Politecnico di Milano
School of Systems Engineering
MSc of Mathematical Engineering

Multirate TR-BDF2 method with error control

Ludovica Delpopolo Carciopolo
823678

Academic year 2014 - 2015

Contents

Chapter 1

Introduction

In this report, we present a c++ code to solve large ODEs system or PDEs converted into an ordinary differential equation system. The core of the code is the multirate TRBDF2 method with error control, but to compare the results we have implemented also other methods. A multirate method is one that can take different step size for different components of the solution.
Multirate methods partition the solution into "active" and "latent" components. The active components are integrated using small steps while the latent ones are computed with larger time steps; however they must be coupled together. We introduced for the multirate method a self-adjusting time step strategy for the numerical solution of ODEs. The step size for a system is determined by the local time variation of the solution. For a given global step, a tentative approximation for all the components is done first. The components, for which the error estimator suggest a refinement are needed, are computed again with a smaller time step. The refinement is continued until the solution of all components is good. The size of a time slab, as the size of a "micro" step is determined automatically.

The algorithm is taken from the phd thesis (7). Ranade has written the code in Matlab in a recursive version. We wrote an interative version of the code and modified the time stepping strategy and the interpolator in all the sub-levels major than two.
I have developed the code during an internship at IFPEN. In some cases I had to adapt the code to their code-structures. For example the geochemistry problem has been defined in order to test the multirate algorithm in some applications of interest to them and in order to make it easier to use the code with their data. We will present the algorithms and how the code should work focusing on some aspects of the implementation. At the end some numerical experiments are presented to test the performance of the multirate method.

Chapter 2

Multirate TR-BDF2 algorithm

To calculate the numerical solution of ODEs system there are a lot of methods that use different local time steps that varying in time, but they are constant over the components. With the multirate methods we are able to have a big time step for the latent components while a smaller time step for the faster ones.

Given a global time step $\Delta t_{n+1} = t_{n+1} - t_n$, where the intervals $[t_n, t_{n+1}]$ are called time slabs, we compute the approximation of the solution at this new level for all components and, for those components that the error estimator is bigger than the tolerance, we calculate the solution with a time step smaller than the previous one, where the size is selected with a self-adjusting strategy.

In this smaller time step, for some components from the refinement set, we will need solution values of components where we do not refine, we can use interpolation based on the information available at the time t_n and t_{n+1}. The refinement is continued until the error estimator is below the prescribed tolerance for all components.

2.1 TR-BDF2 method

The TR-BDF2 method is a composite one step, two stage method, based on the trapezoidal rule and the backward differentiation formula of order 2.

The solution is given by:

$$\begin{aligned} \boldsymbol{y}_{n+\gamma} &= u_n + \frac{\gamma}{2} h(\boldsymbol{f}(t, \boldsymbol{y}_n) + \boldsymbol{f}(t, \boldsymbol{y}_{n+\gamma}) \\ \boldsymbol{y}_{n+1} &= \frac{1}{\gamma(2-\gamma)} \boldsymbol{y}_{n+\gamma} - \frac{(1-\gamma)^2}{\gamma(2-\gamma)} \boldsymbol{y}_n + \frac{1-\gamma}{2-\gamma} h \boldsymbol{f}(t, \boldsymbol{y}_{n+\gamma}) \end{aligned} \tag{2.1}$$

where $\gamma = 2 - \sqrt{2}$.

The method has the following Butcher tableau:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \gamma & d & d & 0 \\ 1 & w & w & d \\ \hline & w & w & d \\ \hline & (1-w)/3 & (3w+1)/3 & d/3 \end{array}$$

where $\gamma = 2 - \sqrt{2}, d = \frac{\gamma}{2}$ and $w = \frac{\sqrt{2}}{4}$ the last row corresponds to the companion third order method that can be used to build a convenient error estimator. Is is used the standard terminology for Butcher arrays

$$\begin{array}{c|c}
\boldsymbol{c} & \mathrm{A} \\
\hline
 & \boldsymbol{b}^T \\
\hline
 & \hat{\boldsymbol{b}}^T
\end{array}$$

2.1.1 Evaluation

Let $h = h_n$ denote the current step size. Instead of iterating on the variable $\boldsymbol{y}$, we define another variable $\boldsymbol{z} = h\boldsymbol{f}(t, \boldsymbol{y})$ and solve for this variable by iteration.
For the initial guess of the first implicit stage, we take $\boldsymbol{z}^0_{n+\gamma} = \boldsymbol{z}_n$. This is a crude approximation, but it works good, an alternative is to obtain an initial guess by extrapolating the derivative of the interpolator from the end of the previous step. For any k we take

$$\boldsymbol{y}^k_{n+\gamma} = (\boldsymbol{y}_n + d\boldsymbol{z}_n) + d\boldsymbol{z}^k_{n+\gamma}. \tag{2.2}$$

Then $\boldsymbol{z}_{n+\gamma}$ is computed by simplified Newton iteration

$$\begin{aligned} (\boldsymbol{I} - hd\boldsymbol{J})\boldsymbol{\Delta}^k &= h\boldsymbol{f}(t_{n+\gamma}, \boldsymbol{y}^k_{n+\gamma}) - \boldsymbol{z}^k_{n+\gamma} \\ \boldsymbol{z}^{k+1}_{n+\gamma} &= \boldsymbol{z}^k_{n+\gamma} + \boldsymbol{\Delta}^k. \end{aligned} \tag{2.3}$$

The initial guess for the final stage is obtained by extrapolating the derivative of the cubic Hermite interpolant to values at t_n and $t_{n+\gamma}$. This leads to

$$\boldsymbol{z}^0_{n+1} = \beta_0\boldsymbol{z}_n + \beta_1\boldsymbol{z}_{n+\gamma}\beta_2(\boldsymbol{y}_{n+\gamma} - \boldsymbol{y}_n) \tag{2.4}$$

with $\beta_0 = 1.5 + \sqrt{2}$, $\beta_1 = 2.5 + 2\sqrt{2}$ and $\beta_2 = -(6 + 4.5\sqrt{2})$.
For any k we take

$$\boldsymbol{y}^k_{n+1} = (\boldsymbol{y}_n + w\boldsymbol{z}_n + w\boldsymbol{z}_{n+\gamma}) + d\boldsymbol{z}^k_{n+1} \tag{2.5}$$

and the simplified Newton iteration becomes

$$\begin{aligned} (\boldsymbol{I} - hd\boldsymbol{J})\boldsymbol{\Delta}^k &= h\boldsymbol{f}(t_{n+1}, \boldsymbol{y}^k_{n+1}) - \boldsymbol{z}^k_{n+1} \\ \boldsymbol{z}^{k+1}_{n+1} &= \boldsymbol{z}^k_{n+1} + \boldsymbol{\Delta}^k. \end{aligned} \tag{2.6}$$

In terms of the scaled derivatives, the estimated local error is

$$\boldsymbol{est} = (\hat{b}_1 - b_1)\boldsymbol{z}_n + (\hat{b}_2 - b_2)\boldsymbol{z}_{n+\gamma} + (\hat{b}_3 - b_3)\boldsymbol{z}_{n+1}, \tag{2.7}$$

but we use a modified error estimate $\boldsymbol{Est}$ as the solution of

$$(\boldsymbol{I} - hd\boldsymbol{J})\boldsymbol{Est} = \boldsymbol{est}. \tag{2.8}$$

2.1.2 Interpolation

At each time step TR-BDF2 provides three solution values and three derivative values that are second order accurate. Let $t_{n+\gamma} = t_n + \gamma h$ and $t_{n+1} = t_n + h$, let $\boldsymbol{y}_n, \boldsymbol{y}_{n+\gamma}$, and $\boldsymbol{y}_{n+1}$ denote the approximations to $\boldsymbol{y}(t_n), \boldsymbol{y}(t_{n+\gamma})$ and $\boldsymbol{y}(t_{n+1})$, respectively, and let $\boldsymbol{z}_n, \boldsymbol{z}_{n+\gamma}$, and $\boldsymbol{z}_{n+1}$ denote the approximations to $h\boldsymbol{y}'(t_n), h\boldsymbol{y}'(t_{n+\gamma})$ and $h\boldsymbol{y}'(t_{n+1})$, respectively. We can obtain intermediate values by interpolating these approximations. An option is to use cubic Hermite interpolation:

$$P(t) = (\boldsymbol{v}_3 - 2\boldsymbol{v}_2)r^3(x) + (3\boldsymbol{v}_2 - \boldsymbol{v}_3)r^2(t) + \boldsymbol{v}_1(t) + \boldsymbol{v}_0, \tag{2.9}$$

where, for $t_n \le t \le t_{n+\gamma}$,

$$\boldsymbol{v}_0 = \boldsymbol{y}_n, \qquad \boldsymbol{v}_1 = \gamma \boldsymbol{z}_n, \qquad \boldsymbol{v}_2 = \boldsymbol{y}_{n+\gamma} - \boldsymbol{y}_n - \boldsymbol{v}_1,$$
$$\mathbf{v}_3 = \gamma(\boldsymbol{z}_{n+\gamma} - \boldsymbol{z}_n), \qquad r(t) = \frac{t - t_n}{\gamma h}$$

and, for $t_{n+\gamma} \le t \le t_{n+1}$,

$$\boldsymbol{v}_0 = \boldsymbol{y}_{n+\gamma}, \qquad \boldsymbol{v}_1 = (1-\gamma)(\boldsymbol{z}_{n+1} - \boldsymbol{z}_{n+\gamma}), \qquad \boldsymbol{v}_2 = \boldsymbol{y}_{n+1} - \boldsymbol{y}_{n+\gamma} - \boldsymbol{v}_1,$$
$$\boldsymbol{v}_3 = (1-\gamma)(\boldsymbol{z}_{n+1} - \boldsymbol{z}_{n+\gamma}), \qquad r(t) = \frac{t - t_{n+\gamma}}{(1-\gamma)h}.$$

2.2 Variable step size control

Let us consider an attempted step from time t_n to $t_{n+1} = t_n + \Delta t_{n+1}$ with step size Δt_{n+1}. Having the estimate (2.8) and a tolerance Tol specified by the user. Let τ_r and τ_a be the error tolerances for relative and absolute errors, respectively. For the i-th component, we impose the condition that the i-th error component must satisfy

$$Est_i = \tau_r |y_n|_i + \tau_a \tag{2.10}$$

for the numerical solution of that component to be accepted. We also define the vector of normalized errors as

$$\boldsymbol{\eta} = \left[\frac{Est_1}{\tau_r |y_n|_1 + \tau_a}, \frac{Est_2}{\tau_r |y_n|_2 + \tau_a}, ..., \frac{Est_N}{\tau_r |y_n|_N + \tau_a} \right]^T \tag{2.11}$$

where N is the number of components. The indices set of the "active" components for time step refinement is given by

$$\mathcal{A} = \{ i : \eta_i > \delta \} \tag{2.12}$$

where $\delta \in (0; 1)$ is a coefficient. The global time step is taken as:

$$h_{new} = \nu \min_j \left(\frac{\tau_r |y_n|_j + \tau_a}{Est_j} \right)^{\frac{1}{3}} \tag{2.13}$$

The active components are refined for $t_n \to t_{n+1}$ using the time-step computed from the error estimates as follows:

$$h_{new} = \nu \min_{k \in \mathcal{A}} \left(\frac{\tau_r |y_n|_k + \tau_a}{Est_k} \right)^{\frac{1}{3}} \tag{2.14}$$

ν is a safety parameter used to make the estimate for the new time steps somewhat conservative in order to avoid repeated step failures. The time-step adaptation mechanism is used in the refinement stages also. If the new time step is such that the $t_{end-1} + t_{new} > t_{end}$

where, t_{end-1} is the current simulation time at the end of a step and tend is the final time for the given refinement stage, then the time step is set to $h = t_{end} - t_{end-1}$ so that the refinement stage is synchronized with the global step. The refinement procedure is continued recursively until all components have error estimates under the prescribed tolerance. The next global time step is computed using (2.13) the new time step would be appropriate for the component which had the worst error out. If the parameter δ is chosen to be very small then at each time step a large number of components are marked for refinement but the latent components are integrated with large global steps. Instead, for a relatively large value of delta, very few components would be refined because the global steps would become smaller.

Chapter 3

C++ code structure

The C++ program has been developed using only an external library: Eigen library for linear algebra. The code is structured in class objects. In the following we will show in detail the most important classes. The code has also a reference manual generated using Doxygen.

3.1 StandardTypes.h

In this file we have some utility that we use in the rest of the code.

Listing 3.1: StandardTypes.h

```
namespace Utils {

typedef double Real;
typedef bool Bool;
typedef int Integer;
typedef std::string String;
typedef char Char;

typedef Eigen::Matrix<String, Eigen::Dynamic, 1> StringVector;
typedef Eigen::Matrix<Real, Eigen::Dynamic, 1> RealVector;
typedef Eigen::Matrix<Integer, Eigen::Dynamic, 1> IntegerVector;

typedef Eigen::SparseMatrix<Real>  RealMatrixSparse;
typedef Eigen::SparseMatrix<Integer>  IntegerMatrixSparse;

typedef std::map<String,Integer> MapInteger;
typedef std::map<String,Real> MapReal;

typedef std::pair<String,Integer> KeyInteger;
typedef std::pair<String,Real> KeyReal;
}
```

3.2 IMasterSolver

To solve in time a ODEs system or a PDE equation we want to implement a code that it could be extended at other methods with different choice of the time stepping strategy. For this reason there is an interface IMasterSolver, its derived classes call a single time step solver.

Inside the code we have implemented as derived classes of IMasterSolver:

- MultiRateTRBDF2Solver
- SingleRateTRBDF2Solver
- SingleRateBESolver
- SingleRateFESolver

In the based class, there are, as pure virtual methods, all the methods set and get for the members, the method compute() to solve the time loop and the method getSolution() to get the output in the main. All this methods must to be implemented in each derived class.

Listing 3.2: IMasterSolver.h

```
class IMasterSolver {
public:
   virtual void setNewtSolv(std::unique_ptr<INewtonSolver> newtSolv)=0;
   virtual void setHinitial(Real hinitial)=0;
   virtual void setHz1initial(const RealVector& hz1initial)=0;
   virtual void setTfinal(Real tfinal)=0;
   virtual void setTinitial(Real tinitial)=0;
   virtual void setYinitial(const RealVector& yinitial)=0;
   virtual void setNewtonTolAbs(Real newtonTolAbs)=0;
   virtual void setNewtonTolRel(Real newtonTolRel)=0;
   virtual void setErrorTolAbs(Real errorTolAbs)=0;
   virtual void compute() = 0;
   virtual void setFun(IRHSFunction* fun) = 0;
   virtual  std::vector<std::unique_ptr<ISolverData>>&& getSolution() = 0 ;

};
```

The solution is a derived class vector of ISolverData, where we store all the information that are necessary to do the post processing. For each derived class there is the specific implementation of this interface.

3.2.1 MultiRateTRBDF2Solver

If we analyze the MultiRateTRBDF2Solver we can see some attributes that are useful to compute the solution:

Listing 3.3: MultiRateTRBDF2Solver.h

```
class MultiRateTRBDF2Solver: public IMasterSolver {
public:
   //! Sets the RHS Function
   void setFun(IRHSFunction* fun);
   //! Sets the Newton Solver
   void setNewtSolv(std::unique_ptr<INewtonSolver> newtSolv);
   //! Sets the time step for the first time slab
   void setHinitial(Real hinitial);
   //! Sets the initial guess for the variable z
   void setHz1initial(const RealVector& hz1initial);
   //! Sets the final time
   void setTfinal(Real tfinal);
   //! Sets the initial time
   void setTinitial(Real tinitial);
   //! Sets the initial condition
   void setYinitial(const RealVector& yinitial);
   //! Gets the solution vector
   std::vector<std::unique_ptr<ISolverData>>&& getSolution();
   //! Sets the absolute Newton tolerance
   void setNewtonTolAbs(Real newtonTolAbs);
   //! Sets the relative Newton tolerance
   void setNewtonTolRel(Real newtonTolRel);
   //! Sets the absolute Error tolerance
   void setErrorTolAbs(Real errorTolAbs);
   //! Sets the relative Error tolerance
   void setErrorTolRel(Real errorTolRel);

   void compute();

private:
   Real tinitial;           /*!< Initial time */
   Real hinitial;           /*!< Initial size of the time step */
   Real Newton_tol_rel;     /*!< Relative tolerance for the Newton solver */
   Real Newton_tol_abs;     /*!< Absolute tolerance for the Newton solver */
   Real Error_tol_rel;      /*!< Relative tolerance for the estimate error */
   Real Error_tol_abs;      /*!< Absolute tolerance for the estimate error */
   RealVector hz_1initial; /*!< Initial guess for z variable */
   RealVector yinitial;     /*!< Initial solution */
   Real tfinal;             /*!< Final time */
   IRHSFunction *fun;       /*!< RHS Function */
   std::unique_ptr<INewtonSolver> newt_solv;      /*!< Newton solver */
   CubicHermiteInterpolationSolver interpolation; /*!< Interpolation */
   TRBDF2Solver internal_solver;                  /*!< Internal time step solver */
   std::vector<std::unique_ptr<ISolverData>> outputDataVector;  /*!< Output data */
};
```

we have to set all the classical variables as the Newton tolerance or the initial and the final time, but also the function that we want to solve (IRHSFunction) and which Newton Solver we want to use (INewtonSolver).

We also have to declare the interpolator that we need to interpolate the solution for latent components inside a refining. In this case we implemented the Cubic Hermite Interpolator as explained in section (2.1.2). Every time that we want to use the function interpolatation.compute(), before we have to set the input, it is a class CubicHermiteInterpolationCubicSolverInput as attributes it has:

- ref a vector of 0 or 1, if a component has value equal to 0, it means that it must to be interpolated,
- the variables y and z at the initial, intermediate and final times of the bigger time step,
- the initial, intermediate and final times where we know the solution at the bigger time step.

After the interpolation, the solution is stored inside an object of type CubicHermiteInterpolationCubicSolverOutput, it gets the y vectors at the intermediate and final time (for the components that should not to be interpolated the values are equal to zero).

Another attribute is an object of type TRBDF2Solver, this one is a class with a method that compute the solution for only one time step with the TR-BDF2 method as explained in section (2.1.1). Inside the method compute() of the MultiRateTRBDF2Solver we have implemented the time stepping strategy of the section (2.2), it calls the method internal_solver.compute() whenever we have to calculate the solution for the next step. The implementation of the MultirateTRBDF2Solver could be outline with this algorithm:

- Initialization of the variables that are necessary to compute the solution,
- initialization of the TR-BDF2 parameters,
- initialization of the vectors that are necessary to store all the variables that we need to do the interpolation in a sub-level.
- time loop, while the current time is less than the final time of a sub-level:
  - set the input for the TR-BDF2Solver class,
  - calls the method compute() to calculate the solution for a single time step (in a sub-level it also calls the interpolant for the latent components),
  - store the solution from the TRBDF2SolverOutput (class where we store the output of the TRBDF2Solver)
  - compute the Error Estimator,
  - verify if the time step is accepted or if we have to refine. In the last case we define the sets of the active and latent components,
  - store the solution in the output vector,
  - **if** we need to refine:
    - store values that are necessary for the interpolation
    - calculate the Δt of the rafinement step with the formula (2.14).
  - **if** the time step is not good and it is rejected, this happens because we have to refine the same components of the bigger time step:
    - we don't accept the solution and we calculate a smaller size of the time step for the same local initial time.
  - **if** the time step is good and we don't need to refine:
    - we accept the solution and we compute the new size of the next time step (2.13).

Whenever we call the TR-BDF2 method to integrate in time, if it is a non linear problem, the method calls the Newton solver. Solution might not necessarily converge, in this case we rejected the size of the time step and we reduce it.

At the end of the while loop, when we exit from a sub-level we have to pay attention at a special case: if the value t_{final} of the previous (or more) sub-level is equal to the current one, we have to set: the correct initial guess for the z variables, the new proposal time step and we have to exit from more than one sub-level.

Every time that we compute the new size for the time step, even if we are in a sub-level, we must check that the time is not over the final time of the sub-level. For the zero sub-level the final time corresponds with the last time of the simulation.

TRBDF2Solver

The class TRBDF2Solver has inside the members and the methods useful to advance by one time step.

Listing 3.4: TRBDF2Solver.h

```
class TRBDF2Solver {
public:
   void setInput(const TRBDF2SolverInput& input);

   const TRBDF2SolverOutput& getOutput() const;

   void setOutput(const TRBDF2SolverOutput& output);

   void setRhsFun(IRHSFunction* rhsFun);

   void setNewtSolv(std::unique_ptr<INewtonSolver> newtSolv);

   void compute();

   const TRBDF2SolverParameters& getParam() const;

   void setParam(TRBDF2SolverParameters& param);


private:
   TRBDF2SolverParameters param;              /*!< Method Parameters */
   TRBDF2SolverInput input;                   /*!< Input parameters to compute the solution */
   TRBDF2SolverOutput output;                 /*!< Output values */
   IRHSFunction* rhs_fun;                     /*!< RHS Function */
   std::unique_ptr<INewtonSolver> newt_solv; /*!< Newton Solver */
};
```

We can see that there are as attributes: the input, the output, the parameters but also the RHS function and the Newton solver. To solve the single time step we call the method compute(), here we will compute the solution respect the variables $\boldsymbol{z}$ as explained in the section (2.1.1). For this reason we need to define the Newton function that the Newton solver is going to solve. The method could be sketched as the following:

- initialization of all the variables and the parameters,
- we have to split the problem in the case if we are in a sub-level or not, because if we are refining we have to compute the solution of a reduce problem, we should select only the active components,
- initialization of the Newton function for the Trapezoidal Rule stage,
- calling the Newton solver, we going to compute the solution for the first stage,
- initialize the initial guess for the second stage and the Newton function for the BDF of order 2 method,
- call the Newton solver to calculate the solution of the second stage,
- store the results inside the TRBDF2SolverOutput.

3.2.2 SingleRateTRBDF2Solver

SingleRateTRBDF2Solver class has the same methods and the members of the MultiRateTRBDF2Solver exception for the interpolator, it is not present. In the single rate version, we are not going to refine the latent components. We calculate the error estimator and than we decide if we will accept or refuse the time step. We use an adaptive time step strategy:

- **if** we accept the time step we calculate the next one with (2.13) equation,
- **if** we refuse the time step we must use a smaller one for all the components.

3.2.3 SingleRateBESover

For the implementation of the Backward Euler method we take a fixed size of the time step, exception for the last one. It could be smaller than the others because it has to respect the following condition:

$$t_{end-1} + \Delta t_{end} = T_{final}. \tag{3.1}$$

Inside we initialize the BENewtonFunction,that is involved to solve the Newton system. To advance by one time step, we have implemented the class BESolver.

3.2.4 SingleRateFESover

The code for the Forward Euler method is very easy, as the Backward Euler we use a fixed size of the time step. In this case is not necessary the Newton solver and the Newton function because, as we know, it is an explicit method. To solve a single time step it calls inside the method compute() of the FESolver class.

For each single step solver, we have implemented the classes input and output to set and get all the variables that are necessary to compute the solution.

3.3 ISolverData

ISolverData is an interface, inside there are all the pure virtual methods that are useful to set and get the information.
Let see the attributes of the MultiRateTRBDF2SolverData class, this is a derived class:

Listing 3.5: MultiRateTRBDF2SolverData.h

```
class MultiRateTRBDF2SolverData: public ISolverData {
public:
   //! Gets the error estimation
   const RealVector& getError() const;
   //! Sets the error estimation
   void setError(const RealVector& error);
   //! Gets the size of the time step
   Real getH() const;
```

```
    //! Sets the size of the time step
    void setH(Real h);
    //! Gets the boolean variable that says if the time step is accepted
    Bool isHasSucceded() const;
    //! Sets the boolean variable that says if the time step is accepted
    void setHasSucceded(Bool hasSucceded);
    //! Gets the time where is computed the solution
    Real getTime() const;
    //! Sets the time where is computed the solution
    void setTime(Real time);
    //! Gets the solution
    const RealVector& getY() const;
    //! Sets the solution
    void setY(const RealVector& y);
    //! Gets the mesh vector, useful to see the active and latent components
    const IntegerVector& getMesh() const;
    //! Sets the mesh vector, useful to see the active and latent components
    void setMesh(const IntegerVector& mesh);
    //! Gets the boolean value refining, it is true if we have to refine the time step
    Bool isRefining() const;
    //! Sets the boolean value refining, it is true if we have to refine the time step
    void setRefining(Bool refining);
    /*! Gets a vector of 0 and 1(if for the i-th component it is equal to 1 means that we are
     *    computed the solution with the TRBDF2 method, if it is equal to 0 it means that we
     *     are interpolating that component's solution)
     */
    const IntegerVector& getNumbComponentsSolve() const;
    //! Sets the vector NumbComponentsSolve
    void setNumbComponentsSolve(const IntegerVector& numbComponentsSolve);

private:
    Real time;  //time where is computed the solution
    Real h;     //Size of the time step
    RealVector y;  //Solution
    RealVector error; //Error estimation
    IntegerVector numb_components_solve;   //Components computed with the method
    IntegerVector mesh;  // Active components in the current iteration
    Bool has_succeded;   // true if we accept the time step
    Bool refining;    // true if we have to refine the time step
};
```

Every time that we compute a time step we store all the information that are necessary to do the post processing, even if it is not accepted; for this reason there are two boolean variables that show us if the time step is accepted and if it is refined.

We want to emphasize the difference between the mesh vector and the numb_components_solve vector; the first one is used to know, when the time step is accepted, which components are computed and which are interpolated. The second one is used to know the number of components that has been solved with the Newton solver, even if we are not accepting the time step.
The others derived classes are very similar to this one, so we don't report the codes.

3.4 INewtonSolver

INewtonSolver class is a base class, its pure virtual method is:

Listing 3.6: INewtonSolver.h

```
class INewtonSolver {
public:
      //! Function to compute the solution with the Newton method for non linear system.
   /*!
    * Returns the solution by reference
    * @param   INewtonfunction      Newton function
    * @param   RealMatrixSparse     Jacobian
    * @param   RealVector           solution
    * @param   Real                 relative tolerance
    * @param   Real                 absolute tolerance
    * @param   Bool                 Newton diverge
    * @param   Integer              maximum number of Newton iteration
    */
   virtual void compute(INewtonFunction& fun, RealMatrixSparse& J,RealVector& Y, const Real
       tol_rel,const Real tol_abs,Bool& diverge_Newton,const Integer MaxNewtonStep )=0;
};
```

We pass as reference the Jacobian that is a sparse matrix and the vector y that contains the initial guess for the Newton Solver. Inside the method, if it converge, we'll update the vector with the non linear system solution.

The two derived classes are:

- BasicNewtonSolver class has implemented the method compute() that for each iteration, until the error is less than the tolerance or the number of iteration is under the value MaxNewtonStep:
  - evaluates the Jacobian and the RHSVector,
  - calculates the error with the L^∞ norm,
  - solves the system for the increment with the SparseLU solver, method of the Eigen library.
  - updates the solution by adding the increase.
- FixedJacobianSolver class uses the same Jacobian evaluation for all the iterations that are necessary to solve the system. To check if the method has converged we calculate the L^∞ and the L^2 norms, both of them should be less than the mixed tolerance to converge.

3.5 INewtonFunction

The pure virtual methods of INewtonFunction are:

Listing 3.7: INewtonFunction.h

```
class INewtonFunction {
public:
   virtual void evalFJ(RealVector& Y, RealVector& F, RealMatrixSparse& J) = 0;
   virtual void evalF(RealVector& Y, RealVector& F) = 0;
};
```

Let see the derived classes that are present in the code:

3.5.1 BESolverNewtonFunction

Listing 3.8: BESolverNewtonFunction.h

```
class BESolverNewtonFunction : public INewtonFunction{
public:
   void setDt(Real dt);
   void setFun(IRHSFunction* fun);
   void setYn(const RealVector& yn);
   void setTnp(Real tnp);

   //! Returns as reference the evaluation of Newton function (F) and Newton Jacobian (J) in Y
       .
   void evalFJ(RealVector& Y, RealVector& F, RealMatrixSparse& J);
   //! Returns as reference the evaluation of Newton function (F) in Y.
   void evalF(RealVector& Y, RealVector& F);

private:
   IRHSFunction* fun;       // RHS Function
   RealVector yn;           // Vector of the solution at the previous time
   Real dt;                 // Size time step
   Real tnp;                // Time where we compute solution
};
```

there are two methods:

- evalFJ is called from the BasicNewtonSolver, it returns as reference the evaluation of the Jacobian matrix and the evaluation of the Newton function in y at time "tnp" .

  In the case of the BESolver the Newton function is:

$$F(y) = y - y_n - \Delta t(f_{RHS}(t_{np}, y)) \tag{3.2}$$

  while the Jacobian matrix is:

$$J = I - \Delta t\frac{\partial f_{RHS}(t_{np}, y)}{\partial y}; \tag{3.3}$$

- evalF returns as reference only the evaluation of the Newton function.

3.5.2 TRBDF2SolverNewtonFunction

Listing 3.9: TRBDF2SolverNewtonFunction.h

```
class TRBDF2SolverNewtonFunction: public INewtonFunction {
public:
   void setDt(Real dt);
   void setFun(IRHSFunction* fun);
   void setYn(RealVector& ya);
   void setTnp(Real tnp);
   void setRef(IntegerVector& ref);
   void setD(Real d);

   //! Returns as references the evaluate of Newton function (F) and Newton Jacobian (J).
   void evalFJ(RealVector& Z, RealVector& F, RealMatrixSparse& J);
   //! Returns as reference the evaluate of Newton function (F).
   void evalF(RealVector& Z, RealVector& F);

private:
   IRHSFunction* fun;    // RHS Function
   RealVector ya;        // vector that corresponds to y at the current time
   Real dt;              // Size time step
   Real tnp;             // Time that we compute solution
   IntegerVector ref;    // Vector to know for which components we have to compute the solution
   Real d;               // Parameters of the TR-BDF2 method
};
```

For the TRBDF2Solver we have to compute two stages, and we have to store the y_a vector that is different if we are computed the TR stage or the BDF stage.

The Newton solver calls the method compute() with the z variable as input, so inside we have to tranform it in a y vector to call the IRHSFunction->evalF() and the IRHSFunction->evaldFdY().
evalFJ() method splits the evaluation into two possible different cases: if we are refining, we want to evaluate only some components; if we are in a global time step and we have to compute all the components.

First of all we have to transform the z variable into y variable,

$$Y = y_a + dZ; \tag{3.4}$$

than we will evaluate the Newton Function and the Newton Jacobian:

$$\begin{gathered} F = Z - \Delta t f_{RHS}(t_{np}, Y) \\ J = I - d\Delta t \frac{\partial f_{RHS}(t_{np}, Y)}{\partial y}. \end{gathered} \tag{3.5}$$

The method evalF() evaluates only the Newton function.

3.6 IRHSFunction

Listing 3.10: IRHSFunction.h

```
class IRHSFunction {
public:
   virtual RealVector evalF(Real time, RealVector& y)=0;
   virtual RealVector evalF(Real time, RealVector& y, IntegerVector& ref)=0;
   virtual RealMatrixSparse evaldFdY(Real time, RealVector& y)=0;
   virtual RealMatrixSparse evaldFdY(Real time, RealVector& y,IntegerVector& ref)=0;
};
```

Inside the derived classes we have to implement the evaluation of the problem that we want to solve, and also the Jacobian, it could be analytic or calculates with finite differences. In this class is present the overloading of the two methods because if we call the methods inside a sub-level, it should returns a smaller vector with the dimension of the active components number.

3.6.1 GeochemistryRHSFunction

The GeochemistryRHSFunction is a zero dimensional ODEs system. To implement its evaluation we need all the parameters that are necessary to compute the right-hand side of the system. It could be written in the following way:

$$\frac{d\boldsymbol{n}}{dt} = \boldsymbol{R}(\boldsymbol{n}) + \boldsymbol{S}_R \tag{3.6}$$

where S_{R_j} is the source rate for the j-th species and

$$Rj = \sum_{l=1}^{N} S_{jl}\pi_l \qquad i = 1...n_{species} \tag{3.7}$$

N is the total number of reactions that are present in the system,
S_{jl} is the stoichiometric matrix and π_l is:

$$\pi_l = \mathcal{K}_l(T)\left[\prod_{S_{jl}>0}\left(\frac{a_j}{kj}\right)^{|S_{jl}|} - \prod_{S_{jl}<0}\left(\frac{a_j}{kj}\right)^{|S_{jl}|}\right] \tag{3.8}$$

$\mathcal{K}_l$ is the kinetic value, it could be a function of the temperature or a constant value, it is depending on the model we want to use,
a_j is the activity of a chemical species, to model them we use the ideal activities for every species,
k_j is the equilibrium constant for the j-th species, we store $log_{10}(k_j)$ inside the vector. We have to define inside the parameters class the following members:

Listing 3.11: GeochemistryParameters.h

```
private:
   StringVector Phase;         /*!< Phases present in the system  */
   StringVector PhaseModel;    /*!< Name of the phases present in the system */
   StringVector PhaseLaw;      /*!< Laws of the phases present in the system */
   StringVector Species;       /*!< Species present in the system */
   StringVector SpeciesPhase;  /*!< In which phase the species are in the system */
   StringVector SpeciesModel;  /*!< Name of the species present in the system */
   StringVector SModel;        /*!< Model of the species */
   RealVector LogK;            /*!< Logarithmic values of the k parameters */
   StringVector Reaction;      /*!< Reactions present in the system  */
   RealVector KReaction;       /*!< kinetic values of the reactions  */
   RealMatrixSparse S_matrix;  /*!< Stoichiometric matrix  */
   RealVector Scostant;        /*!< Source Rate vector */
```

In the method evaldFdY(), we have implemented the Jacobian that is computed with finite differences, it is difficult to know the analytic one because the activity of a chemical species depends on the components.

To evaluate the right hand side of the problem, the method calls some useful functions:

Listing 3.12: GeochemistryRHSUtils.h

```
namespace GeochemistryRHSUtils{

RealVector PhaseActivityModel(RealVector& n,const GeochemistryParameters& parameters);

void ReactionThermodynamics(RealVector& Omega_reac, RealVector& Omega_prod, RealVector&
    activity, const GeochemistryParameters& parameters);

void ReactiveChemistryPb(RealVector& R, RealVector& reac,const GeochemistryParameters&
    parameters);
}
```

PhaseActivityModel takes the concentration of the species and the parameters, it returns the activity vector that is computed with the appropriate law.
ReactionThermodynamics takes the activity vector and the parameters, it returns as reference the vectors $\Omega_{reactants}$ and $\Omega_{products}$, they represent:

$$\Omega_{reac_j} = \prod_{S_{jl}>0} \left(\frac{a_j}{k_j}\right)^{|S_{jl}|} \qquad \Omega_{prod_j} = \prod_{S_{jl}<0} \left(\frac{a_j}{k_j}\right)^{|S_{jl}|} \tag{3.9}$$

ReactiveChemistryPb returns as reference the vector $\boldsymbol{R}$ (3.7).

Set the Parameters

To store all the values we need some classes and some tools that are able to convert the information in a GeochemistryParameters object.

Let see how we can store a simple system with two reactions:

$$\begin{aligned} &H_2O \rightleftharpoons H^+ + OH^- \\ &CaCO_3 + H^+ \rightleftharpoons Ca^{2+} + HCO_3^- \end{aligned} \tag{3.10}$$

Listing 3.13: SetProblemSHPCO2SourceRate.h

```
namespace SetProblemSHPCO2SourceRate{
GeochemistryParameters getParameters(RealVector& y0){
       std::cout << "Get␣Parameters␣for␣TestSHPCO2␣" << "\n" ;

       ///============ System

       ChemicalSystem system;

       ///============ Phase

       ChemicalPhase phaseWater("w","Water","Ideal");
       system.addPhase(phaseWater);

       ChemicalPhase phaseCalcite("m1","Calcite","Ideal");
       system.addPhase(phaseCalcite);

       ///============ Species

       ChemicalSpecies speciesH2O("H2O","w","Solvent","Ideal",0.0);
       system.addSpecies(speciesH2O);

       ChemicalSpecies speciesH("H+","w","Solute","Ideal",0.0);
       system.addSpecies(speciesH);

       ChemicalSpecies speciesCa("Ca++","w","Solute","Ideal",0.0);
       system.addSpecies(speciesCa);

       ChemicalSpecies        speciesHCO3("HCO3-","w","Solute","Ideal",-6.2206340);
       system.addSpecies(speciesHCO3);

       ChemicalSpecies speciesOH("OH-","w","Solute","Ideal",-13.235362);
       system.addSpecies(speciesOH);

       ChemicalSpecies speciesCaCO3("(CaCO3)s","m1","Pole","Ideal",-7.7454139);
```

```
 system.addSpecies(speciesCaCO3);

 ///============ Reaction

 ChemicalReaction reactionH2O("reactionH2O",1.e3);
 KeyReal sp1("H2O",1);
 reactionH2O.addSpecies(sp1);
 KeyReal sp2("OH-",-1);
 reactionH2O.addSpecies(sp2);
 KeyReal sp3("H+",-1);
 reactionH2O.addSpecies(sp3);
 system.addReaction(reactionH2O);

 ChemicalReaction reactionCaCO3("reactionCaCO3",1.e-1);
 KeyReal sp4("Ca++",1);
 reactionCaCO3.addSpecies(sp4);
 KeyReal sp5("HCO3-",1);
 reactionCaCO3.addSpecies(sp5);
 KeyReal sp6("(CaCO3)s",-1);
 reactionCaCO3.addSpecies(sp6);
 KeyReal sp7("H+",-1);
 reactionCaCO3.addSpecies(sp7);
 system.addReaction(reactionCaCO3);

///============ System State

ChemicalSystemState systemInit;

KeyReal sp8("H2O",8072.37);
systemInit.setAmountSpecies(sp8);

KeyReal sp9("H+",0.000347237);
systemInit.setAmountSpecies(sp9);

KeyReal sp10("Ca++",5.84686);
systemInit.setAmountSpecies(sp10);

KeyReal sp11("HCO3-",0.288906);
systemInit.setAmountSpecies(sp11);

KeyReal sp12("OH-",3.40577e-06);
systemInit.setAmountSpecies(sp12);

KeyReal sp13("(CaCO3)s",4331.16);
systemInit.setAmountSpecies(sp13);

///============ Source Rate

ChemicalSourceRate sourceRate;

KeyReal sp14("H2O",50.0);
sourceRate.setSourceRate(sp14);

KeyReal sp15("H+",15.0);
sourceRate.setSourceRate(sp15);

KeyReal sp16("Ca++",13.0);
sourceRate.setSourceRate(sp16);

KeyReal sp17("HCO3-",20.0);
sourceRate.setSourceRate(sp17);
```

```
    KeyReal sp18("OH-",32.0);
    sourceRate.setSourceRate(sp18);

    KeyReal sp19("(CaCO3)s",23.0);
    sourceRate.setSourceRate(sp19);

    GeochemistryParameters param= ChemicalProblemsUtils::buildParameters(system,sourceRate);

    StringVector species = param.getSpecies();
    Integer size = species.size();

    y0.resize(size);
    y0 = ChemicalProblemsUtils::buildInitialCondition(systemInit,species);

    return param;
}
```

We start to declare the ChemicalSystem. We have to add at this object the phases that are present in the system, in this case we have two phases the "Water" and the "Calcite", so we create two objects ChemicalPhase. In each one we store, in this order: a identifier for the phase, the phase's name and the name of the model law that we use.
Then we add the species to the system, to initialize them we have to specify: the name of the species, in which phase it is present, its type , the law we use to create the activity vector, the equilibrium constant $log_{10}(k)$.
To add the reactions, we have to initialize the ChemicalReaction object with the name of the reaction and the kinetic value, than we have to insert the species involved in the specific reaction: we initialize a KeyReal with the stoichiometric value. When we finish to add all the species to the reaction, we will add the reaction to the system object.
To insert the initial condition, we have to declare an object of type ChemicalSystemState, we set the amount of the species initializing a KeyReal with the name of the species and the value.
At the end we set the source rate term only if it is present, otherwise we can omit it. To build the GeochemistryParameters and the RealVector of the initial condition we have created two functions ChemicalProblemsUtils::buildParameters and ChemicalProblemsUtils::buildInitialCondition. The first one takes two objects of type ChemicalSystem and ChemicalSourceRate. The second one takes only the species vector from the parameters, to know in which order the species are insert in the vector and the object of type ChemicalSystemState where there are the initial values.

Let see the private attributes of this new classes:

Listing 3.14: ChemicalSystem.h

```
private:
   std::vector<ChemicalPhase> Phase;          /*!< Phases that are present in the system */
   std::vector<ChemicalSpecies> Species;      /*!< Species that are present in the system */
   std::vector<ChemicalReaction> Reaction;    /*!< Reactions that are present in the system */
```

Listing 3.15: ChemicalPhase.h

```
private:
   std::vector<ChemicalPhase> Phase;          /*!< Phases that are present in the system */
   std::vector<ChemicalSpecies> Species;      /*!< Species that are present in the system */
   std::vector<ChemicalReaction> Reaction;    /*!< Reactions that are present in the system */
```

Listing 3.16: ChemicalSpecies.h

```
private:
   String Species;       /*!< name of the species */
   String SpeciesPhase; /*!< In which phase it is present */
   String SpeciesModel; /*!< Type species */
   String SModel;        /*!< Name of the model adopted */
   Real Logk;            /*!< Logarithmic value of the k coefficient  */
```

Listing 3.17: ChemicalReaction.h

```
private:
   String NameReaction;           /*!< Name of the reaction */
   std::vector<KeyReal> Species; /*!< Species that are involved with the stochiometric value*/
   Real KinValue;                 /*!< Reaction speed */
```

Listing 3.18: ChemicalSystemState.h

```
private:
   std::vector<KeyReal> StateSpecies;/*!< Vector of pair: name of species and initial value */
```

Listing 3.19: ChemicalSourceRate.h

```
private:
   std::vector<KeyReal> SourceRate;/*!< Vector of the source rate's values for each species */
```

3.6.2 HyperbolicEquationRHSFunction

We begin our study of Hyperbolic PDE considering the scalar case.

$$\frac{\partial u(x(t),t)}{\partial t} + \frac{\partial f(u(x(t),t))}{\partial x} = 0 \tag{3.11}$$

To discretize in space we use finite volume with uniform grid, but we have to decide which kind of numerical flux we want to use.

In this RHS function we have implemented a first order upwind method, to apply to one-dimensional linear advection equation.

Let see how we have implemented the method: first of all we have created a interface for the flux $f(u)$:

Listing 3.20: IFluxFunction.h

```
class IFluxFunction {
public:
   virtual Real evalF(Real time, Real y_sol) = 0;
   virtual Real evalFderivative(Real time, Real y_sol) = 0;

};
```

we can see that, as pure virtual methods, there are the evaluation of the flux function and the evaluation of its derivative; both of them are component wise evaluations; the last one is not necessary for the implementation of the upwind method but it could be needful for others discretization's types.

We have to define also the boundary condition, I have implemented only Dirichlet condition but the code could be generalized very easily.

Listing 3.21: IBCFunction.h

```
class IBCFunction {
public:
   virtual Real getValue(Real time) = 0;
};
```

Again, it is a interface, the derived classes will have to be implemented with the method getvalue() that takes the time and returns a real number that corresponds to $u(x = x_0, t)$.

3.6.3 AllenCahnRHSFunction

The third derived class consists of a finite differences uniform space discretization of the Allen-Cahn equation with Neumann boundary conditions

$$\frac{\partial u}{\partial t} = \sigma \frac{\partial^2 u}{\partial x^2} + u(1 - u^2), \tag{3.12}$$

in this case we have not implemented a generic code for the boundary conditions because we used it to compare the algorithm with the results obtained in [8].

3.7 Restab

Restab library consists in a single header file, we want to use this library to generate the data files with the extension .restab. They can be viewed through the java program: DataExplorer.jar.

- GenerateGenericRestabFile::GenericRestab

  ```
  int GenericRestab(std::vector<MasterSolution>& vect_solution);
  ```

  It takes a vector of MasterSolution, this one is a class that contains two attributes, the time and the solution, only for the time steps that are accepted. Thanks to the function MasterSolverOutputManipulator::getOutputVectorRestab we can convert a vector of ISolverData, where there are all the solutions, in a MasterSolution vector.
  If we call this function the code generates two files, one is the file with the data and the another one is a file where we can found a brief description of the variables and the index. With this function we have as index the time step while as variables the time of the time steps and the solution $y_i(t) \quad i = 1 \cdots N$ where N in the number of components. So we can see the solution of each component that depends on the time.

- GenerateGeochemistryRestabFile::GeochemistryRestab

  ```
  int GeochemistryRestab(GeochemistryParameters& param, std::vector<MasterSolution>&
      vect_solution);
  ```

  This function generates a restab file that is very similar to the previous one, the only difference is that this one is specific for the Geochemistry problem, in fact it takes a vector of MasterSolution and GeochemistryParameters object. In the java program we can select which species would like to see the solution in time.

- GenerateNumbComponentsRestabFile::ComponentsRestab

```
int ComponentsRestab( std::vector<std::unique_ptr<ISolverData>>&& vect_solution);
```

The restab file generated by this function is useful if we are solving the problem with the multirate solver because, we can see, the number of components involved in each time step of the simulation.

- GenerateMeshRestabFile::MeshRestab

```
int MeshRestab(Real x0, Integer Nx, Real xL, std::vector<std::unique_ptr<ISolverData>>&&
    vect_solution);
int MeshRestab(std::vector<std::unique_ptr<ISolverData>>&& vect_solution);
```

  Also this functions is serviceable if, the method used to integrate in time, is the multirate because we are plotting for each time step which components are computed and witch are interpolated (if the i-th component has value equal to 1 it means that it is computed 0 otherwise). If all the components have value equal to zero it means that the time step is rejected. Function overloading is being done because we can compute the solution of an ODEs or a PDE and in the second case the components coincide with the x values.

- GeneratePDERestabFile::PDERestab

```
int PDERestab(Real x0, Real Nx, Real xL,  std::vector<MasterSolution>& vect_solution);
```

  To plot the solution at the initial and final time we use this function only in the case that we are solving a PDE problem. It would be inconsistent if we are solving a ODE system.

- GenerateDebugRestabFile::DebugRestab

```
int DebugRestab(Real x0, Real Nx, Real xL,  std::vector<std::unique_ptr<ISolverData>>&&
    vect_solution);
```

  This function it useful when we want to analyze the solution of a PDE problem at each time step, it plots the same result of the previous function but for all time steps. If the time step is rejected from our time stepping strategy we will see all the component values equal to zero.

3.8 Utility

We are interested to know if the multirate is a mass conservative method, for this reason we have implemented a function that check the conservation of the mass for the hyperbolic problem.

```
namespace IsConservativeMethod{
//! Checks if the method is mass conservative.
Bool check(std::vector<std::unique_ptr<ISolverData>>&& vect_solution, Real dx, IFluxFunction&
    f_flux, Real tol);
}
```

The function takes as input the solution vector, the size of the space interval, the flux function and the tolerance, this one depends by the Newton solver tolerance. At each time step, if the step is accepted and if it has not to be refined, we check:

$$\left| \sum_{i=0}^{N-1} u_i^{n+1} - \sum_{i=0}^{N-1} u_i^{n} - (f(t,u_0) - f(t,u_{N-1}))\frac{\Delta t}{\Delta x} \right| < tol \tag{3.13}$$

where N is the nodes number, $f(t,u)$ is the flux function.

Chapter 4

Tests

The following tests are conducted to see the difference between the multirate TR-BDF2 method and the single rate version. For all the non-linear tests we use the Fixed Jacobian Newton solver.

4.1 Allen-Cahn problem

The first test case is implemented to compare the results with the paper (8), the Allen-Chan equation is:

$$\frac{\partial u}{\partial t} = \sigma \frac{\partial^2 u}{\partial x^2} + u(1-u^2), \tag{4.1}$$

for $t > 0, -1 < x < 2$, with initial and boundary conditions

$$\frac{\partial u(-1,t)}{\partial x} = 0, \qquad \frac{\partial u(2,t)}{\partial x} = 0, \qquad u(x,0) = u_0(x). \tag{4.2}$$

We take $\sigma = 9 \cdot 10^{-9}$ and initial profile

$$u_0(x) = \begin{cases} \tanh\left(\dfrac{x+0.9}{2\sqrt{\sigma}}\right) & \text{for } -1 < x < -0.7 \\ \tanh\left(\dfrac{0.2-x}{2\sqrt{\sigma}}\right) & \text{for } -0.7 < x < 0.28 \\ \tanh\left(\dfrac{x+0.36}{2\sqrt{\sigma}}\right) & \text{for } 0.28 < x < 0.4865 \\ \tanh\left(\dfrac{0.613-x}{2\sqrt{\sigma}}\right) & \text{for } 0.4865 < x < 0.7065 \\ \tanh\left(\dfrac{x-0.8}{2\sqrt{\sigma}}\right) & \text{for } 0.7065 < x < 2 \end{cases} \tag{4.3}$$

For this problem we used a uniform grid of 400 points.
We take as Newton tolerance $[1 \cdot 10^{-11}, 1 \cdot 10^{-11}]$ where the first value is the absolute tolerance while the second is the relative tolerance, as tolerance for the error estimator we take $[1 \cdot 10^{-4}, 1 \cdot 10^{-6}]$, as initial time step size we take 0.1.

The solution at time $t_0 = 0$ starts with three "wells", the first well, on the left, persisting during the integration interval. The second well is somewhat thinner then the others and it collapses at time $t \approx 41s$, whereas the third well collapses at $t \approx 141s$ as we can see in Figure (4.1). The output is considered for $T = 142s$.

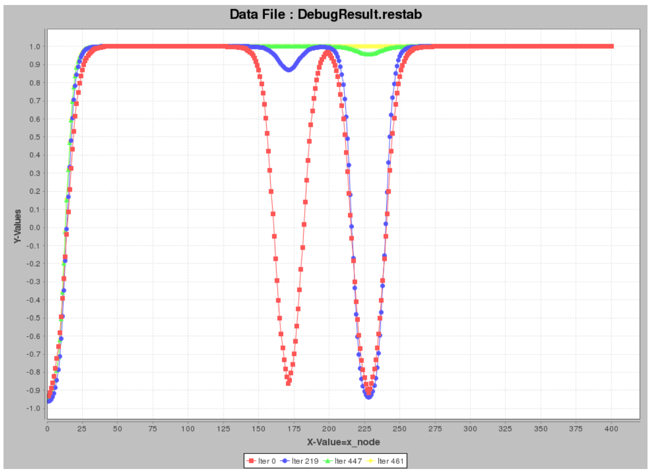


Figure 4.1: Solution at the time step 0 that represents the initial condition, time step 219 is the time $41s$, time step 447 is the time $141s$ and finally the output solution at time $142s$.

If we look the number of components involved in each time step we can see that, for most of the simulation, we compute the solution with the TR-BDF2 method only for a sub-set as showed in Figure (4.2), it implies a computational cost lower than the single rate method (Table (4.1)) even if the number of time steps is almost the same.

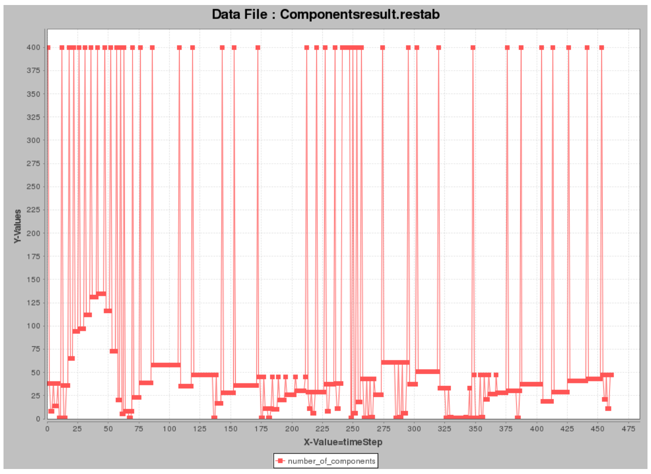


Figure 4.2: Number of components where the TR-BDF2 method is applied for each time step.

Method	Computation time	Numb. time steps
Singlerate TR-BDF2	$903.44s$	468
Multirate TR-BDF2	$124.90s$	461

Table 4.1: Computation time and numbers of time steps for the Allen-Chan problem.

4.2 Geochemistry problem

Let see a test for reactive transport in the context of CO_2 geological storage. In the first test case we consider a compositional system formed of 12 species divided into 4 phases:

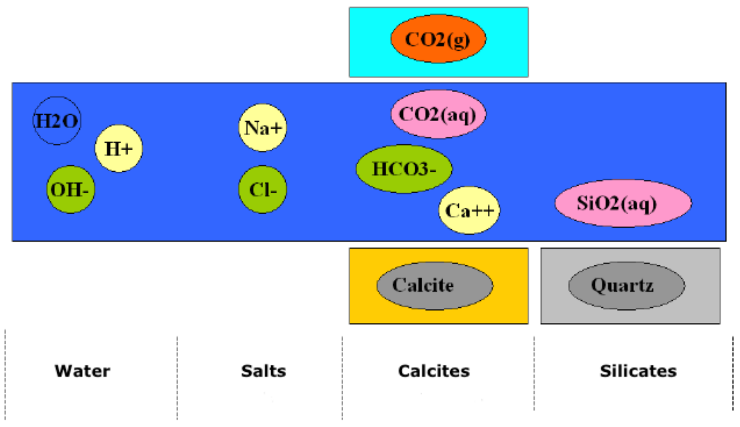


Figure 4.3: Structure of the compositional system

- Phase 1. Gas: $CO_2(g)$
- Phase 2. Aqueous solution: $H_2O, H^+, CO_2(aq), Cl^-, Na^+, Ca^{+2}, SiO_2(aq), HCO_3^-, OH^-$
- Phase 3. Calcite mineral: $CaCO_3(s)$
- Phase 4. Quartz mineral: $SiO_2(s)$

We consider the following three equilibrium reactions:

- Req 1. Hydrolysis of water

  $H_2O \rightleftharpoons H^+ + OH^-$

- Req 2. Dissolution of $CO_2(g)$ in water

  $CO_2(g) \rightleftharpoons CO_2(g)$

- Req.3 Dissociation of $CO_2(aq)$

  $H_2O + CO_2(aq) \rightleftharpoons HCO_3^- + H^+$

We model dissolution-precipitation reactions by kinetics:

- Rkin 1. Dissolution of Calcite

  $CaCO_3(s) + H^+ \rightarrow Ca^{2+} + HCO_3^-$

- Rkin2. Precipitation of Calcite

  $CaCO_3(s) + H^+ \leftarrow Ca^{2+} + HCO_3^-$

- Rkin3. Dissolution of Quartz

  $SiO_2(s) \rightarrow SiO_2(aq)$

- Rkin 4. Precipitation of Quartz

  $SiO_2(s) \leftarrow SiO_2(aq)$

As explained in section (3.6.1) we have to specify the laws to model the parameters. The activity of a chemical species is linked to its chemical potential so we propose to use ideal activities for every phase:

$$a_i = \begin{cases} x_i = \dfrac{n_{H_2O}}{\sum\limits_{i \in w} n_i} & i = H_2O \\ m_i = \dfrac{n_i}{n_{H_2O}} \dfrac{1}{18 \cdot 10^{-3}} & i \in w \neq H_2O \\ x_i = 1 & i \in M_1, M_2, G_1. \end{cases} \tag{4.4}$$

where w represents Aqueous solution, M_1 is Mineral Calcite phase, M_2 is Mineral Quartz phase and G_1 is Gas phase.

In the following tables we can see the parameters of the species (Table 4.2) and the reaction kinetic values (Table 4.3) that, in our model, are constant.

The problem considered has initial time $t_0 = 0s$ and final time $t_{final} = 300s$, we take as Newton tolerance $[1 \cdot 10^{-12}, 1 \cdot 10^{-10}]$; the tolerance for the error estimator is taken equal to $[1 \cdot 10^{-4}, 1 \cdot 10^{-5}]$, the initial time step is equal to $1 \cdot 10^{-4}$.

We can see in Figure (4.4) that we obtain equilibrium at time 125. The species with a initial high value and the ones not involved in any reaction stay constant, while the others increase or decrease their concentration.

Phase	Species	$log_{10}(k)$	initial concentration $[mol/m^3]$
Aqueous	H_2O	0	8071.20559
Aqueous	H^+	0	0.00377657
Aqueous	$CO_2(aq)$	0	119.23091
Aqueous	Cl^-	0	156.803187
Aqueous	Na^+	0	145.432629
Aqueous	Ca^{+2}	0	7.01164993
Aqueous	$SiO_2(aq)$	0	0.03770169
Aqueous	HCO_3^-	-6.2206340	2.62191679
Aqueous	OH^-	-13.235362	$3.091 \cdot 10^{-7}$
Mineral Calcite	$CaCO_3(s)$	-7.7454139	$4.33 \cdot 10^3$
Mineral Quartz	$SiO_2(s)$	3.5862160	$2.82 \cdot 10^4$
Gas	$CO_2(g)$	2.0861861	$4.27 \cdot 10^2$

Table 4.2: Thermodynamic parameters of the species

Reaction	Type	Name	$\mathcal{K}$
Req 1	Aqu	Hydrolisis of water	$1 \cdot 10^3$
Req 2	Gas-Aqu	Dissolution of $CO_2(g)$ in water	$1 \cdot 10^1$
Req 3	Aqu	Dissociation of $CO_2(aq)$	$1 \cdot 10^3$
Rkin 1	Min-Aqu	Dissolution of Calcite	$-1 \cdot 10^{-1}$
Rkin 2	Min-Aqu	Precipitation of Calcite	$1 \cdot 10^{-1}$
Rkin 3	Min-Aqu	Dissolution of Quartz	$-1 \cdot 10^{-1}$
Rkin 4	Min-Aqu	Precipitation of Quartz	$1 \cdot 10^{-1}$

Table 4.3: Thermodynamic parameters of the reactions

Method	Computation time	Numb. time steps
Singlerate TR-BDF2	$0.34s$	98
Multirate TR-BDF2	$0.33s$	178

Table 4.4: Computation time and numbers of time steps for the geochemistry problem.

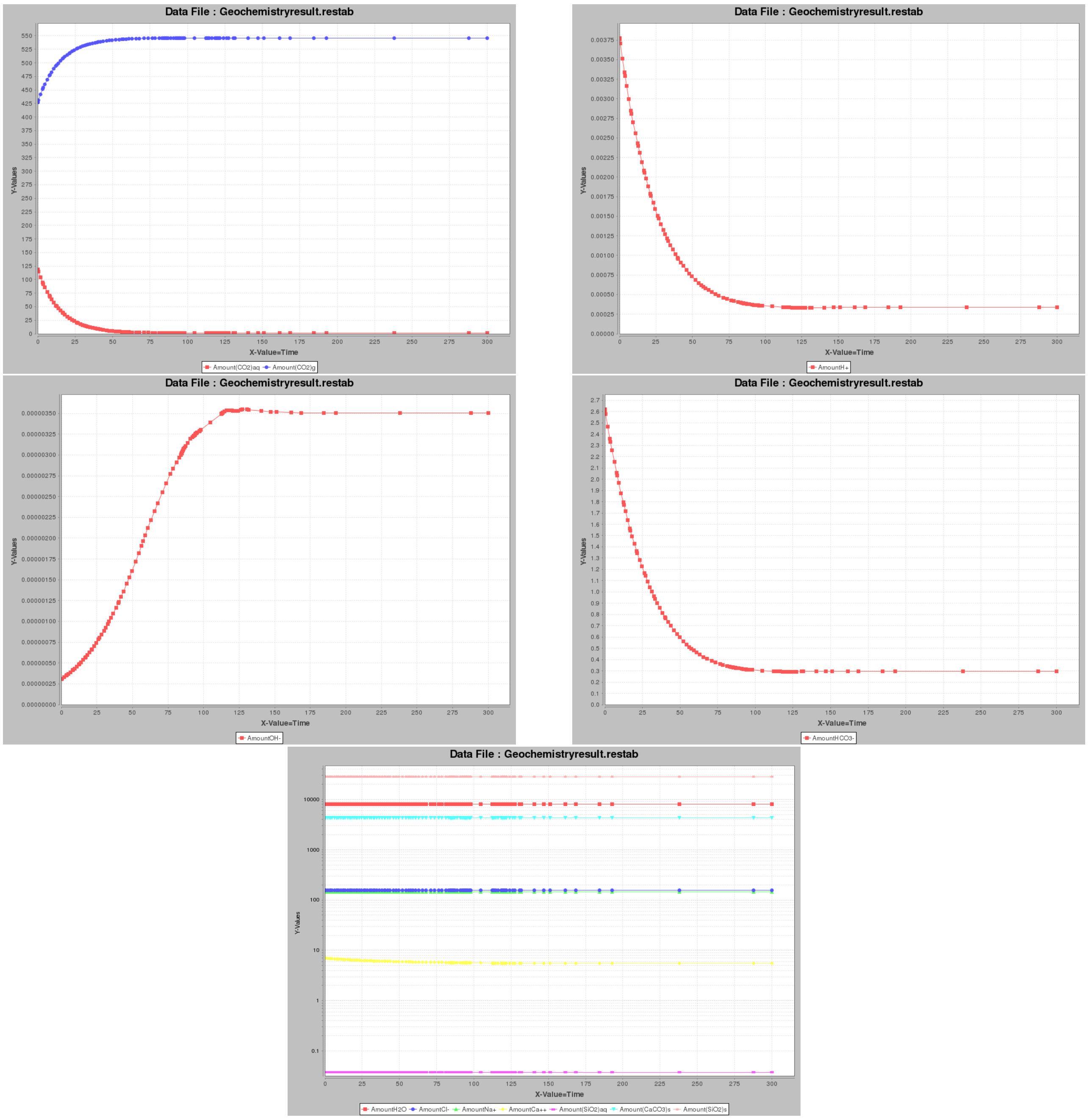


Figure 4.4: Amount (from the top to the bottom, from left to right) of $CO_2(g)$ and $CO_2(aq)$, H^+, OH^-, HCO_3^-, others species computed with the multirate method.

If we focus our attention to the OH^- species, we can see how the multirate interpolation causes a difference in the solution between the time $[75, 100]s$ from the single rate method (Figure (4.5)). In this

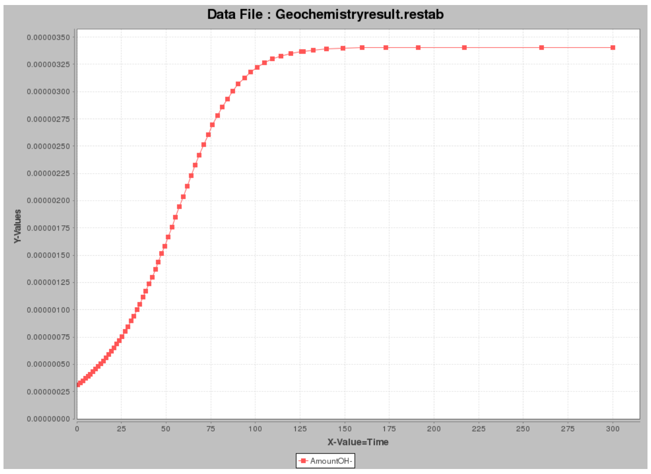


Figure 4.5: OH^- amount computed with singlerate TR-BDF2 method.

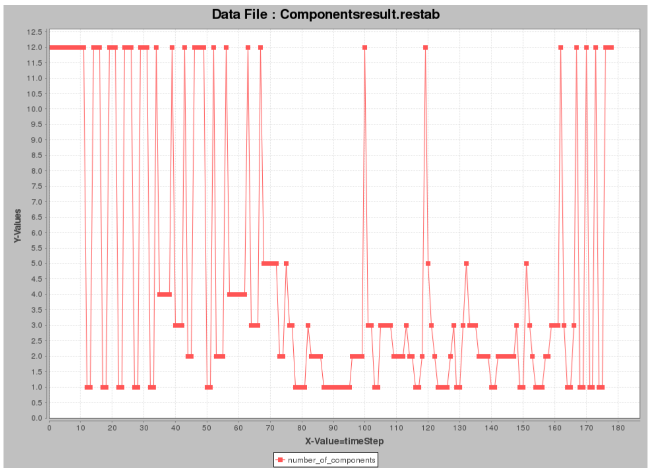


Figure 4.6: Number of components where the TR-BDF2 method is applied for each time step.

interval the multirate method interpolates the OH^- species that generates an error on the solution. In this test we can not see any advantage of the multirate method: the execution time is almost the same, but the multirate requires more iteration (Table (4.4)).

To test the performance of the multirate method in this problem we want to extend the species set. We add in Aqueous phase 100 species that will not interact with each other or with the previous ones and that will be present in no reaction. We set the initial concentration of these species equal to the Cl^- species value.

If we look the Table (4.5), there is not a substantial difference between the two methods, the computation time of the single rate method is bigger than the multirate but we do more iterations.

Method	Computation time	Numb. time steps
Singlerate TR-BDF2	$17.28s$	124
Multirate TR-BDF2	$10.69s$	273

Table 4.5: Computation time and numbers of time steps for the extended geochemistry problem.

Method	Computation time	Numb. time steps
Single rate TR-BDF2	$1000.41s$	425
Multirate TR-BDF2	$91.86s$	423

Table 4.6: Computation time and numbers of time steps for the transport equation.

4.3 Hyperbolic equation

We now test the method on another PDE problem:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \tag{4.5}$$

for $t > 0$ and $-20 < x < 20$.

We considered $a = 1$ so we use as boundary condition $u(x_0, t) = u_0(t_0)$, while as initial condition $y(x, 0) = \exp(-x^2)$ for $x \in [-20, 20]$.

The interval time is $[0, 3]s$ with a number of cells equal to 400, the error tolerance is $[1 \cdot 10^{-6}, 1 \cdot 10^{-8}]$. As initial size of the time step was taken it equal to $1 \cdot 10^{-2}$.

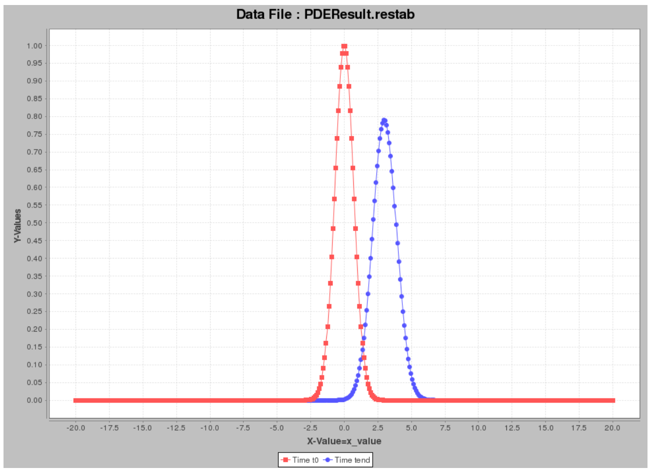


Figure 4.7: Initial and final solution computed with multirate TR-BDF2 method.

In Figure (4.7) is represented the solution for the initial and final time, the upwind discretization has a diffusion term which is responsible for the spreading of the initial condition. This diffusion also ensures that the number of computed components increases as time progresses (Figures (4.8) and (4.9)).

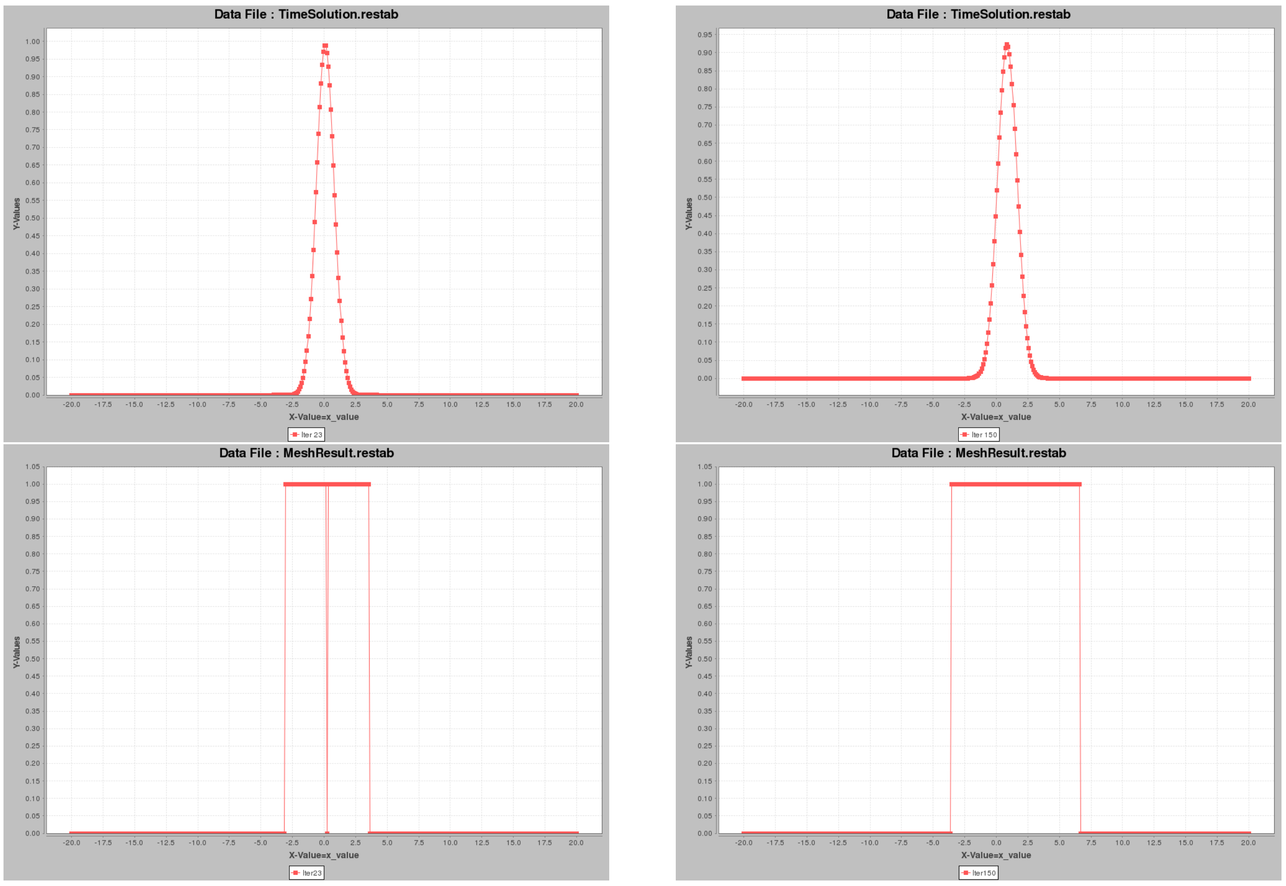


Figure 4.8: Solution with the subset of components that are computed at time 0.09s (left) and 0.87s (right).

In this test case we have mass conservation because we interpolate the solution only where it is constant. Generally this does not happen and interpolating it on non-constant values should lose a part of the mass as explained in [4].

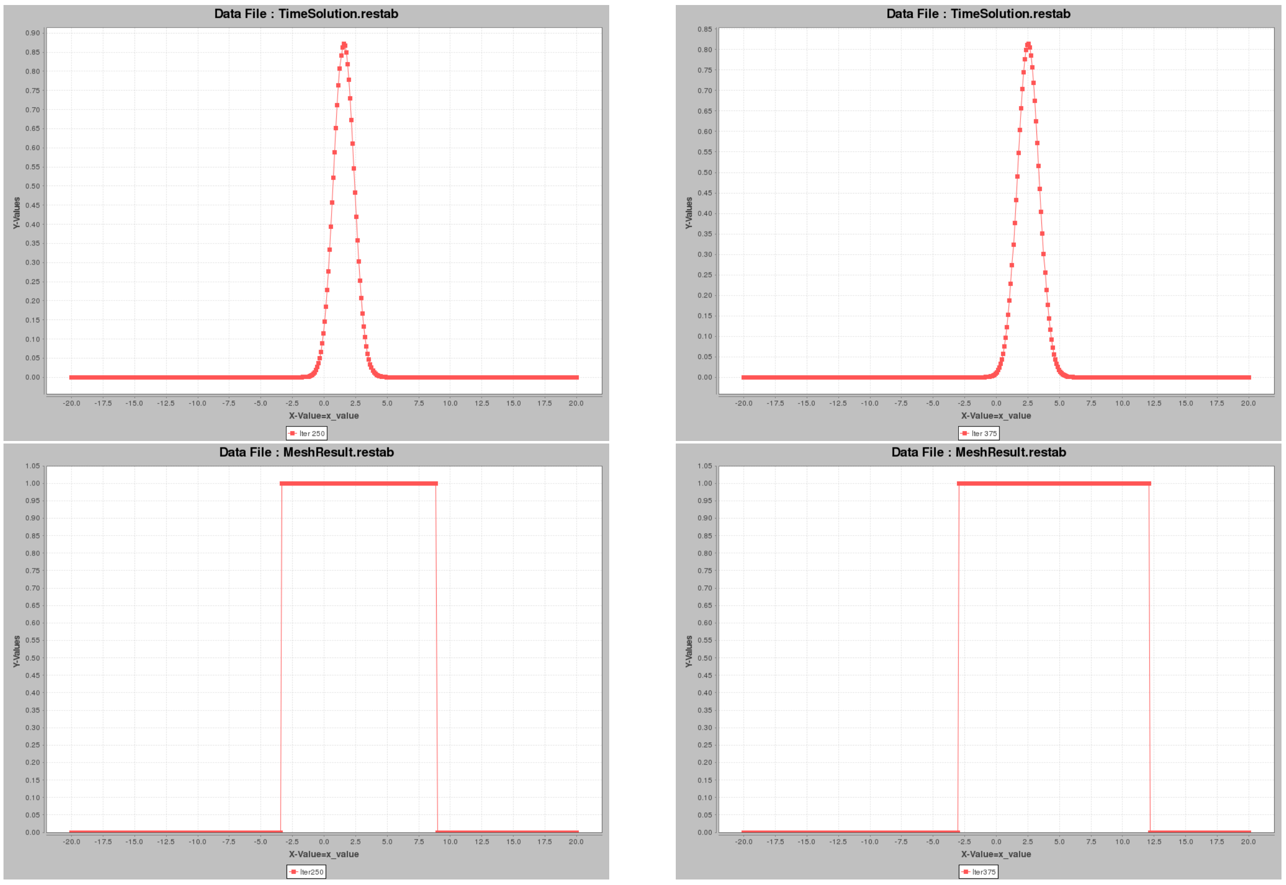


Figure 4.9: Solution with the subset of components that are computed at time 1.57s (left) and 2.53s (right).

Chapter 5

Further developments

In this work, we have implemented some algorithms to integrate in time a ODE system. Some classes were developed to solve the problem with our specific methods, but they could be extended to give generality at the solving techniques. For example, it would be interesting investigate the problems resolution with different interpolators creating a interface with inherit also for it. Another important further work is to generalize the boundary condition for the Hyperbolic problem, it could be useful to implemented also the Newman condition. For smooth transitions it would be very useful created a parser for the initial conditions, in this way it is not necessary to recompiling the code every time that we want to do some modification to the problem. A interesting developments could be to parallelizing the code to have better computation time performances.
We are working on a new hyperbolic test case with a non linear flux to show the multirate method does not conserve mass.

Appendix A

Tutorial to compile and run the code

GetPot library allows the parsing of options specified by command line or by configuration files at runtime. In particular, in the present project it has been used to select the MasterSolver, the Newton Solver's type, and which kind of problem with the respective parameters we want to solve. A configuration file has been created (data.pot), which contains all of them.

Listing A.1: data.pot

```
[common_parameters]
   rhs             = AllenCahn                 # Geochemistry, HyperbolicEquationUpwind,
                                               # AllenCahn
   Solver          = MultiRateTRBDF2Solver     # MultiRateTRBDF2Solver, SingleRateTRBDF2Solver,
                                               # SingleRateBESolver, SingleFESolver
   NewtonSolver    = FixedJacobianNewtonSolver # BasicNewtonSolver, FixedJacobianNewtonSolver
   Newton_tol_abs   = 1.e-11
   Newton_tol_rel   = 1.e-11
   Error_tol_rel    = 1.e-4
   Error_tol_abs    = 1.e-6
   delta_t          = 1.e-1
   tInitial         = 0.0
   tFinal           = 142.0
[../]

[Geochemistry]

[../]

[HyperbolicEquationUpwind]
   nx         = 400
   x0         = -20.0
   xL         = 20.0
   BC_value   = 0.0        # Boundary condition value
   alpha      = 1.0        # velocity hyperbolic equation
[../]

[AllenCahn]
   nx     = 400
   x0     = -1.0
   xL     = 2.0
   sigma  = 9.e-4;
[../]
```

For the GeochemistryRHSFunction all parameters that we need to define the problem are in two header file:

- SetProblemSHPCO2SourceRate where there are the initial conditions, the set of species, reactions and phases of the system and, if it is present, the source rate terms for the the base case test(as we have showed in section (3.6.1)),
- SetProblemGeochemEX where we set the same parameters for the extended test case.

To set the initial conditions of the Allen-Cahn problem and the hyperbolic equation, we have to define the function $u(x, t_0) = u_0(x)$. Inside the folder TestCases are present:

- SetProblemAllenCahn an header file where we have implemented the function (4.3).
- SetProblemLinearHyperbolicEquation where we can found two different initial conditions for the pde problem.

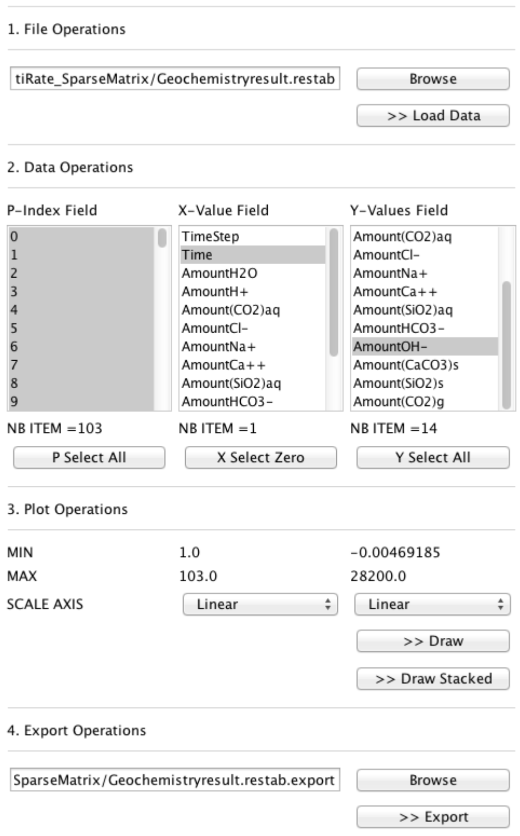
Appendix B

Data visualization

To visualize the data we used the DataExplorer.jar program. To launch it from the terminal the command is:

```
>> java -jar DataExplorer.jar
```

To choose the .restab file that we want to plot, click on Browse in the upper right and select it. Than click on »Load Data, select the x-values and the y-values and then click on » Draw.



1. File Operations

tiRate_SparseMatrix/Geochemistryresult.restab

2. Data Operations

P-Index Field	X-Value Field	Y-Values Field
0	TimeStep	Amount(CO2)aq
1	Time	AmountCl-
2	AmountH2O	AmountNa+
3	AmountH+	AmountCa++
4	Amount(CO2)aq	Amount(SiO2)aq
5	AmountCl-	AmountHCO3-
6	AmountNa+	AmountOH-
7	AmountCa++	Amount(CaCO3)s
8	Amount(SiO2)aq	Amount(SiO2)s
9	AmountHCO3-	Amount(CO2)g
NB ITEM =103	NB ITEM =1	NB ITEM =14

3. Plot Operations

MIN	1.0	-0.00469185
MAX	103.0	28200.0
SCALE AXIS	Linear	Linear

4. Export Operations

SparseMatrix/Geochemistryresult.restab.export

Bibliography

[1] M.E. Hosea, L.F. Shampine (1996), Analysis and implementation of TR-BDF2, *Applied Numerical Mathematics*, vol. 20, 21-37.

[2] P.W Fok (2015), A linearly fourth order multirate Runge-Kutta method with error control, *J. Sci. Comput.*, vol. 66(1), 177-195.

[3] W. Hundsdorfer, V. Savcenco, Analysis of a multirate theta-method for stiff ODEs, *J. Appl. Num. Math.*

[4] W. Hundsdorfer, A. Mozartova, V. Savcenco (2007), Analysis of explicit multirate and partitioned Runge-Kutta schemes for conservation laws, *CWI report, MAS-E0715.*

[5] A. Quarteroni (2008), *Modellistica numerica per problemi differenziali*, Springer.

[6] A. Quarteroni, R. Sacco, and F. Saleri (2007), *Numerical Mathematics*, vol. 37, Springer.

[7] A. Ranade, (2015) Multirate algorithms based on DIRK methods for large scale system simulation, *PhD thesis.* Politecnco di Milano.

[8] V. Savcenco, W. Hundsdorfer, J.G. Verwer, (2007), A multirate time stepping strategy for stiff ordinary differential equations, *BIT 47*, 137-155.