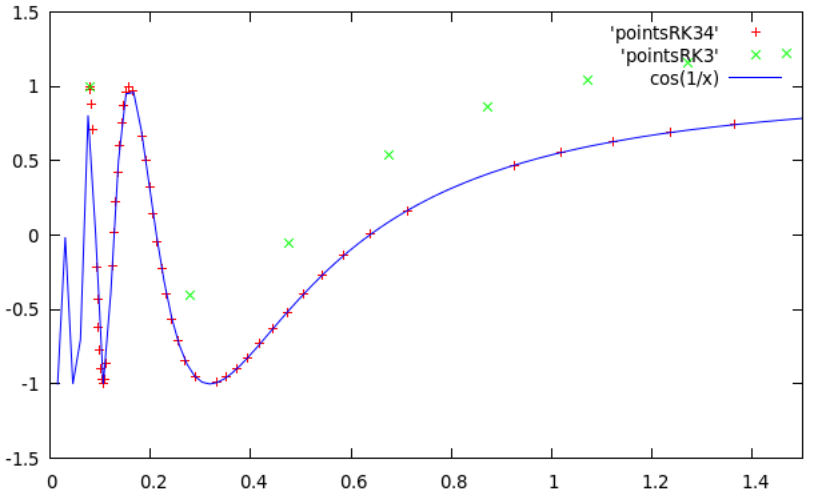


FIGURE 3.12 – La même image zommée

On voit ici clairement les limites de la méthode à pas adaptatif, qui réagit assez mal à ces oscillations aussi violentes. La méthode à pas constant est toujours aussi limitée, sauf si l'on se donne un pas très élevé, ce qui n'est pas très optimisé et difficile à prévoir. On peut toutefois voir qu'en prenant une erreur de consistance plus faible, la méthode reste correcte :

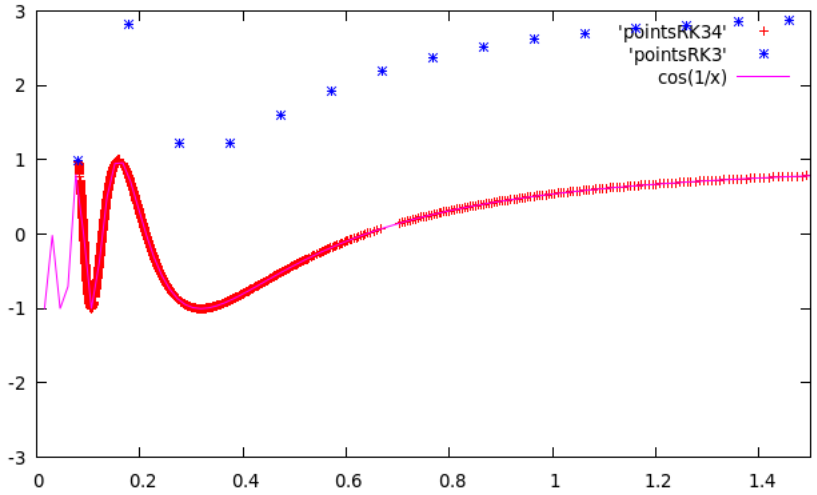


FIGURE 3.13 – Pour l'intervalle $[0.08, 10]$, seuil à 10^{-10}, pas de 50

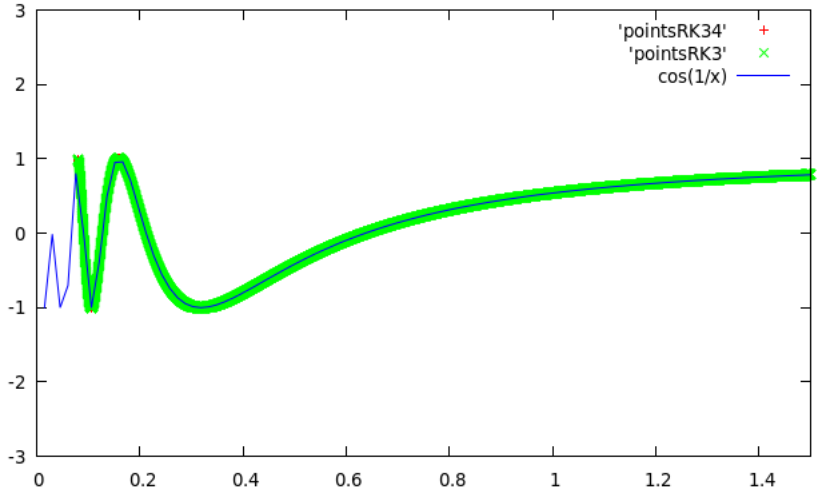


FIGURE 3.14 – Pour l'intervalle $[0.08, 10]$, seuil à 10^{-10}, pas de 100000

On remarque ensuite que à partir d'une certaine valeur de a critique, notre courbe se décale complétement de la solution réelle pour un seuil de 10^{-10} :

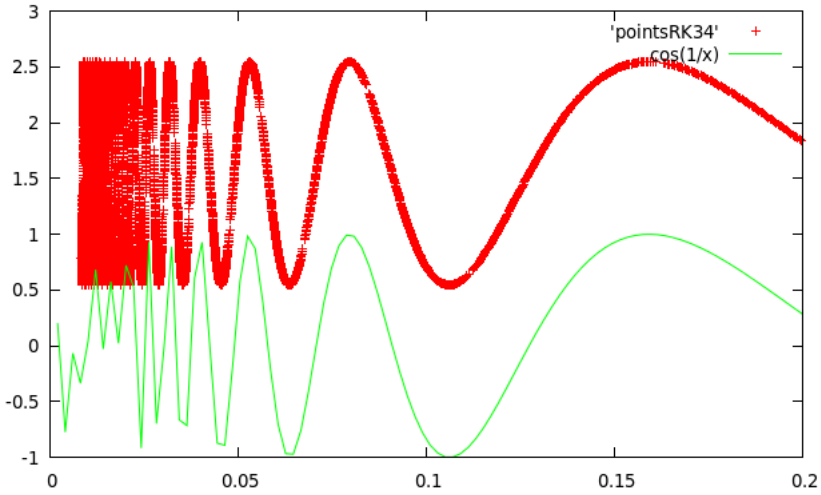


FIGURE 3.15 – Valeur critique à $a = 0.05$ pour RK34

Comparaison des résultats

Nous avons ici vu que le pas adaptatif garantit une adaptation du pas à l'erreur choisie : si la solution oscille beaucoup, alors il y aura beaucop de points. Pour un pas constant, il faut être capable de prévoir ce comportement pour la solution, ce qui n'est pas évident.

3.1.4 Retour aux racines

On se donne $f(x, y) = \sqrt{x}y^2$.

Résolution directe

On a donc :

$$\begin{aligned}\frac{dy}{y^2} = \sqrt{t}dt \Leftrightarrow \int_a^x \frac{dy}{y^2} &= \int_a^x \sqrt{t}dt \\ \Leftrightarrow -\frac{1}{y} &= \frac{2}{3}(x\sqrt{x} - a\sqrt{a}) - \frac{1}{y_0} \\ \Leftrightarrow y(x) &= \frac{1}{\frac{2}{3}(a\sqrt{a} - x\sqrt{x}) + \frac{1}{y_0}}\end{aligned}$$

Application de la méthode

On choisit $y_0 = 1$.

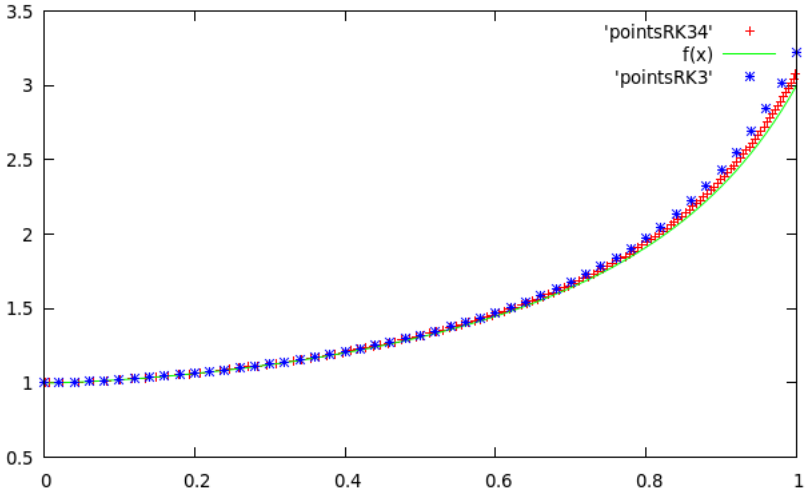


FIGURE 3.16 – Pour l'intervalle $[0, 1]$, seuil à 10^{-5}, pas de 50

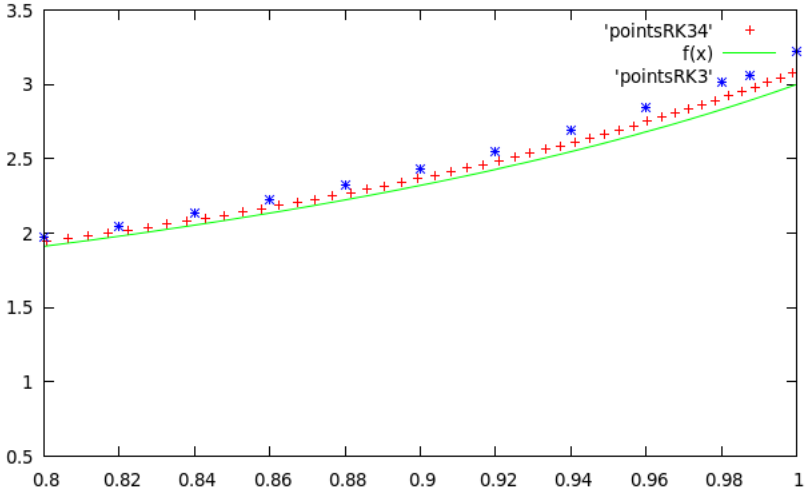


FIGURE 3.17 – La même image zoomée

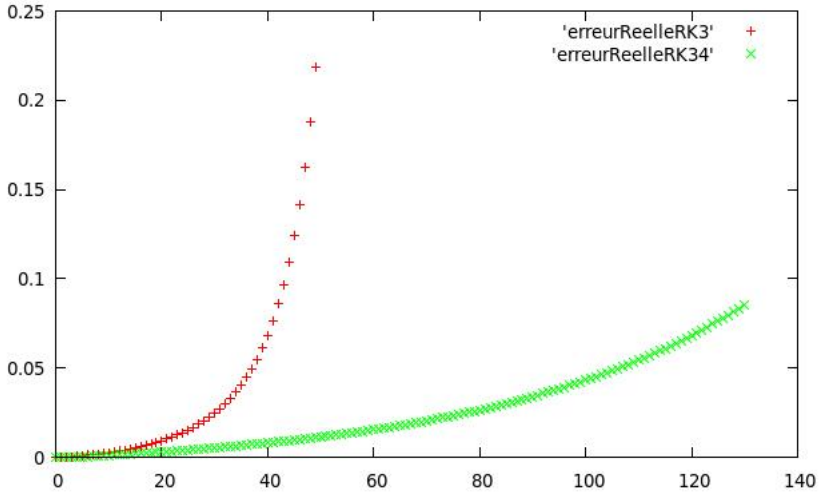


FIGURE 3.18 – Erreur réelle

3.1.5 Normal

Pour retrouver les tables de la loi normale, nous avons décidé de prendre l'exemple $f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

Application de la méthode

On choisit l'intervalle d'étude $[-4,4]$ avec $y_0 = \Phi(-4)$.

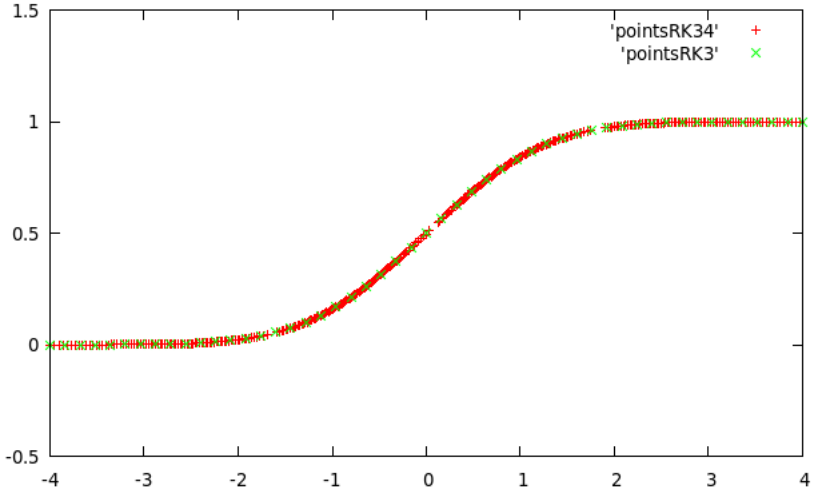


FIGURE 3.19 – Loi normale, seuil de 10^{-5}, pas de 50 pour la méthode RK_3

On retrouve bien le graphe de Φ. Les valeurs données par la méthode correspondent à 10^{-3} près aux valeurs des tables, ce qui est une bonne approximation.

3.2 Bilan concernant la méthode RK_{34}

Tout d'abord expliquons les exemple étudiés. Nous avons dans un premier temps décidé de tester nos programmes sur des équations différentielles faciles à résoudre afin de vérifier nos résultats. Ensuite nous nous sommes rendus compte que l'erreur variait si la solution était convergente et divergente. C'est pourquoi nous avons utilisé les cas $f(x, y) = y$ et $f(x, y) = -y$. Par la suite, nous avons voulons étudier le cas dune fonction qui oscillait rapidement, typiquement $cos\left(\frac{1}{x}\right)$. Ensuite , nous avons cherché une équation différentielle qui dépendait des deux paramètres x et y. En dernier lieu, nous avions pour objectif de trouver une application concrète de cette méthode. Nous avons alors eu l'idée de la loi normale.

Finalement, après étude de plusieurs cas différent, il apparaît que la méthode de résolution de Runge Kutta 34 à pas variable nous donne une meilleure approximation que la méthode de Runge Kutta 3 dans presque tout les cas de figure. En plus, elle permet de limiter les calculs en adoptant un pas optimal et s'adapte à la fonction étudiée. Si nous avions voulu obtenir les même résultats à pas constant, il aurait fallu trouver le pas vérifiant $\max_k \epsilon_k(h) < seuil$.

Conclusion

Ce projet nous a permis d'étudier une méthode de résolution des équations différentielles ordinaires : la méthode RK_{34} à pas adaptatif. Nous avons implémenté cette méthode d'abord en fortran à pas constant, puis à pas variable avec des tableaux dynamique en fortran et en C. Après avoir revu notre code afin d'éviter l'utilisation de tableaux dynamiques, notre programme nous a offert à la fois de bonnes approximations des points discrétisés, mais aussi une optimisation en coût et en mémoire lors de l'exécution.

Nous avons testé la pertinence de notre programme en comparant nos résultats aux valeurs exactes et nous nous sommes alors rendu compte que ces derniers étaient très proches des valeurs attendues, à l'exception des fonctions admettant des valeurs critiques au voisinage de certains points (par exemple $cos\left(\frac{1}{x}\right)$). L'étude de ces exemples de fonctions convergentes, divergentes et oscillantes avec les méthodes RK_{34} et RK_3 nous a montré l'intérêt du pas adaptatif et ses limites.

Tous les deux passionnés par ce sujet, nous avons apprécié ce projet, même s'il a été difficile de s'organiser. Nous aurions aimé ajouter certains éléments à ce projet :

— une étude supplémentaire permettant une généralisation de la methode de $RK_{pp'}$, où l'utilisateur pourrait entrer les coefficients lui même ;
— passer la fonction $f(x, y)$ en paramètre du programme ;
— diversifier et approfondir les exemples ;
— généraliser le domaine d'étude au cas $\mathbb{R}^m$.

Bibliographie

[1] *Nicolas Forcadel*, *Cours d'Analyse Numérique*, Institut National des Sciences Appliquées de Rouen.

[2] *Andre Draux*, *Cours d'Analyse Numérique*, Institut National des Sciences Appliquées de Rouen.

[3] *Olivier Mgbra*, *Cours sur les méthodes de résolution des EDO*, Youtube.

[4] *Ernst Hairer, Syvert P. Nørsett, Gerhard Wanner*, *Solving Ordinary Differential Equations I – Nonstiff Problems*, Springer, 2008.

[5] *Ernst Hairer, Christian Lubich, Gerhard Wanner*, *Geometric Numerical Integration – Structure-Preserving Algorithms for Ordinary Differential Equations*, Springer.

[6] *John C. Butcher*, *A history of Runge-Kutta methods*, Université d'Auckland (Nouvelle Zélande), 1996.

[7] `http://showard.sdsmt.edu/Math373/_AppliedNumMethodsText_SMH/RK_History.htm`, History of Runge-Kutta methods, (Valide à la date du 05/03/2014).