Josselin Massot
CMAP équipe HPC@Maths

ponio : vous reprendrez bien un peu d'intégrateur en temps ?

Pourquoi ponio ?

- Pas ou peu de bibliothèque d'intégration en temps en C++
  - Boost::odeint, GSL
- Les quelques bibliothèques disponibles ont peu de méthodes
  - Boost:odeint : surtout méthodes hamiltoniennes
  - GSL : méthodes très classiques (RK4, Euler explicite et implicite)
- Ces bibliothèques s'adaptent mal avec n'importe quelle structure de données
  - Utilisation avec samurai pour maillage adaptatif

Pourquoi ponio ?

Objectifs :

- Une bibliothèque d'intégrateurs en temps pour EDO et EDP
- Avoir des intégrateurs de nouvelle génération (ROCK, PIROCK, ESERK)
- Avoir des méthodes classiques de référence (eRK, DIRK)
- Avoir des méthodes adaptées à certains problèmes (Lawson, expRK, méthodes hamiltoniennes)
- Un code open-source pour la communauté
  `https://github.com/hpc-maths/ponio`

ponio n'a pas encore la prétention d'être un équivalent un équivalent de DifferentialEquations.jl

Plan

1. Résolution d'une EDO : problème de Curtiss-Hirschfelder
   - Méthode Runge-Kutta explicite
   - Méthode Runge-Kutta diagonale implicite
   - Méthode de Lawson
   - Méthode exponentielle Runge-Kutta
   - Méthode de splitting
2. Résolution d'une EDP : modèle de Belousov-Zhabotinsky 1d à 3 équations
   - Méthode à stabilité étendue
   - Méthode IMEX un peu spéciale
3. Conclusion

Plan

Problème de Curtiss-Hirschfelder

Problème raide classique :

$$\begin{cases} \dot{y} = k(\cos(t) - y) \\ y(0) = y_0 \end{cases}$$

avec $k = 50$ et $y_0 = 2$

Objectif :

Montrer l'interface de ponio, et sa facilité d'utilisation

Résolution par une méthode eRK

Résolution de

$$\dot{y} = f(t, y)$$

Se fait en lisant le tableau de Butcher

$$\begin{aligned} y^{(i)} &= y^n + \Delta t \sum_j a_{ij} k_j \\ k_i &= f\left(t^n + c_i \Delta t, y^{(i)}\right) \\ y^{n+1} &= y^n + \Delta t \sum_i b_i k_i \end{aligned}$$

Résolution par une méthode eRK

Avec ponio il suffit de définir son problème :

```
double f(double t, double y)
{
  double k = 50;
  return k*(std::cos(t) - y);
}
```

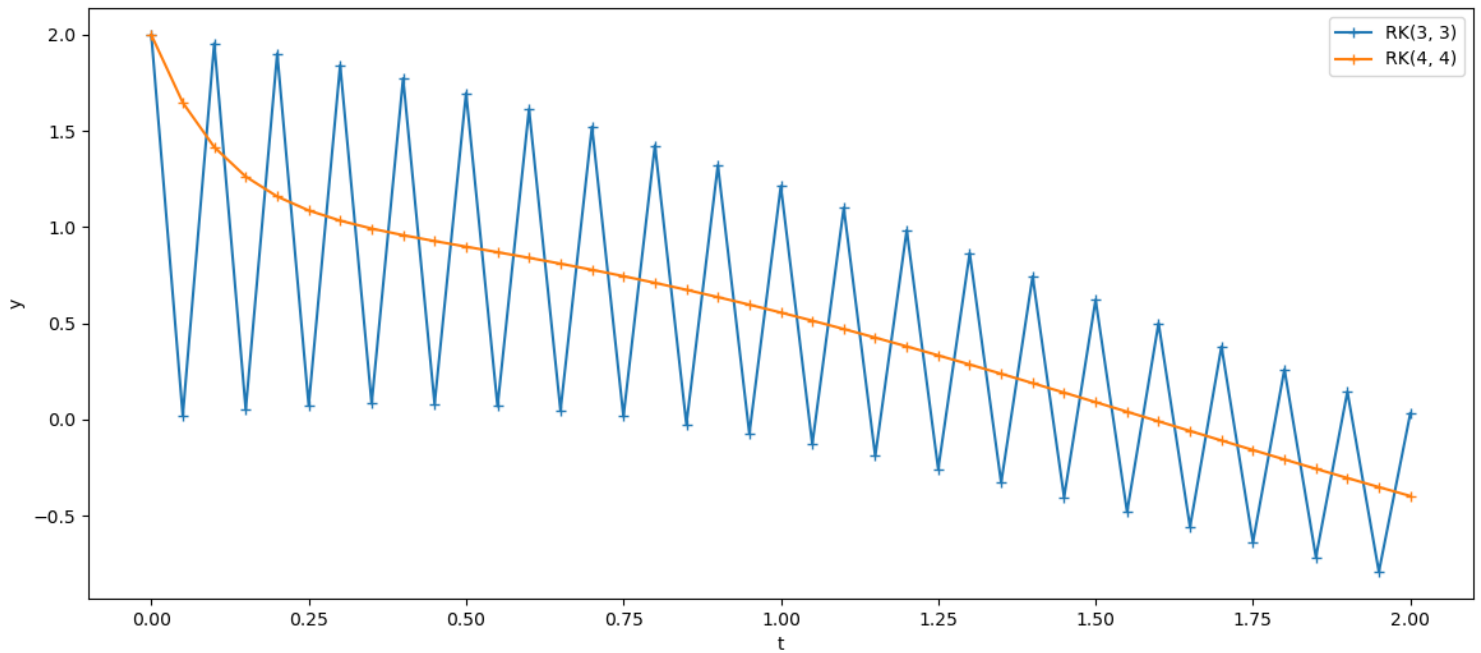
et de faire `solve` !

```
ponio::solve( f, ponio::runge_kutta::rk_44(),
              y0,
              {0., 2.}, dt,
              "output.txt"_fobs
            );
```

Résolution par une méthode eRK

Avec nonic il suffit de définir son problème :

et



Résolution par une méthode DIRK

Même schéma que précédemment, **mais** 1 système non-linéaire à résoudre par étage

Donc une méthode de Newton par étage, il faut une jacobienne (pour le moment pas encore d'estimateur numérique de jacobienne *see you next CANUM in 2 years* ;)

```
double df(double t, double y)
{
  double k = 50;
  return -k;
}
```

Besoin d'une structure pour lier la fonction et sa jacobienne

```
auto pb = ponio::make_implicit_problem(f, df);
```

puis on résout avec la même fonction `ponio::solve` :

```
ponio::solve( pb, ponio::runge_kutta::dirk34(),
              y0,
              {0., 2.}, dt,
              "output.txt"_fobs
            );
```

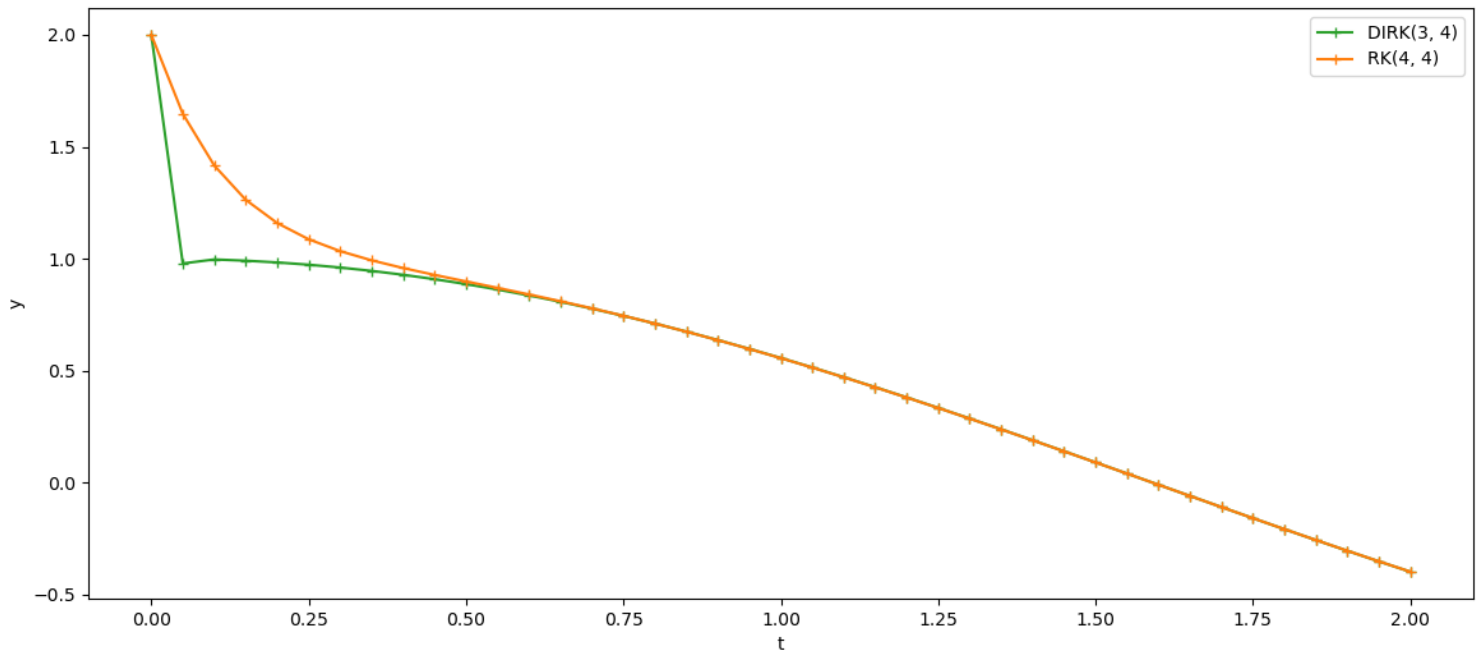
Résolution par une méthode DIRK

Même schéma que précédemment, **mais** 1 système non-linéaire à résoudre par étage

Donc une méthode de Newton par étage, il faut une jacobienne (pour le

m

C



B

au

pu

```
        y0,
        {0., 2.}, dt,
        "output.txt"_fobs
    );
```

Ajout simple de tableaux de Butcher

Il suffit d'ajouter fichier JSON

- génération de code C++ via un script Python
- analyse de la méthode (pour vérifier son ordre)

```
{
    "label": "RK (3,2) best",
    "A": [
        [ "0", "0", "0" ],
        [ "1/2", "0", "0" ],
        [ "0", "1/2", "0" ]
    ],
    "b": [ "0", "0", "1" ],
    "c": [ "0", "1/2", "1/2" ],
    "doi": "10.1016/j.jcp.2020.109688"
}
```

Mais ponio contient sans doute déjà la méthode que vous cherchez (et sans doute plus)

Résolution par une méthode de Lawson

Résolution de

$$\dot{y} = Ly + N(t, y)$$

Changement de variable $z(t) = e^{-Lt}y(t)$, le problème se réécrit sous la forme

$$\dot{z} = e^{-Lt}N(t, e^{-Lt}z)$$

on résout avec une méthode eRK, puis on réécrit le schéma dans la variable y

$$y^{(i)} = y^n + \Delta t \sum_j a_{ij} k_j, \quad i = 1, \ldots, s$$

$$k_i = e^{-c_i \Delta t L} N\left(t^n + c_i \Delta t,\ e^{c_i \Delta t L} y^{(i)}\right)$$

$$y^{n+1} = e^{\Delta t L}\left(y^n + \Delta t \sum_i b_i k_i\right)$$

Résolution par une méthode de Lawson

Besoin de définir une partie linéaire et non-linéaire et d'une structure pour les lier

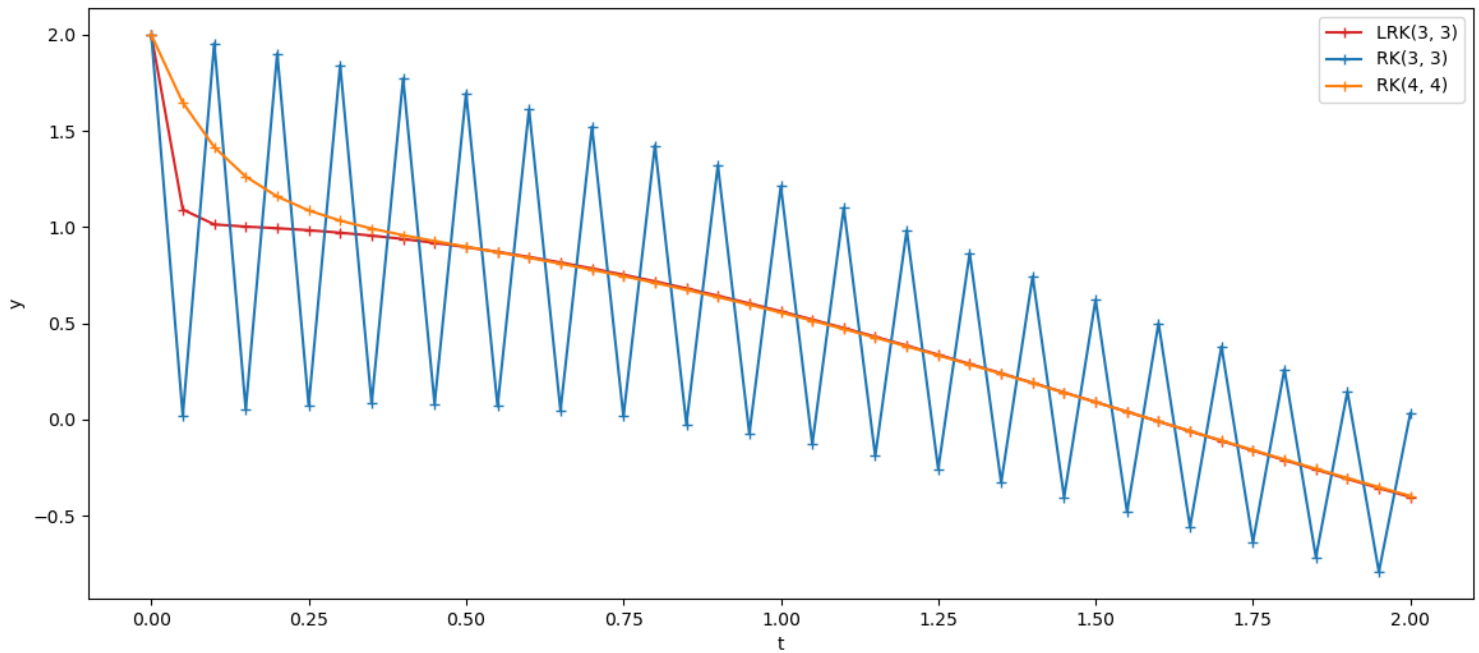
```
double L = k;

double N(double t, double y)
{
  double k = 50;
  return k*std::cos(t);
}

auto pb = ponio::make_lawson_problem(L, N);
```

puis on résout !

```
ponio::solve( pb, ponio::runge_kutta::lrk_44(std::exp), y0,
                  {0., 2.}, dt, "output.txt"_fobs );
```

Résolution par une méthode de Lawson

Besoin de définir une partie linéaire et non-linéaire et d'une structure pour les



pu

```
                {0., 2.}, dt, "output.txt"_fobs );
```

Résolution par une méthode expRK

Résolution de

$$\dot{y} = Ly + N(t, y)$$

On intègre entre t^n et $t^{n+1} = t^n + \Delta t$

$$y(t^n + \Delta t) = e^{\Delta t L} y + \int_0^{\Delta t} e^{(\Delta t - s)L} N(t^n + s, u(t^n + s))\,\mathrm{d}s$$

Construction d'une méthode RK spécialement pour interpoler cette intégrale (les coefficients font intervenir des fonctions φ_ℓ)

$$\begin{aligned} y^{(i)} &= y^n + \Delta t \sum_j a_{ij}(\Delta t L) \cdot (k_j + L y^n), \quad i = 1, \ldots, s \\ k_i &= N(t^n + c_i \Delta t, y^{(i)}) \\ y^{n+1} &= y^n + \Delta t \sum_i b_i(\Delta t L) \cdot (k_i + L y^n) \end{aligned}$$

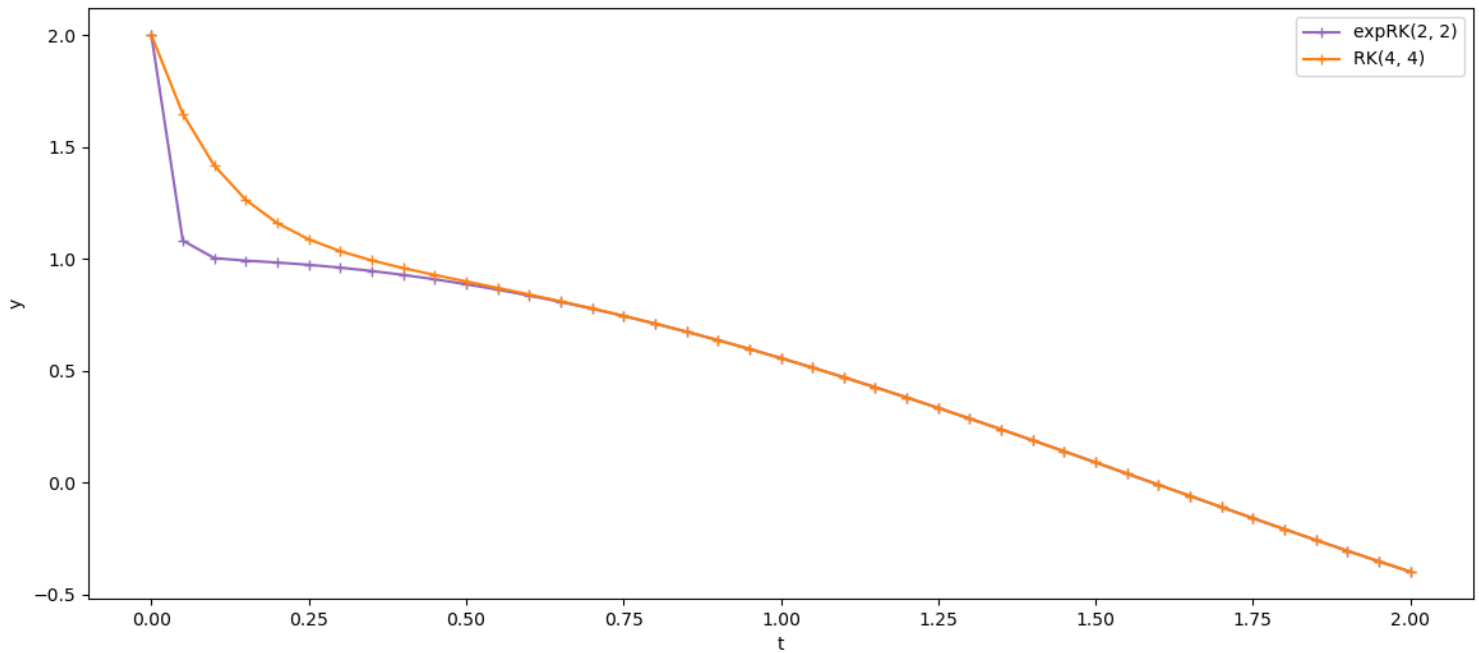
Résolution par une méthode expRK

Même structure que pour les méthodes de Lawson, à la résolution on appelle une méthode expRK

```
ponio::solve( pb, ponio::runge_kutta::exprk22(), y0,
                    {0., 2.}, dt, "output.txt"_fobs );
```

Résolution par une méthode expRK

M
ap



Résolution par une méthode de splitting

Les méthodes de splitting, Lie ou Strang, sont simples à implémenter mais absentes des bibliothèques d'intégration numérique

Attention

On ne fait pas ici du splitting hamiltonien, les sous-étapes sont résolues par une autre méthode d'intégration (RK, expRK, splitting, ...)

Résolution par une méthode de splitting

On split avec le splitting linéaire/non-linéaire mais le splitting à N étapes fonctionne avec ponio

```
double phi_1( double t, double y )
{ return -k*y; }
double phi_2( double t, double y)
{ return k*std::cos(t); }
auto pb = ponio::make_problem(phi_1, phi_2);
```

Il faut définir la méthode de résolution (et son pas de temps) pour chaque sous-problème

```
auto strang = ponio::make_strang_tuple(
  std::make_pair(ponio::runge_kutta::rk_33(), 0.01),
  std::make_pair(ponio::runge_kutta::rk_53(), 0.02)
);
```

puis on résout

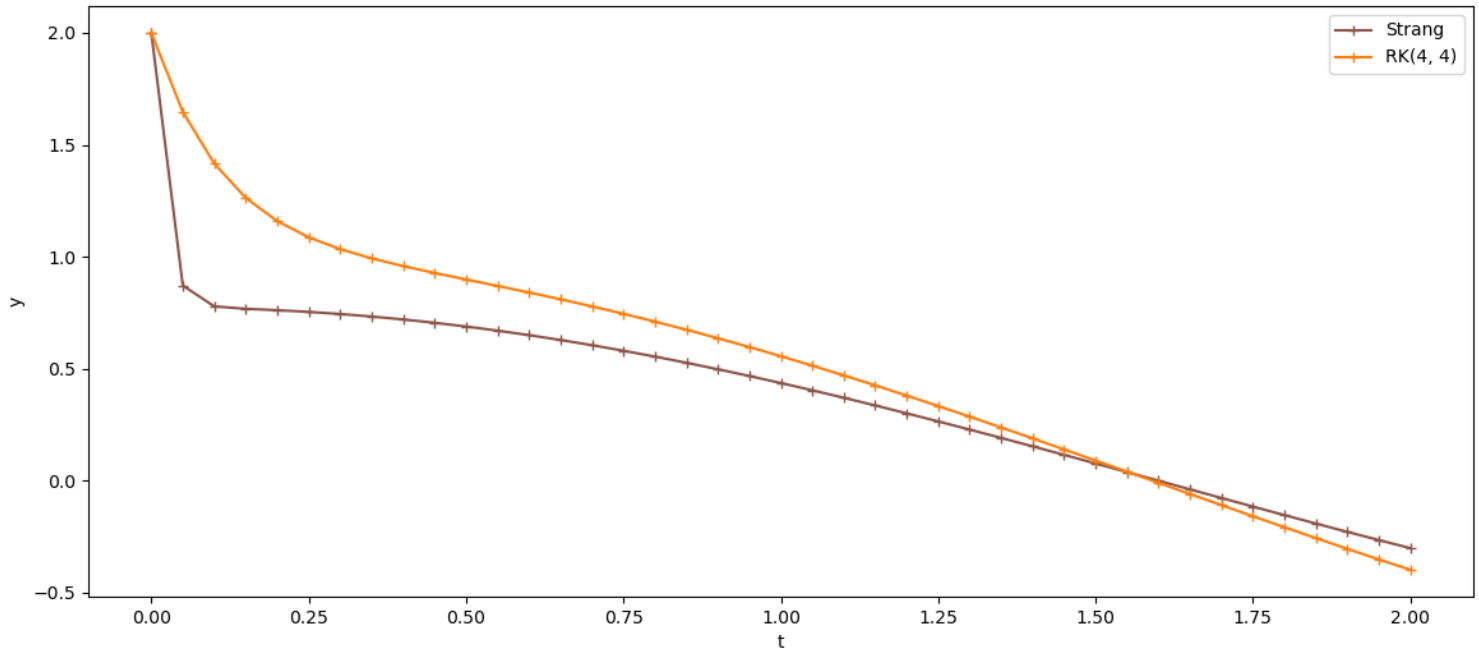
```
ponio::solve( pb, strang,
    y0, {0., 2.}, dt, "output.txt"_fobs );
```

Résolution par une méthode de splitting

On split avec le splitting linéaire/non-linéaire mais le splitting à N étapes fonctionne avec ponio

Il · e

so



puis on résout

```
ponio::solve( pb, strang,
    y0, {0., 2.}, dt, "output.txt"_fobs );
```

Plan

Modèle de Belousov-Zhabotinsky 1d à 3 équations

Réaction chimique périodique avec 3 espèces

$$\begin{cases} \partial_t a = D_a \partial_{xx} a + \frac{1}{\mu}(-qa - ab + fc) \\ \partial_t b = D_b \partial_{xx} b + \frac{1}{\epsilon}(qa - ab + b(1-b)) \\ \partial_t c = D_c \partial_{xx} c + b - c \end{cases}$$

avec $\epsilon = 10^{-2}$, $\mu = 10^{-5}$, $f = 3$, $q = 2 \cdot 10^{-4}$
et $D_a = \frac{1}{400}$, $D_b = \frac{1}{400}$, $D_c = \frac{0.6}{400}$

Opérateur de diffusion ($\partial_{xx}\cdot$) **donc** difficile à stabiliser avec une méthode explicite.

Résolution avec la méthode ROCK2 et ROCK4

Méthode inspirée par les méthodes Runge-Kutta-Chebyshev, avec optimisation des coefficients pour augmenter la stabilité sur l'axe réel négatif

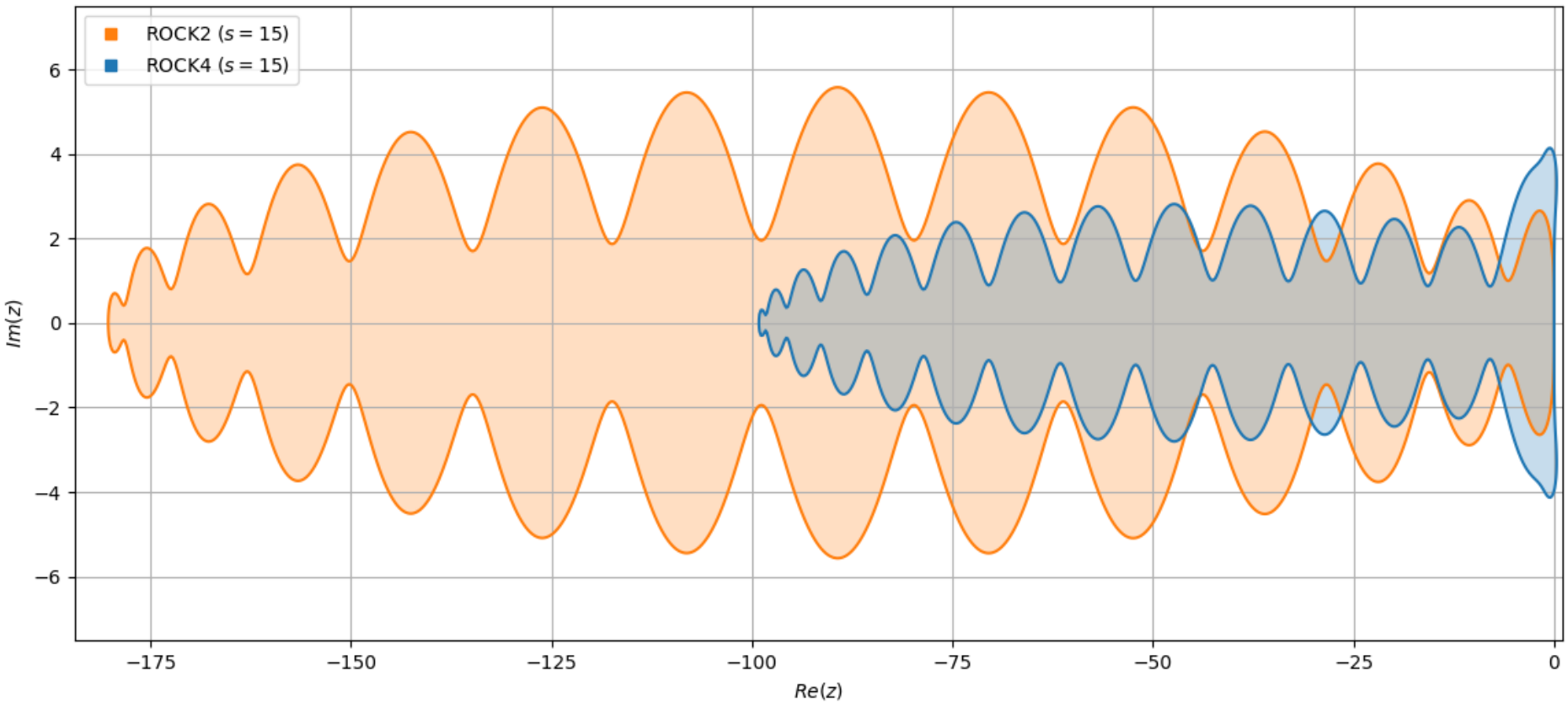


Figure: Domaine de stabilité des méthodes ROCK2 et ROCK4

Résolution avec la méthode ROCK2 et ROCK4

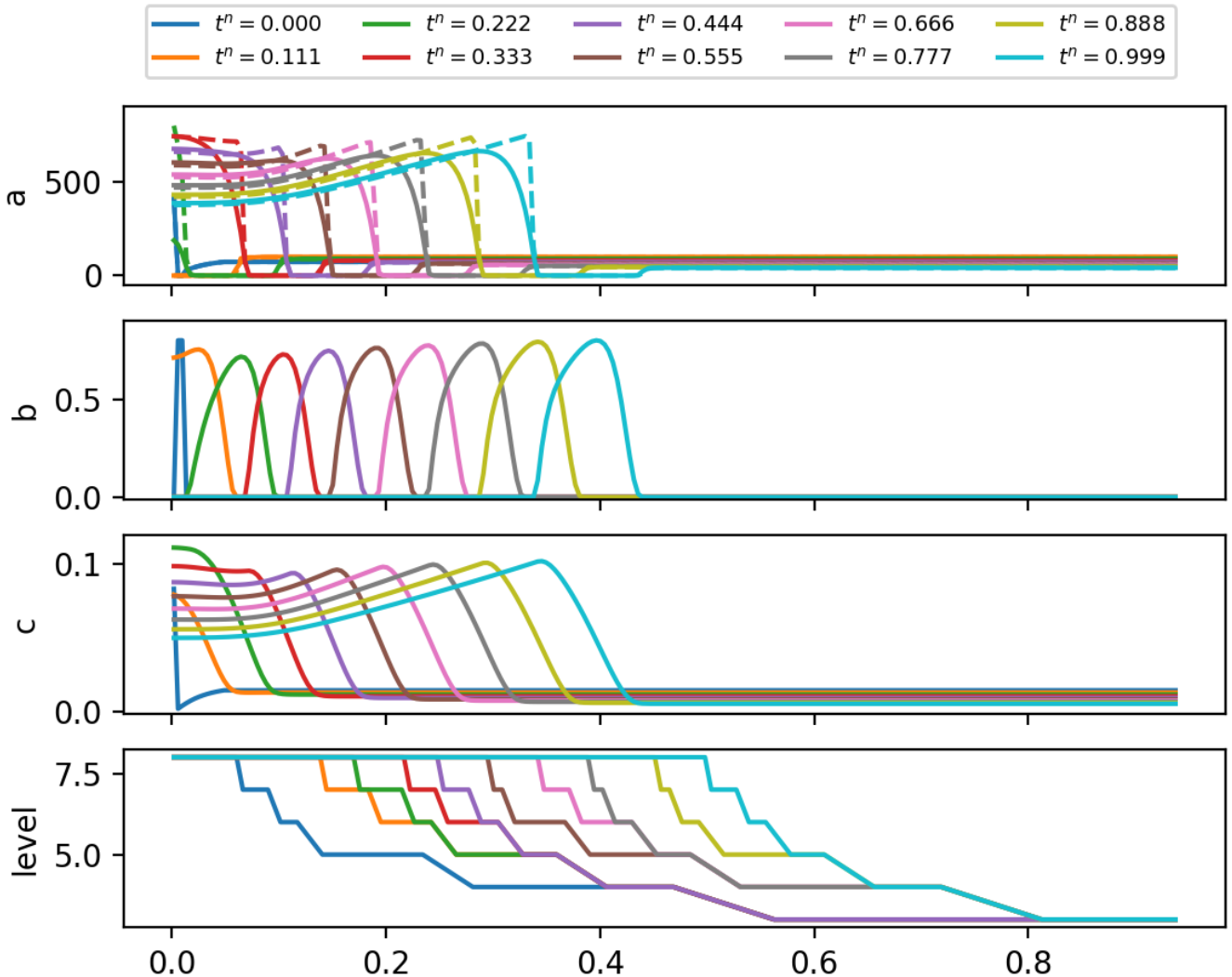


Figure: Résultats de Belousov-Zhabotinsky avec ROCK4

Résolution avec PIROCK

PIROCK : une méthode IMEX pour de la réaction-diffusion résolue par un couplage d'une méthode DIRK et ROCK2

Résolution avec PIROCK

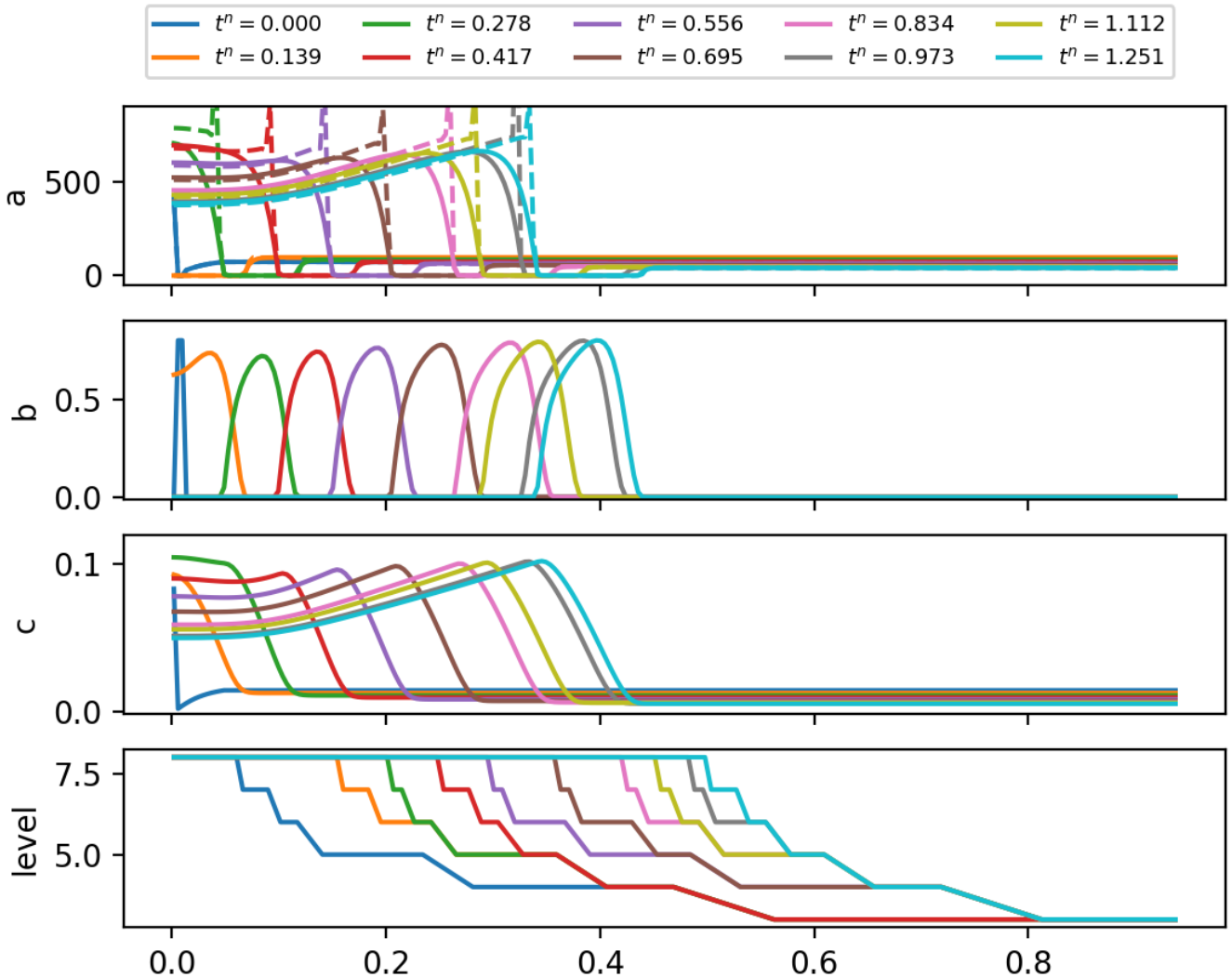


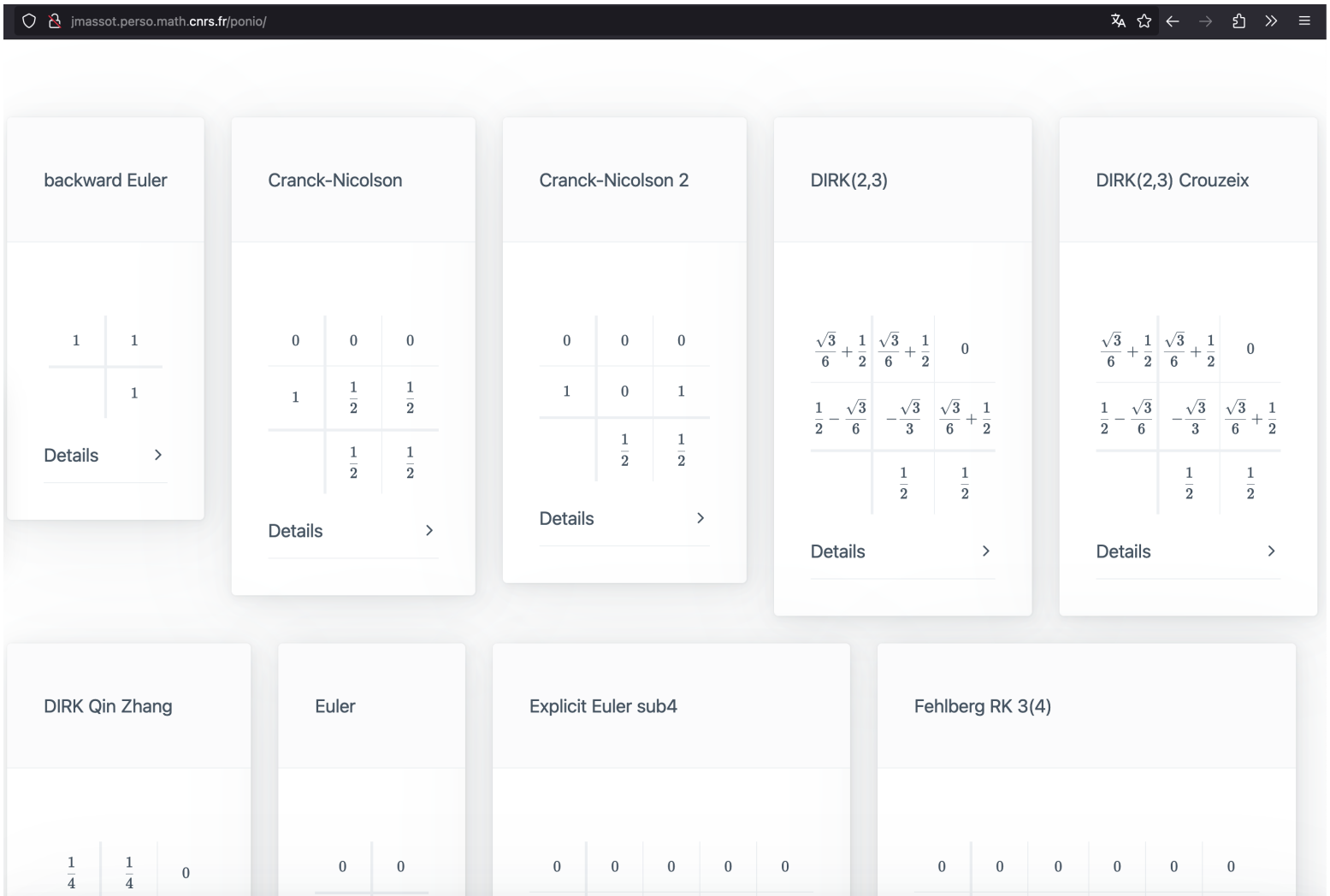
Figure: Résultats de Belousov-Zhabotinsky avec PIROCK

Plan

Que contient ponio ?

Une analyse de méthodes RK :

http://jmassot.perso.math.cnrs.fr/ponio/



Une petite liste de méthodes de type Runge-Kutta

Que contient ponio ?

Un solver C++ : `https://github.com/hpc-maths/ponio`

Conclusion

- ✔ Une analyse automatique des méthodes Runge-Kutta
  - eRK, DIRK, **[WIP]** expRK
- ✔ Une interface simple pour le solveur C++
- ✔ Simplicité d'ajout de méthodes
- ✔ De nombreuses méthodes eRK, DIRK, Lawson et expRK
- ✔ Méthodes de *splitting* (bientôt Suzuki !)
- ✔ Des intégrateurs de nouvelles génération (ROCK, PIROCK)
- ✔ Interface simple avec Eigen et samurai
- ∼ Performances comparables à boost::odeint, quelques idées pour les améliorer
- ✗ Bientôt de l'IMEX **[WIP]**
- ✗ Une interface Python **[WIP]**
- ∼ Collaboration avec le CEA pour code de simulation de risque hydrogène

Vous pouvez contribuer `https://github.com/hpc-maths/ponio` !

```
conda install conda-forge::ponio
```

Merci de votre attention