Implementing Runge–Kutta schemes for reusability*

Krister Åhlander†

Abstract

Four different strategies for implementing Runge–Kutta time stepping schemes are discussed. Explicit Runge–Kutta, ERK, schemes are studied, but the results can be generalized to diagonally implicit Runge–Kutta methods as well. The four different strategies combine the concepts of forward and reverse communication with and without a state machine approach. The idea of using state machines is novel.

The different implementation strategies are studied for a model problem, in a parallel multi-block grid code, and in Compose, a PDE solving environment, which uses overlapping grids and finite differences. It is found that the state machine approach does not degrade performance.

The state machine approach with forward communication is advantageous in the context of a general PDE solving environment. The reason is that this approach combines the flexibility of reverse communication with the simplicity of forward communication, which allows for a high reusability. The incompressible Navier–Stokes equations are shown as a case where a state machine ERK component is reusable, whereas a traditional implementation with forward communication would require modifications in the ERK implementation.

The conclusion is that the the novel ERK communication strategies are comparable with respect to performance. The choice of implementation may therefore be driven by other objectives, such as reusability. In this respect, the state machine implementation is advantantageous.

Keywords: object-oriented, PDE, Navier–Stokes, Runge–Kutta

1 Introduction

When developing general numerical software, two well-known strategies for handling the communication between the calling program and the reusable component (traditionally just a function), are *forward* communication and *reverse* communication. Forward communication implies a calling sequence where a

*Supported by the Swedish Research Council for Engineering Sciences

†Department of Scientific Computing, Uppsala University, email: krister@tdb.uu.se

component itself calls other components to carry out sub-tasks. Reverse communication returns the thread of control to the calling program, when intermediate results are needed. Generally speaking, reverse communication allows for more flexibility than forward communication, at the price of a more complicated user interface [8]. In [7], it is formulated: "Reverse communication has the advantage of increasing the functionality of the integrator, but it complicates the structure of the calling program". Our work is motivated by the search for reusable software components that combine the flexibility of reverse communication with the easy usage of forward communication, without degrading the performance. This kind of reusable building blocks are important for general PDE solving environments, e.g. our research project Compose [1, 2]. In this paper, we use the term "component" to refer to a piece of code that is intended for reuse, either an object, as in object-oriented programming, or a function, as in structured programming.

In this paper, we investigate four different communication strategies in the context of developing reusable time integration components of explicit Runge–Kutta (ERK) type. These components are intended to solve partial differential equations, PDEs, with the Method of Lines. We particularly study two apparently new variations of implementing ERK methods, where we utilize a state machine approach for implementing the dynamic behaviour. We have not seen state machines for ERK methods elsewhere in the literature, even though there exist many projects with a scope similar to ours, e.g. [3, 6, 11].

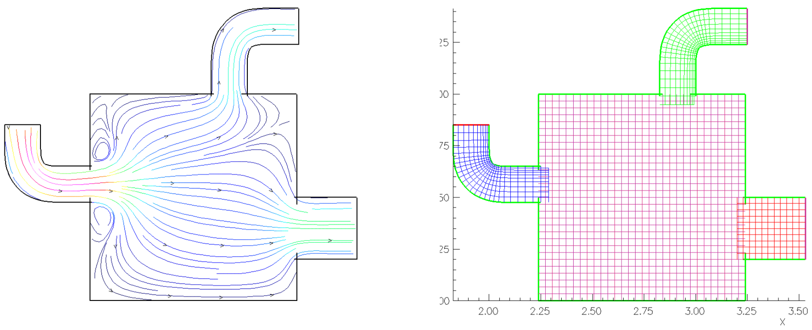
Our present study is further motivated in Section 2. In Section 3, we give a more detailed presentation of the different strategies for implementing explicit Runge-Kutta methods: forward communication, reverse communication, forward communication as a state machine, and reverse communication as a state machine. The generalization to diagonally implicit Runge–Kutta methods is outlined. In Section 4, we compare these approaches with respect to performance, and it is found that all strategies are comparable. A model problem is used in order to illustrate the ideas without unnecessary details and to make the comparisons between the approaches as fair as possible. In Section 5, we study more realistic problems. First, we investigate the effect of replacing the traditional ERK scheme with the state machine approach in an existing, parallel, Navier–Stokes solver. Here, we solve the conservative Navier–Stokes equations for a compressible fluid. The program is written in Fortran 90 and executed in parallel with the aid of the Cogito software tools [13]. Then, we make comparisons between a traditional ERK component and the state machine approach in the context of reusable components in the Compose framework. As a typical example of where our approach is advantageous, we present a solver for the incompressible Navier–Stokes equations, the equations of motion for an incompressible fluid. The main point is that the time stepping method, due to the choice of interface design, is *independent* of the elliptic constraint that we apply in each Runge–Kutta stage.

The conclusion is that, among the investigated ways of implementing ERK methods, the state machine approach with forward communication is the most advantageous. This is motivated by a high degree of flexibility, and thus reusability. Moreover, our numerical results show that it does not degrade performance.

2 Motivation

The main incentive for this study is to investigate the design of ERK time stepping methods as reusable software components in a framework or a library for PDEs. An example of such a framework is our own research project Compose [1, 2], which is designed as an object-oriented framework, extendable via inheritance. For the discussion here, object-orientation is not particularly relevant, and most of our examples are therefore given in Fortran 90.

As a motivating example, we show in Figure 1a a numerical solution of the incompressible Navier–Stokes equations for fluid motion in an interior domain. In Figure 1b, we show the composite grid that cover the domain. Even if this is not a real-world application, it is a good illustration of the kinds of problems that we have in mind.



(a) Solution

(b) Grid

Figure 1: Fluid streamlines in a chamber with one inlet and two outlets. The overlapping grid is generated with the Overture tool Ogen [10].

Reusable time marching methods for this kind of problems have to meet a number of demands. We emphasize the following issues:

Equation dependencies The ERK component is dependent on the Equation, which can imply restrictions on the generality. For example, it is not easy to foresee all possible sets of input to a general right hand side f in the equation $u_t = f(\ldots)$. Thus, if f is coded as a function, it is not easy to choose parameters. The ERK component is dependent on the function interface, which can lead to restrictions on the set of solvable problems. Forward communication in the traditional sense can not easily deal with this, which motivates the search for alternatives.

Algorithmic variations Different contexts require algorithmic variations of the ERK method, for instance when applying boundary conditions. With careful design, forward communication can handle some of these variations, but in Section 5 we show examples which are more difficult. Again, alternatives to traditional forward communication are sought.

Moreover, an ERK component should preferrably be independent from the geometry and from the choice of space differentiation. In our present study, we assume that this is addressed by abstractions on lower levels. For example, COMPOSE is built with the aid of Overture [5], a class library with abstractions for overlapping grids. The ERK component relies on the data abstractions on a lower level, such as `Grids` and `Grid Functions`. These issues are therefore not in focus for the present study.

The design of ERK methods in a general setting is an important issue, since we want a flexible and general component with a clear interface, but without inherent performance inefficiencies. In the remainder of this paper, we present and study different alternatives for implementing ERK methods, including two novel strategies that utilize state machines.

3 Algorithm description

We present a number of implementations of a simple ERK. The purpose is to show the differences between the different kinds of communication between the Runge–Kutta method and the calling program. We use Fortran 90 for demonstrating and comparing different algorithms.

A general ERK algorithm may be written:

$$\begin{aligned} k_1 &= f(t_0, y_0) \\ &\dots \\ k_s &= f(t_0 + c_s h, y_0 + h(a_{s,1}k_1 + \dots + a_{s,s-1k_{s-1}})) \\ y_1 &= y_0 + h(b_1 k_1 + \dots + b_s k_s) \end{aligned} \tag{1}$$

The algorithm examples in this section are given for a simple three stage Runge–Kutta scheme (Heuns method of order 3, see i.e. [9]).

$$\begin{aligned} k_1 &= f(t_0, y_0) \\ k_2 &= f(t_0 + 1/3h, y_0 + 1/3hk_1) \\ k_3 &= f(t_0 + 2/3h, y_0 + 2/3hk_2) \\ y_1 &= y_0 + h(1/4k_1 + 3/4k_3) \end{aligned} \tag{2}$$

The generalization to other schemes are straightforward, and in Section 5 we use other ERK schemes as well.

A PDE on semi-discrete form

$$u_t = Q(u)$$

is solved. In our example, Q is an operator for the equation, including space discretizations and boundary conditions. It is coded in a function `Qu(u,t)`. A more general setting would naturally require a more general interface, but we use this simple example for clarity.

3.1 Forward communication

Forward communication has the longest history of the interface variations we discuss. Here, the function interface of the right hand side has to be known to the subroutine that performs the time integration.

A sample implementation is given below.

```
subroutine trad1RKadvance(u,t,h)
  real, dimension(:), intent(inout)     :: u   ! solution
  real, intent(in)                      :: t   ! time
  real, intent(in)                      :: h   ! time step

  real, dimension(size(u))              :: k1, k2, k3

  k1 = Qu(u,t)
  k2 = Qu(u + (1.0/3)*h*k1, t + (1.0/3)*h)
  k3 = Qu(u + (2.0/3)*h*k2, t + (2.0/3)*h)

  u = u + h*(0.25*k1 + 0.75*k3)
end subroutine trad1RKadvance
```

A straightforward calling sequence is exemplified below.

```
do i = 1,iNrIt
   call trad1RKadvance(u,t,dt)
   t = t + dt;
end do
```

3.2 Reverse communication

Reverse communication is a well-known technique, and for a time integration routine it implies that the subroutine returns to the calling program whenever the right-hand side needs to be updated. We notice that reverse communication yields a more flexible ERK component, since it is less dependent on other components (such as the right hand side function).

A sample implementation is given below. Here, we have chosen to declare the temporary variables k_i outside the function, but within the module. Allocation and deallocation of these are taken care of by constructor and destructor like function calls. The stage is determined by the corresponding variable.

```
subroutine reverseRKadvance(u,t,h,qu,stage)
  real, dimension(0:), intent(inout)    :: u   ! solution
  real, intent(inout)                   :: t   ! time
  real, intent(in)                      :: h   ! time step
  real, dimension(0:), intent(in)       :: qu  ! Q(u)-value
  integer, intent(inout)                :: stage

  if (stage == 1) then
     uOld = u
```

```
      tOld = t
      ! k1 = Qu(u,t)
      k1 = qu ! used in stage 3
      u = uOld + (1.0/3)*h*qu
      t = tOld + (1.0/3)*h
      stage = 2
   else if (stage == 2) then
      ! k2 = Qu(u,t)
      u = uOld + (2.0/3)*h*qu
      t = tOld + (2.0/3)*h
      stage = 3
   else
      ! k3 = Qu(u,t)
      u = uOld + h*(0.25*k1 + 0.75*qu)
      t = tOld + h
      stage = 1
   end if
end subroutine reverseRKadvance
```

A typical calling sequence is given below. Note that the main program is responsible for calculating the right hand side, and the stage has to be communicated back and forth. The number of loop iterations becomes multiplied with the number of stages, in this case three.

```
  stage=1
  do i = 1,iNrStages*iNrIt
     q = Qu(u,t)
     call reverseRKadvance(u,t,dt,q,stage)
  end do
```

3.3 State machine with reverse communication

The implemention of reverse communication is simplified if the ERK algorithm (1) is interpreted as a state machine. Each stage in the ERK algorithm may be interpreted as a state in the state machine. Carrying out this idea, we implement the ERK as an *object* (or in a Fortran 90 module), with a temporal behaviour according to the state machine of Figure 2, plus extra operations for proper initialization and so forth. Compared with the previous example, the interface to the time integration method becomes cleaner.

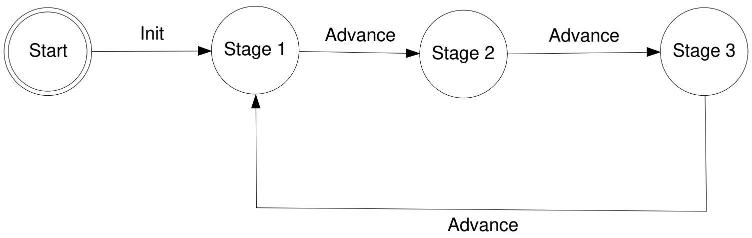


Figure 2: The three stage ERK algorithm may be described as a state machine. After initialization, each subsequent call advances a stage.

A sample implementation is given below.

```
subroutine stateReverseRKadvance(u,t,h,qu)
  real, dimension(0:), intent(inout)    :: u   ! solution
  real, intent(inout)                   :: t   ! time
  real, intent(in)                      :: h   ! time step
  real, dimension(0:), intent(in)       :: qu  ! Q(u)-value

  if (state == 1) then   ! Remark: state is declared in the module
     uOld = u
     tOld = t
     !k1 = Qu(u,t)
     k1 = qu ! used in stage 3
     u = uOld + (1.0/3)*h*k1
     t = tOld + (1.0/3)*h
     state = 2
  elseif (state == 2) then
     !k2 = Qu(u,t)
     u = uOld + (2.0/3)*h*qu
     t = tOld + (2.0/3)*h
     state = 3
  else
     !k3 = Qu(u,t)
     u = uOld + h*(0.25*k1 + 0.75*qu)
     t = tOld + h
     state = 1
  end if

end subroutine stateReverseRKadvance
```

The calling sequence exemplified below bears many similarities with the previous example. The difference is that the stage is internal to the component.

```
  do i = 1,iNrStages*iNrIt
     q = Qu(u,t)
     call stateReverseRKadvance(u,t,dt,q)
  end do
```

3.4 State machine with forward communication

Combining the state machine approach with forward communication is yet another possibility. Here, we use forward communication in the sense that the ERK component call the right hand side component, but we implement the algorithm with a state machine approach. The result is a calling sequence which is easier than for reverse communication, and more flexible than for forward communication.

A sample implementation is given below.

```
subroutine stateRKadvance(u,t,h)
  real, dimension(0:), intent(inout)    :: u   ! solution
  real, intent(inout)                   :: t   ! time
  real, intent(in)                      :: h   ! time step

  if (state == 1) then
     uOld = u
     tOld = t
     k1 = Qu(u,t)
```

```
    u = uOld + (1.0/3)*h*k1
    t = tOld + (1.0/3)*h
    state = 2
  elseif (state == 2) then
    k2 = Qu(u,t)
    u = uOld + (2.0/3)*h*k2
    t = tOld + (2.0/3)*h
    state = 3
  else
    k3 = Qu(u,t)
    u = uOld + h*(0.25*k1 + 0.75*k3)
    t = tOld + h
    state = 1
  end if

end subroutine stateRKadvance
```

The calling of this ERK component is simple. Note that the main program optionally can do calculations inbetween the ERK stages, which enhances flexibility.

```
  do i = 1,iNrStages*iNrIt
    call stateRKadvance(u,t,dt)
  end do
```

3.5 Algorithm summary

We have presented four basic kinds of variations of the implementation of an ERK component. Forward communication is the most straightforward alternative, but it lacks in flexibility since the algorithm is fixed and since the right hand side has to have an interface known in advance. Reverse communication remedies both of these shortcomings, on behalf of a more complicated interface.

The novel state machine approach yields a simpler interface for reverse communication, since the component itself knows its state. Forward communication may also be implemented as a state machine, which combines a simple user interface with algorithmic flexibility. Regarding the dependence on the right hand side interface, reverse communication is more flexible than forward communication. But, as detailed in [1], and exemplified in Section 5, another way to address this issue is to treat the right hand side as an object. If this approach is taken, the state machine with forward communication is as flexible as the reverse communication alternatives.

The different approaches generalize to diagonally implicit Runge–Kutta methods, since the stages are independent. For fully implicit Runge–Kutta methods, where the stages are coupled, it is obviosly not possible to treat each step separately.

Given the motivations in Section 2, the different implementation variations meet the demands with different success. State machine implementations or reverse communication are better off when the context of the ERK component changes. The question is, then, do the different implementations affect performance?

4 Comparisons for a model problem

In the previous section, we have presented four different ways of implementing a simple ERK scheme. In this chapter, we compare these strategies with respect to performance. This kind of comparison is very delicate, since it is difficult or almost impossible to isolate the "true" differences between the strategies. Therefore, as described below, we compare a few variations of implementing the algorithms. The purpose is to isolate the comparison of the algorithms from effects that depend merely on variuos implementation details.

4.1 Experiment description

In order to compare the performance of the algorithms we solve $u_t = u_x$ in the domain $x \in [0, 1]$ with periodic boundaries. The algorithms from the previous chapter are used. It is impossible to carry out tests of algorithms independent of the language that we use for implementation, and in our study we also investigated some variations with respect to the declaration of local temporary variables. The objective is to find out if certain programming techniques are favourable for the compiler. Below, we describe the different algorithm implementations.

Forward The forward algorithm as in the previous section, with local variables declared in the time integration routine. This is the usual way to implement the traditional algorithm.

Reverse The reverse algorithm as in the previous section. Local variables are declared in the module and they are allocated by a "constructor" call.

State Reverse The reverse state algorithm as in the previous section. Local variables are declared in the module and they are allocated by a "constructor" call.

State Forward The forward state algorithm as in the previous section. Local variables are declared in the module and they are allocated by a "constructor" call.

Forward Module The forward algorithm as before, but with local variables declared in the module and allocated by a "constructor" call. The objective is to use the same conditions for memory usage as in the three previous algorithms.

Forward Fixed The traditional algorithm with fixed size data arrays, and no need for dynamic allocation. The drawback is of course that the program has to be recompiled when the size of the data arrays change. Here, we want to isolate the effect of dynamic allocation.

Reverse Fixed The reverse algorithm with fixed size data arrays.

State Fixed The forward state algorithm with fixed size data arrays.

The experiments are carried out for three problem sizes, $n = 10, 100$ and 1000, where each execution uses n time steps and data arrays of size n in the x-dimension. We use a Sun Ultra 1 workstation with the native `CC` compiler. Two compiler flag settings are used, with and without (`-fast`) optimization. The purpose is to study the different implementation's sensitivity to compiler optimization. In Section 5, we do experiments on other computers.

The alternatives with fixed data arrays are included in this study in order to get an idea about the importance of memory layout. We do not find it likely that people want to implement ERK components with fixed data arrays, even if these implementations are faster than their dynamic counterparts.

4.2 Results

In Tables 1 and 2, the non-optimized and the optimized results are shown, respectively. The optimization efficiency, i.e. the quota between a non-optimized figure and a corresponding optimized figure, is shown in Table 3.

Some general conclusions can be drawn from these tables.

1. The choice of interface between the algorithms described in Section 3 has little effect on the performance. For different problem sizes, different algorithm choices yield the best performance.

2. The memory layout is of importance. The usage of fixed arrays gives better performance than corresponding alternative with dynamic memory allocation. The improvement is actually not very high, though. This is encouraging since fixed size arrays are not the ideal choice for flexible programs.

3. Comparing the traditional Forward algorithm and the Forward Module algorithm shows small differences in performance. It seems that the usage of temporaries as local module variables is a competitive alternative to local function variables.

4. The optimization affects the algorithms very differently. Note, in Table 3, the difference for size 100 between Forward Fixed (1.6) and Reverse Fixed (9.4)! An explanation might be that Forward Fixed happened to have a favorable memory layout even for non optimized code.

The objective of this study is not to find the fastest possible way to implement ERK algorithms, but to find suitable alternatives for ERK components in a general setting. The results in this section indicate that none of the studied algorithm variations has a decisive lead in performance. The best choice for optimized code with dynamic memory is actually, in this specific case, the State Forward algorithm.

n	10	100	1000
Forward	0.0026	0.15	15
Reverse	0.0032	0.16	15
State Reverse	0.0025	0.15	15
State Forward	0.0027	0.14	14
Forward Module	0.0023	0.12	11
Forward Fixed	0.0014	0.09	10
Reverse Fixed	0.0014	0.14	14
State Fixed	0.0015	0.09	11

Table 1: Non optimized results. The increase of work between each column is 100, since the work is proportional to n^2. Figures represent cpu time.

n	10	100	1000
Forward	0.0007	0.025	2.3
Reverse	0.0006	0.045	2.2
State Reverse	0.0007	0.022	2.2
State Forward	**0.0006**	**0.021**	**2.2**
Forward Module	0.0009	0.057	2.3
Forward Fixed	0.0003	0.056	1.8
State Fixed	0.0006	0.017	2.1
Reverse Fixed	0.0003	0.015	2.0

Table 2: Optimized results. In our particular experiment, the State Forward algorithm seems like the best choice when dynamic memory is used. Figures represent cpu time.

5 Components in complicated calculations

In the previous section, we used a model problem in order to compare all communication variations. In this section, we show that the novel state approach is not restricted to simple model problems, and that it is competitive for more complicated calculations as well.

Our first example investigates the effect of replacing the Forward time stepping integration in a solver for the compressible Navier–Stokes equations with the state machine variant with forward communication. This program is written in Fortran 90 and utilizes Cogito tools [13] so that it can be executed in parallel on a patched grid. The purpose is to show that our approach is competitive with respect to performance.

Second, we study the difference in performance between a state machine ERK component and a traditional forward communication component in the Compose framework. As mentioned in Section 2, Compose is a set of high level C++ classes which is designed particularly with reusability aspects in focus. In this framework, we show that performance of the state machine approach is comparable with the performance of a traditional forward implementation. In

n	10	100	1000
Forward	3.5	5.9	6.3
Reverse	5.3	3.5	6.7
State Reverse	3.5	6.6	6.7
State Forward	4.8	6.7	6.3
Forward Module	2.6	2.1	5.1
Forward Fixed	4.1	1.6	5.2
State Fixed	2.6	5.3	5.1
Reverse Fixed	4.6	9.3	6.8

Table 3: Optimization efficiency. The quota between non-optimized and optimized performance is shown. Variations are large.

fact, the state machine is slightly faster. Finally, we emphasize that the forward communication approach is not reusable when the elliptic constraint of the incompressible Navier–Stokes equation is taken into account. The state machine approach, on the other hand, can be reused without changing its implementation.

5.1 Comparisons on a multi-block grid and with parallel execution

Can the state machine approach be used without severe loss of performance for solving a PDE on a complicated geometry in parallel? The experiments shown in this subsection intend to investigate that question. We have studied the effects of replacing the time integration routine in an existing solver with the state machine approach with forward communication.

The existing program utilizes Cogito tools and is described in more detail in [12]. We calculate the 2D compressible Navier–Stokes equations in the geometry shown in Figure 3. The geometry is discretized with five curvilinear grids. Each grid has a size of 17×21 grid points. A Runge–Kutta method with five stages is used.

Table 4 summarizes the result. The executions were carried out on a DEC-alpha machine. We advance a 100 time steps, and compare the elapsed time for the traditional approach with the time for the state approach.

Regardless of the number of processors, the new state machine approach yields approximately the same performance as the traditional calling sequence. Since this is a problem of modest size, the speed-up is low. However, Cogito has been shown to have a good scalability [12].

5.2 Time stepping components in a broader perspective

The main concern for this study is to investigate the design of time stepping methods as reusable software parts in a broad perspective. COMPOSE is a software project which emphasizes the flexibility and reusability of components for

N:o of processors	1	2	4	8
Original, forward communication	39.9	24.5	18.8	14.9
State Forward communication	39.9	24.5	18.7	14.90

Table 4: The new State forward approach performs as good as the original, forward calling implementation. The number of processors is irrelevant for this observation.

PDE solvers [1, 2]. Compose is written in C++ and uses inheritance intensively, in order to decrease the burden of developing new PDE solvers. One main issue is that it should be easy to combine different PDEs with different numerical methods, and that equations as well as numerical methods shall be reusable in different contexts. In this section we demonstrate the last issue, focused on ERK time stepping methods.

In Compose, explicit time stepping methods and equations are derived through inheritance as shown in Figure 4. The new class implements the time stepping method. Time stepping objects know which equation to solve. The connections to other objects are taken care of by the ancestor `Time Equation Discretizer` and other parts of the framework. Code for deriving `Predictor Corrector`—an ERK scheme also known as modified Euler—is shown below.

```
ErrorFlag PredictorCorrector::advance()
{
  eq->updateExplicitRhs( rhs, thisLevelTime );

  switch( state ) {

  case predictor:
    // Store the current U for use in corrector
    Uold = U;

    // Take a half time step.
    U += (0.5*dt)*rhs;

    state = corrector; // next call, corrector state will be done
```

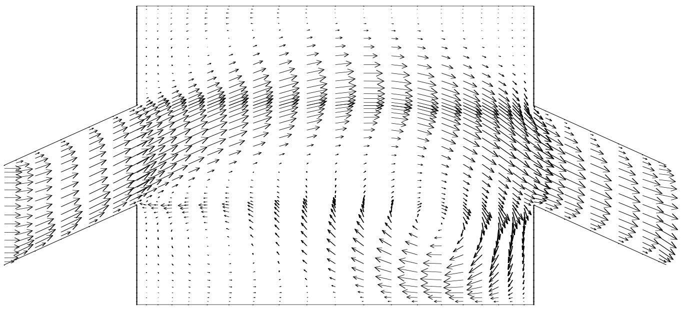


Figure 3: The parallel executions simulate airflow in this geometry, an expanding contracting tube. Here, we show streamlines.

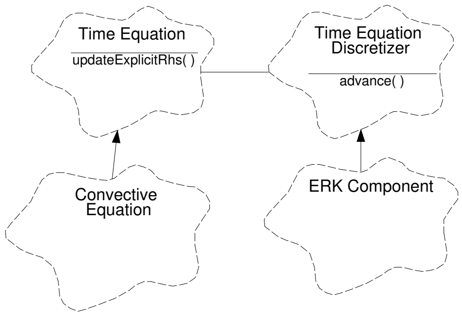


Figure 4: Concrete equations inherit from `Time Equation`. The subclass implements the operation `updateExplicitRhs`, which corresponds to the function `Qu` in earlier examples. Concrete time stepping methods inherit from `Time Equation Discretizer` and implement the operation `advance`. (Notation according to Booch [4].)

```
    break;

  case corrector:

    // Take a full time step.
    U = Uold + dt*rhs;

    state = predictor; // next  call, predictor state will be done
    break;
  default:
    error.assertion( false,
     "PredictorCorrector::advance: unknown state!");
  }

  return error.status();
}
```

We want to compare this implementation with a traditional forward communication scheme. As a model example, we solve the following system of examples, on a square with periodic boundary conditions.

$$\left\{ \begin{array}{l} u_t = a_{11}u_x + a_{12}v_x + b_1 u_{xx} \\ v_t = a_{21}u_x + a_{22}v_x + b_2 v_{xx} \end{array} \right. \tag{3}$$

A traditional way of implementing would yield the standard scheme shown below.

```
ErrorFlag StandardPredictorCorrector::advance()
{
  eq->updateExplicitRhs( rhs, thisLevelTime );

  Uold = U;

  // Take a half time step.
  U += (0.5*dt)*rhs;
```

```
  U.finishBoundaryConditions(); // Update periodic BCs if any.

  eq->updateExplicitRhs( rhs, thisLevelTime+0.5*dt );

  // Take a full time step.
  U = Uold + dt*rhs;

  return error.status();
}
```

One difference is that the `Standard Predictor Corrector` needs to treat the boundary conditions after stage 1, whereas the `Predictor Corrector` leaves this to the framework. The call `U.finishBoundaryConditions();` will update the periodic boundary conditions. The `Standard Predictor Corrector` is therefore applicable as is only for a limited range of problems, those with periodic boundaries as in our model example. Here, we use it for comparing the two techniques with respect to efficiency. Table 5 shows that the performance is insensitive to which communication scheme we use, and that this is irrespective of the increase in work, both with respect to the number of iterations and memory usage.

Size (n)	40	80	160	320
Forward State	0.45	3.0	45	183
Forward Traditional	0.45	3.0	45	183

Table 5: The work is proportional to n^3. The figures are in seconds, and show that also in a more complicated framework, traditional Forward communication and State Forward communication are equivalent with respect to performance.

In this particular example, we note that the application of boundary conditions is a detail which makes it more difficult to adapt the traditional strategy to different contexts. A more careful algorithmic design could overcome this problem, by calling a more general routine for applying boundary conditions, instead of just applying periodic boundary conditions. In our next example we demonstrate the benefits of the state machine strategy for a more complicated case.

The incompressible Navier–Stokes equations model the flow of an incompressible fluid. In two spatial dimensions, they can be written in the following form, with p as pressure, and with u and v as velocities in the x and y directions respectively:

$$\left\{ \begin{array}{rcl} u_t + uu_x + vu_y + p_x & = & \nu \triangle u \\ v_t + uv_x + vv_y + p_y & = & \nu \triangle v \\ \triangle p & = & -(u_x^2 + 2v_x u_y + v_y^2) \end{array} \right. \tag{4}$$

Here, the convective equations involve a time derivative, and it is possible to solve them with an explicit time stepping method. The pressure equation is sometimes referred to as an elliptic constraint. If we are to use the `Predictor`

`Corrector` scheme above, we want to apply this constraint at each stage. This is achieved by composing the objects according to Figure 5. Each equation that is needed for the problem, including boundary conditions (BCs), is represented as a separate object. Each equation object will be discretized by appropriate numerical objects.

Note that if the tradititional algorithm had been used, this simple model would not have worked. Traditional forward communication would have required special tailoring for the application of the elliptic constraint. The traditional forward calling sequence is therefore less flexible. In contrast, the state machine approach could be reused without changes.

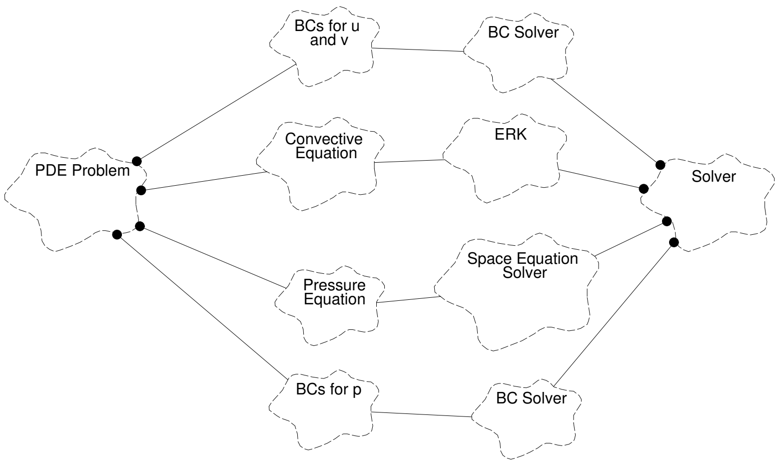


Figure 5: A simplified picture of the objects involved in the solution of the incompressible Navier–Stokes equations. The PDE Problem aggregate consists of the individual equation objects, which are discretized by proper counterparts in the Solver aggregate. (Notation according to Booch [4].)

6 Conclusions

In this report, we have investigated different strategies for implementing numerical time stepping schemes in the context of general PDE solving environments, such as our own research project Compose [1, 2].

We have studied four basic variations for implementing explicit Runge–Kutta time stepping methods. In particular, we present two novel strategies of implementing these, namely by extending the well-known techniques of forward and reverse communication with a state machine approach.

We argue that the state machine approach with forward communication is the most suitable alternative for implementing ERK methods in the context of general PDE solving environments. With respect to flexibility, traditional forward communication can not adapt to algorithmic variations, and therefore this option is impractical in a general setting. Among the remaining three alternatives, the state machine approach with forward communication is the most attractive, because of its simplicity.

Our numerical experiments show that none of the algorithms has a decisive lead with respect to performance. In our experiments, the important finding is that all communication strategies has comparable performance. This is supported by experiments on a model problem, by numerical experiments on a parallel computer, and by experiments in COMPOSE. Therefore, the choice of implementation may be driven by other objectives, such as flexibility and reusability.

We have pointed out the disadvantage with traditional forward communication in a general PDE solving environment such as our own research project COMPOSE. The incompressible Navier–Stokes equations constitute an example where the state machine approach yields a reusable ERK component, whereas traditional forward communication does not. Therefore we advocate the usage of state machines with forward communication, when ERK components shall be implemented in general PDE solving environments.

The conclusions drawn here do not apply for linear multistep methods, since they do not require intermediate stages for their time discretization. The conclusions may be generalized to diagonal implicit Runge–Kutta schemes, though.

Acknowledgements

I am grateful to Jarmo Rantakokko for providing me with a suitable parallel test program.

References

[1] K. Åhlander. An extendable PDE solver framework. Technical Report 197, Dept. of Scientific Computing, Uppsala University, Uppsala, Sweden, 1997.

[2] K. Åhlander. Rapid Development of PDE Solvers. Technical Report 212, Dept. of Scientific Computing, Uppsala University, Uppsala, Sweden, 1998.

[3] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne Natl. Lab., Argonne, IL, 1995.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Reading, Massachusetts, 2nd edition, 1994.

[5] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations. In Y. Ishikawa, R. Oldehoeft, J. Reynders, and M Tholburn, editors, *Scientific Computing in Object-oriented Parallel Environments*, pages 177–184, Berlin, 1997.

Springer-Verlag. See also Overture web page, `http://www.c3.lanl.gov/~henshaw/Overture/Overture.html`.

[6] A. M. Bruaset, E. J. Holm, and H. P. Langtangen. Increasing the efficiency and reliability of software development for systems of PDEs. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 247–268. Birkhäuser, 1997.

[7] W. H. Enright, J. H. Desmond, B. Owren, and P. W. Sharp. A survey of the explicit Runge–Kutta method. Technical Report TR 291/94, Department of Computer Science, University of Toronto, 1994. Revised April, 1995.

[8] H. W. Gerlach. Reverse communication and the ODE solver STRIDE. *Annals of Numerical Mathematics*, 1:399–410, 1994.

[9] E. Hairer, Nørsett. S.P., and G. Wanner. *Solving Ordinary Differential Equations I.* Springer-Verlag, 1993.

[10] W. D. Henshaw. Ogen: An overlapping grid generator for Overture. Technical Report LA-UR-96-3466, Los Alamos National Laboratory, Los Alamos, NM, 1996.

[11] G. Nelissen and P. Vankeirsbilck. Electrochemical modelling and software genericity. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 81–104. Birkhäuser, 1997.

[12] J. Rantakokko. *Data Partitioning Methods and Parallel Block-Oriented PDE Solvers.* PhD thesis, Uppsala University, Dept. of Scientific Computing, Uppsala, Sweden, 1998.

[13] M. Thuné, et al. Object-oriented construction of parallel PDE solvers. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 203–226. Birkhäuser, 1997.