Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Runge–Kutta methods with minimum storage implementations

David I. Ketcheson

King Abdullah University of Science and Technology, Box 4700, Thuwal 23955-6900, Saudi Arabia

A R T I C L E   I N F O

Article history:
Received 8 March 2009
Received in revised form 16 October 2009
Accepted 9 November 2009
Available online 29 November 2009

Keywords:
Runge–Kutta methods
Low-storage
Method of lines

A B S T R A C T

Solution of partial differential equations by the method of lines requires the integration of large numbers of ordinary differential equations (ODEs). In such computations, storage requirements are typically one of the main considerations, especially if a high order ODE solver is required. We investigate Runge–Kutta methods that require only two storage locations per ODE. Existing methods of this type require additional memory if an error estimate or the ability to restart a step is required. We present a new, more general class of methods that provide error estimates and/or the ability to restart a step while still employing the minimum possible number of memory registers. Examples of such methods are found to have good properties.

1. Introduction

In this work we are concerned with the numerical solution of the initial value problem (IVP)

$$u' = F(t,u) \quad u(0) = u_0, \tag{1}$$

where $u \in \Re^N$ and $F : \Re \times \Re^N \rightarrow \Re^N$. For brevity, we will typically omit the time-dependence and write $F(u)$. Here N represents the size of the system of ODEs (1); when N is large, as in the case of a semi-discretization of a PDE, the storage required to solve the system of equations (1) can be the limiting factor for practical computations.

In particular, we are interested in the approximate solution of (1) via the Runge–Kutta method [4,11]

$$y_i = u^n + \Delta t \sum_{j=1}^{i-1} a_{ij} F(t^n + c_j \Delta t, y_j) \quad 1 \leqslant i \leqslant m, \tag{2a}$$

$$u^{n+1} = u^n + \Delta t \sum_{j=1}^{m} b_j F(t^n + c_j \Delta t, y_j). \tag{2b}$$

Here u^n is an approximation to $u(t^n)$ and $\Delta t = t^{n+1} - t^n$. The intermediate values y_i are referred to as stages, and the Butcher coefficients a_{ij}, b_j, c_j define the method. A straightforward implementation of (2b) requires $m + 1$ memory registers of length N, where m is the number of stages and N is the number of ordinary differential equations. In early computers, only a very small amount of memory was available for storing both the program and intermediate results of the computation, which led Gill to devise a four-stage fourth-order Runge–Kutta method that could be implemented using only three memory registers [10]. The method relied on a particular algebraic relation between the coefficients, such that only certain combinations of previous stages (rather than all of the stages themselves) needed to be stored. This is the basic idea underlying all low-storage methods, including those of the present work. Blum later provided a three-register implementation of the classical Runge–Kutta method (with rational coefficients) [2]. Fyfe showed that all four-stage fourth-order methods are capable of three-register implementation [9]. Shampine devised a variety of techniques for reducing the storage requirements of meth-

E-mail addresses: david.ketcheson@kaust.edu.sa, ketch@amath.washington.edu

0021-9991/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2009.11.006

ods with many stages [16]. Williamson devised a two-register algorithm [21]; he showed that "all second-order, many third-order, and a few fourth-order" methods can be implemented in this fashion. One of his third-order methods is among the most popular low-storage methods; however, his fourth-order methods are not generally useful because they apply only to the special case in which $F(u)$ is bounded as $u \rightarrow \infty$.

On modern computers, storage space for programs is no longer a concern; however, when integrating very large numbers of ODEs, fast memory for temporary storage during a computation is often the limiting factor. This is typically the case in method of lines discretizations of PDEs, and modern efforts have focused on finding low-storage methods that are also optimized for stability and or accuracy relative to particular semi-discretizations of PDEs. Exploiting Williamson's technique, Carpenter and Kennedy [8] developed four-stage, third-order, two-register methods with embedded second-order methods for error control. They also derived five-stage fourth-order two-register methods [7]. In perhaps the most thorough work on low-storage methods, Kennedy et. al. [14] generalized a type of low-storage implementation originally due to van der Houwen [20]. They provide many methods of various orders, optimized for a variety of accuracy and stability properties. Further development of low-storage Runge–Kutta methods in the last decade has come from the computational aeroacoustics community [13,18,5,6,3,1,19].

All of the two-register methods in the literature use one of two algorithms (referred to below as 2N and 2R). These algorithms rely on particular assumptions regarding the evaluation and storage of F. Recently, in [15], a new type of low-storage algorithm was proposed, based on a different assumption on F. The aim of the present work is to present a general algorithm based on this assumption, which includes the 2N and 2R algorithms as special cases, but allows additional degrees of freedom in designing the methods.

It is often important, when solving an ODE numerically, to have an estimate of the local truncation error. Methods that provide such error estimates are known as embedded methods. Existing low-storage embedded methods always require an extra memory register to provide the error estimate. If a desired error tolerance is exceeded in a given step, it may be necessary to restart that step. In this case, another additional register is necessary for storing the previous step solution. In some applications where no error estimate is used, restarting may still be required based on some other condition (such as a CFL condition) that is checked after each step. In this case, again, existing low-storage methods require the use of an extra register.

In the present work we present improved low-storage methods that use the theoretical minimum number of registers in each of these situations, i.e. two registers if an error estimate or the ability to restart is required, and three registers if both are needed. In each case these methods use one register fewer than any known methods.

In Section 2, we review existing low-storage methods and explicitly define the assumptions required for their implementation. In Section 3, we observe that these methods have sparse Shu–Osher forms. Based on this observation, we introduce a new, more general class of low-storage methods. In Section 5, we explain how the low-storage methods can be implemented for integrating an important class of PDE semi-discretizations. In Section 6, we present new low-storage methods.

2. Two-register methods

Before proceeding, it is helpful to define precisely what is meant by the number of registers required by a method. Let N denote the number of ODEs to be integrated (typically the number of PDEs multiplied by the number of gridpoints). Then we say a method requires M registers if each step can be calculated using $MN + o(N)$ memory locations.

Let $\mathtt{S}_1, \mathtt{S}_2$ represent two N-word registers in memory. Then it is always assumed that we can make the assignments

$$\mathtt{S}_1 := \mathtt{F}(\mathtt{S}_2),$$

and

$$\mathtt{S}_1 := \mathtt{c}_1\mathtt{S}_1 + \mathtt{c}_2\mathtt{S}_2,$$

without using additional memory beyond these two registers. Here $\mathtt{a} := \mathtt{b}$ means 'the value of $\mathtt{b}$ is stored in $\mathtt{a}$'. Using these only these two types of assignments, it is straightforward to implement an m-stage method using $m+1$ registers.

Various Runge–Kutta algorithms have been proposed that require only two registers. Each requires some additional type of assignment, and takes one of the following two forms.

2.1. Williamson (2N) methods

Williamson methods [21] require 2 registers and take the following form:

Algorithm 1. Williamson (2N)

```
($y_1$)   $S_1 := u^n$
          for i = 2:m+1 do
              $S_2 := A_i S_2 + \Delta t F(S_1)$
($y_i$)       $S_1 := S_1 + B_i S_2$
          end
          $u^{n+1} = S_1$
```

with $A_2 = 0$. The methods proposed in [8,7,18,1,12] are also of Williamson type. Observe that an m-stage method has $2m - 1$ free parameters. The coefficients above are related to the Butcher coefficients as follows:

$$\begin{aligned}
B_i &= a_{i+1,i} \quad i < m \\
B_m &= b_m \\
A_i &= \frac{b_{i-1} - a_{i,i-1}}{b_i} \quad b_i \neq 0 \\
A_i &= \frac{a_{i+1,i-1} - c_i}{a_{i+1,i}} \quad b_i = 0.
\end{aligned}$$

In order to implement these methods with just two registers, the following assumption is required:

Assumption 1 (*Williamson*). Assignments of the form

$$\mathtt{S}_2 := \mathtt{S}_2 + \mathtt{F}(\mathtt{S}_1), \tag{4}$$

can be made with only $2N + o(N)$ memory.

Williamson notes that (converting some notation to the present)

No advantage can be gained by trying to generalize [Algorithm 1] by including a term [proportional to $\mathtt{S}_1$] in the expression for [$\mathtt{S}_2$]...this is because the additional equations determining the new parameters turn out to be linearly dependent on those for A_i and B_i.

Indeed, it appears that the algorithm above is the most general possible using only two registers and Assumption 1.

2.2. van der Houwen (2R) methods

A different low-storage algorithm was developed by van der Houwen [20] and Wray [14]. It is similar to, but more aggressive than the approach used by Gill [10]. The implementation is given as Algorithm 2.

Algorithm 2. van der Houwen (2R)

```
          $S_2 := u^n$
          for i = 1:m do
($y_i$)     $S_1 := S_2 + (a_{i,i-1} - b_{i-1})\Delta t S_1$
            $S_1 := F(S_1)$
            $S_2 := S_2 + b_i \Delta t S_1$
          end
          $u^{n+1} = S_2$.
```

The coefficients a_{ij}, b_j are the Butcher coefficients, and we define $a_{10} = b_0 = 0$. Again, an m-stage method has $2m - 1$ free parameters. The class of methods proposed in [5,6] is equivalent. In [14], an implementation is given that requires swapping the roles of the two registers at each stage. Here we have followed the implementation of [6] as it is less complicated in that the roles of the two registers need not be swapped at each stage.

Methods of van der Houwen type have also been proposed in [19]. The low-storage methods of [13] can be viewed as a subclass of van der Houwen methods with especially simple structure.

In order to implement these methods with just two registers, the following assumption is required:

Assumption 2 (*van der Houwen*). Assignments of the form

$$\mathtt{S}_1 := \mathtt{F}(\mathtt{S}_1),$$

may be made with only $N + o(N)$ memory.

Again, it seems that the above algorithm is the most general possible using only two registers and Assumption 2.

3. Low-storage methods have sparse Shu–Osher forms

The low-storage methods above can be better understood by considering the Shu–Osher form [17] for a Runge–Kutta method. This form allows a given method to be expressed in a variety of ways. It was first introduce in order to study strong stability preserving Runge–Kutta methods. Its usefulness in the present context lies in its ability to elucidate linear dependencies among the Runge–Kutta stages.

The Shu–Osher form of a Runge–Kutta method is

$$y_1 = u^n, \tag{5a}$$

$$y_i = \sum_{j=1}^{i-1} (\alpha_{ij} y_j + \beta_{ij} \Delta t F(y_j)) \quad 2 \leqslant i \leqslant m+1, \tag{5b}$$

$$u^{n+1} = y_{m+1}. \tag{5c}$$

For convenience we have shifted the indexing of α, β relative to the usual Shu–Osher form in order to make the stages y_i agree with the Butcher form. The Shu–Osher form for a given method is not unique. By writing (5c) as a homogeneous linear system, it follows that the method is invariant under the transformation (for any t and $i,j > 1$)

$$\alpha_{ij} \Rightarrow \alpha_{ij} - t, \tag{6a}$$

$$\alpha_{ik} \Rightarrow \alpha_{ik} + t\alpha_{jk} \quad k \neq j, \tag{6b}$$

$$\beta_{ik} \Rightarrow \beta_{ik} + t\beta_{jk}. \tag{6c}$$

It is convenient to define the matrices:

$$(\boldsymbol{\alpha})_{ij} = \begin{cases} 0 & i = 1, \\ \alpha_{ij} & i > 1, \end{cases} \tag{7}$$

$$(\boldsymbol{\beta})_{ij} = \begin{cases} 0 & i = 1, \\ \beta_{ij} & i > 1. \end{cases} \tag{8}$$

Defining further $\boldsymbol{\alpha}_0, \boldsymbol{\beta}_0$ to be the upper $m \times m$ parts of $\boldsymbol{\alpha}, \boldsymbol{\beta}$, and $\boldsymbol{\alpha}_1, \boldsymbol{\beta}_1$ to be the remaining last rows, the relation between the Butcher array and the Shu–Osher form is

$$A = (I - \boldsymbol{\alpha}_0)^{-1} \boldsymbol{\beta}_0 = \left(\sum_{i=0}^{m-1} \boldsymbol{\alpha}_0^i \right) \boldsymbol{\beta}_0, \tag{9a}$$

$$b^{\mathrm{T}} = \boldsymbol{\beta}_1 + \boldsymbol{\alpha}_1 A. \tag{9b}$$

It turns out that Williamson and van der Houwen methods possess a Shu–Osher form in which the matrices $\boldsymbol{\alpha}, \boldsymbol{\beta}$ are very sparse. This is not surprising, since low-storage algorithms rely on partial linear dependencies between the stages, and such dependencies can be exploited using the transformation (6c) to introduce zeros into these matrices.

3.1. 2N methods

By straightforward algebraic manipulation, Williamson methods can be written in Shu–Osher form with

$$y_i = \alpha_{i,i-2} y_{i-2} + (1 - \alpha_{i,i-2}) y_{i-1} + \beta_{i,i-1} \Delta t F(y_{i-1}) \quad 1 < i \leqslant m+1, \tag{10}$$

where $\alpha_{i,i-2} = -\frac{B_i A_i}{B_{i-1}}$ and $\beta_{i,i-1} = B_i$. Here and elsewhere, any coefficients with nonpositive indices are taken to be zero. Notice that $\boldsymbol{\alpha}$ is bidiagonal and $\boldsymbol{\beta}$ is diagonal.

3.2. 2R methods

Similarly, van der Houwen methods can be written in Shu–Osher form with

$$y_i = y_{i-1} + \beta_{i,i-2} \Delta t F(y_{i-2}) + \beta_{i,i-1} \Delta t F(y_{i-1}) 1 < i \leqslant m+1 \tag{11}$$

where $\beta_{i,i-2} = b_{i-1} - a_{i-1,i-2}$ and $\beta_{i,i-1} = a_{i,i-1}$. Notice that $\boldsymbol{\alpha}$ is diagonal and $\boldsymbol{\beta}$ is bidiagonal.

4. 2S methods

Based on the Shu–Osher forms presented above for 2N and 2R methods, it is natural to ask whether it is possible to implement a method with just two registers if $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ have other types of sparse structure. Perhaps the most obvious generalization is to allow both matrices to be bidiagonal, i.e.

$$y_i = \alpha_{i,i-2} y_{i-2} + (1 - \alpha_{i,i-2}) y_{i-1} + \beta_{i,i-2} \Delta t F(y_{i-2}) + \beta_{i,i-1} \Delta t F(y_{i-1}). \tag{12}$$

It turns out that this is possible, under the following assumption, which was introduced in [15]:

Assumption 3. Assignments of the form

$$\mathtt{S}_1 := \mathtt{S}_1 + \mathtt{F}(\mathtt{S}_1),$$

can be made with only $N + o(N)$ memory.

Examining the Shu–Osher form (12), it is clear that 2S methods may be implemented (under Assumption 3) using two registers if one is willing to evaluate $F(y_i)$ twice for each stage i. With a little care, however, this doubling of the number of function evaluations can be avoided. The resulting algorithm is given as Algorithm 3.

Algorithm 3. 2S

```
            $S_2 := 0$
($y_1$)     $S_1 := u^n$
            for i = 2:m+1 do
              $S_2 := S_2 + \delta_{i-1} S_1$
($y_i$)       $S_1 := \gamma_{i1} S_1 + \gamma_{i2} S_2 + \beta_{i,i-1} \Delta t F(S_1)$
            end
            $u^{n+1} = S_1$
```

The value of δ_m makes no essential difference (any change can be compensated by changing $\gamma_{m+1,1}, \gamma_{m+1,2}$), so we set it to zero. Consistency requires that $\delta_1 = 1, \gamma_{22} = 1$, and

$$\gamma_{i,1} = 1 - \gamma_{i,2} \sum_{j=1}^{i-1} \delta_j \quad 2 \leqslant i \leqslant m+1,$$

leaving $3s - 3$ free parameters – significantly more than for the 2N or 2R methods. We refer to these methods as 2S methods. Clearly this class includes the 2N and 2R methods as well as new methods.

While the Butcher coefficients are not needed for implementation, they are useful for analyzing the properties of the methods. They can be obtained from the low-storage coefficients as follows. The coefficients $\beta_{i,i-1}$ appearing in Algorithm 3 are Shu–Osher coefficients. In terms of the low-storage coefficients, the remaining nonzero Shu–Osher coefficients are

$$\beta_{i+1,i-1} = -\frac{\gamma_{i+1,2}}{\gamma_{i,2}} \beta_{i,i-1} \quad 2 \leqslant i \leqslant m,$$
$$\alpha_{i+1,i-1} = -\frac{\gamma_{i+1,2}}{\gamma_{i,2}} \gamma_{i,1} \quad 2 \leqslant i \leqslant m,$$
$$\alpha_{i+1,i} = 1 - \alpha_{i+1,i-1} \quad 2 \leqslant i \leqslant m.$$

The Butcher coefficients are obtained by substituting the above values into (9b).

If $\gamma_{i2} = 0$ for some i, the low-storage method cannot be written in the bidiagonal Shu–Osher form; however, it will still possess a sparse (in fact, even more sparse) Shu–Osher form, and can be implemented using two registers in a slightly different way. We do not investigate such methods in detail, since they have a smaller number of free parameters than the general case. However, note that some of the methods of [15] are of this type.

4.1. 2S* methods

It is common to check some accuracy or stability condition after each step, and to reject the step if the condition is violated. In this case, the solution from the last timestep, u^n, must be retained during the computation of u^{n+1}. For 2R/2N/2S methods, this will require an additional register. On the other hand, in [15], methods were proposed that can be implemented using only two registers, with one register retaining the previous solution. We refer to these as 2S* methods. These methods have Shu–Osher form

$$y_i = \alpha_{i,1} u^n + \alpha_{i,i-1} y_{i-1} + \beta_{i,i-1} \Delta t F(y_{i-1}). \tag{14}$$

Here we give a general algorithm for such methods (Algorithm 4), which is straightforward given the sparse Shu–Osher form. It is equivalent to the usual 2S algorithm with $\gamma_{i2} = \alpha_{i,1}$ and $\delta_i = 0$ except $\delta_1 = 1$. Remarkably, these methods have as many free parameters (2m-1) as 2N/2R methods.

Algorithm 4. 2S*

```
($y_1$)     $S_1 := u^n \quad S_2 := u^n$
            for i = 2:m+1 do
($y_i$)         $S_1 := (1 - \alpha_{i,1})S_1 + \alpha_{i,1} S_2 + \beta_{i,i-1} \Delta t F(S_1)$
            end
            $u^{n+1} = S_1$
```

4.2. 2S embedded pairs

It is often desirable to compute an estimate of the local error at each step. The most common way of doing this is to use an embedded method, i.e. a second Runge–Kutta method that shares the same matrix A (hence the same stages) but a different vector of weights $\hat{b}$ in place of b. The methods are designed to have different orders, so that their difference gives an estimate of the error in the lower order result. As it is common to advance the higher order solution ('local extrapolation'), we refer to the higher order method as the principal method, and the lower order method as the embedded method. Typically the embedded method has order one less than the principal method.

In [14], many 2R embedded pairs are given; however, a third storage register is required for the error estimate. The 2S algorithm can be modified to include an embedded method while still using only two storage registers. The implementation is given as Algorithm 5.

Algorithm 5. 2S embedded
```
                 $S_2 := 0$
($y_1$)          $S_1 := u^n$
                 for i = 2:m+1 do
                   $S_2 := S_2 + \delta_{i-1} S_1$
($y_i$)            $S_1 := \gamma_{i1} S_1 + \gamma_{i2} S_2 + \beta_{i,i-1} \Delta t F(S_1)$
                 end
                 $u^{n+1} = S_1$
($\hat{u}^{n+1}$)    $S_2 := \frac{1}{\sum_{i=2}^{m+1} \delta_i} (S_2 + \delta_{m+1} S_1)$
```

Here $\hat{u}^{n+1}$ is the embedded solution. Note that there are two additional free parameters, δ_m, δ_{m+1}, effectively determining the weights $\hat{b}$. Since the conditions for first and second-order are (for fixed abscissas c) a pair of linear equations for the weights, it would appear that if the embedded method is at least second-order, then necessarily $\hat{b} = b$. However, by including additional stages, the added degrees of freedom can be used to achieve independence of $b, \hat{b}$.

The relation between the coefficients in Algorithm 5 and the Shu–Osher coefficients for the principal method is again given by (17), and the Butcher coefficients can then be obtained using (9b). The embedded method has the same Butcher arrays A, c, with the weights $\hat{b}$ given by

$$\hat{b}_j = \frac{1}{\sum_k \delta_k} \left(\delta_{m+1} b_j + \sum_i \delta_i a_{ij} \right), \tag{15}$$

where b_j, a_{ij} are the Butcher coefficients of the principal method.

4.3. 3S* methods

Our experience indicates that the class of 2S* methods above typically have relatively unfavorable error constants (though they are not so large as to be unusable for practical applications). Hence we are led to consider methods with Shu–Osher form

$$y_i = \alpha_{i,1} u^n + \alpha_{i,i-1} y_{i-1} + \alpha_{i,i-2} y_{i-2} + \beta_{i,i-2} \Delta t F(y_{i-2}) + \beta_{i,i-1} \Delta t F(y_{i-1}). \tag{16}$$

These methods can be implemented using three registers, while retaining the previous solution. Hence we refer to them as 3S* methods. Because they allow more free parameters than 2S* methods, 3S* methods can be found with much smaller error constants. Furthermore, it is possible to design embedded pairs within this framework. The corresponding algorithm is given as Algorithm 6. Note that these are the first methods to provide both error control and the ability to restart a step with only three memory registers. No further improvement is possible, since the new solution, the previous solution, and an error estimate must be available simultaneously.

Algorithm 6. 3S* embedded
```
                 $S_3 := u^n$
($y_1$)          $S_1 := u^n$
                 for i = 2:m+1 do
                     $S_2 := S_2 + \delta_{i-1} S_1$
($y_i$)              $S_1 := \gamma_{i1} S_1 + \gamma_{i2} S_2 + \gamma_{i3} S_3 + \beta_{i,i-1} \Delta t F(S_1)$
                 end
($\hat{u}^{n+1}$)        $S_2 := \frac{1}{\sum_{j=1}^{m+2} \delta_j} (S_2 + \delta_{m+1} S_1 + \delta_{m+2 S_3})$
                 $u^{n+1} = S_1$
```

Note that including terms in $\mathtt{S}_{tt2}$ proportional to $\mathtt{S}_3$ is superfluous. Consistency requires $\delta_1 = 1$ and we take $\gamma_{22} = 1$, $\gamma_{21} = \gamma_{23} = \gamma_{33} = \gamma_{43} = 0$ to eliminate additional spurious degrees of freedom. Thus the primary method has $4m - 6$ free parameters, with 3 more available for the embedded method. Once again, to avoid having $\hat{b} = b$, additional stages are necessary.

Again, the coefficients $\beta_{i,i-1}$ in Algorithm 6 are just the corresponding Shu–Osher coefficients. The remaining Shu–Osher coefficients are

$$\begin{aligned}
\beta_{i+1,i-1} &= -\frac{\gamma_{i+1,2}}{\gamma_{i,2}}\beta_{i,i-1} \quad 2 \leqslant i \leqslant m,\\
\alpha_{i+1,1} &= \gamma_{i+1,3} - \frac{\gamma_{i+1,2}}{\gamma_{i,2}}\gamma_{i,3} \quad 2 \leqslant i \leqslant m,\\
\alpha_{i+1,i-1} &= -\frac{\gamma_{i+1,2}}{\gamma_{i,2}}\gamma_{i,1} \quad 2 \leqslant i \leqslant m,\\
\alpha_{i+1,i} &= 1 - \alpha_{i+1,i-1} - \alpha_{i+1,1} \quad 2 \leqslant i \leqslant m.
\end{aligned}$$

The Butcher array for the principal method can then be obtained using (9b). The embedded method is again identical except for the weights, which are given by (15).

5. Feasibility of low-storage assumptions

In this section we discuss the feasibility of the various low-storage assumptions. We will assume the method is applied to integrate a semi-discretization of a PDE on a structured grid where the stencil is local; in 1D this means that the Jacobian is sparse with narrow bandwidth; i.e., the formula for updating a given cell depends on a local stencil whose size is independent of the overall grid size. This is typical of many finite difference, finite volume, and discontinuous Galerkin methods. For semi-discretizations with dense jacobian, it appears that an additional 'working space' memory register will always be necessary. In one dimension, we write the semi-discretization as:

$$\frac{\partial u_{ij}}{\partial t} = F(u_{i-r}, \ldots, u_{i+r}), \tag{17}$$

for some (small) fixed integer r.

5.1. 2N methods

Recall that 2N methods require Assumption 1, which involves assignments of the form

$$\mathtt{S}_1 := \mathtt{S}_1 + \mathtt{F}(\mathtt{S}_2). \tag{18}$$

For systems of the form (17), implementation of 2N methods is completely straightforward, since the register from which F is being calculated is different from the register to which it is being written. The algorithm simply marches along the grid, calculating F at each point.

5.2. 2R methods

Recall that 2R methods require Assumption 2, which involves assignments of the form

$$\mathtt{S}_1 := \mathtt{F}(\mathtt{S}_1). \tag{19}$$

A naive implementation like that prescribed for 2N methods above will overwrite solution values that are needed for subsequent computations of F. It is thus necessary to maintain a small buffer with old solution values that are still needed. In one dimension, the buffer need only be the size of the computational stencil (i.e., 2r+1). The algorithm looks roughly like this (letting S denote the memory register and w the buffer)

$$\begin{aligned}
&\mathtt{w}[tt1 : 2r] := \mathtt{w}[2 : 2\mathtt{r} + 1],\\
&\mathtt{w}[2\mathtt{r} + 1] := \mathtt{S}[\mathtt{i} + \mathtt{r}],\\
&\mathtt{S}[\mathtt{i}] := \mathtt{F}(\mathtt{w}).
\end{aligned}$$

In higher dimensions a similar strategy can be used, depending on whether the stencil is one-dimensional or multi-dimensional. If it is 1D, then one can simply apply the algorithm above along each slice. If it is d-dimensional then a buffer containing $2r + 1$ slices of dimension $d - 1$ is required. In either case, the buffer size is much smaller than a full register.

5.3. 2S methods

For the type of spatial discretizations under consideration here, implementation of 2S methods is no more difficult than implementation of 2R methods. The algorithm in 1D is identical except that one assigns

Table 1
Properties of low-storage methods.

															Registers Required			
Method	S_I	S_R	$A^{(p+1)}$	$A^{(p+2)}$	$\widehat{S}_I$	$\widehat{S}_R$	$\widehat{A}^{(\hat{p}+1)}$	$B^{(\hat{p}+1)}$	$C^{(\hat{p}+1)}$	D	$E^{(\hat{p}+1)}$	$r_{\mathcal{F}_2}$	$r_{\mathcal{L}_\infty}$	$r_{\mathcal{F}_\infty}$	N	R	EE	R,EE
Classical RK4	0.707	0.696	1.45e−02	1.60e−02								1.000	1.000	0.000	4	5	–	–
RK4()4[2S]	0.707	0.696	2.81e−02	3.02e−02								0.000	1.000	0.000	2	3	–	–
RK4()6[2S]	0.597	1.600	4.17e−03	5.30e−03								0.000	2.087	0.000	2	3	–	–
RK4()5[2S*]	0.619	0.671	1.49e−02	1.88e−02								0.000	1.301	0.000	2	2	–	–
RK4(3)5[2R+]C	0.665	0.963	5.12e−03	7.45e−03	0.634	0.900	8.83e−03	1.409	0.964	0.845	0.580	0.000	1.717	0.000	2	3	2	3
RK4(3)6[2S]	0.733	0.586	2.58e−02	3.64e−02	0.327	0.518	3.87e−02	0.643	1.193	1.961	0.667	0.000	1.204	0.000	2	3	2	3
RK4(3)5[3S*]	0.668	0.930	5.52e−03	7.97e−03	0.399	0.833	6.38e−02	1.198	1.159	3.624	0.087	0.000	1.664	0.000	3	3	3	3

```
    s[i] := s[i] + F(w),
```

at each point. The extension to multiple dimensions follows the same pattern.

6. Improved low-storage methods

In this section we present some new low-storage methods of the types developed in the previous section. This is only intended to demonstrate what is possible; a thorough investigation of 2S methods optimized for various properties, like that done in [14] for 2R methods, is left for future work.

Similarly to [14], we refer to a method as RK-$p(\hat{p})m[X]$, where m is the number of stages, p is the order of accuracy, $\hat{p}$ is the order of accuracy of the embedded method (if any), and X indicates the type of method (2R, 2S, 2S*, etc.).

By writing a three-stage Runge–Kutta method in Shu–Osher form (5c) and using transformations of the form (6c) to write the method in the form (12), it is seen that any three-stage method may be implemented in 2S form except in the special case that $\beta_{31} = \alpha_{31}\beta_{21}$. In this case the method may be implemented with two registers using a slight modification of the 2S algorithm. Hence all three-stage Runge–Kutta methods can be implemented using two registers.

Throughout the development of low-storage Runge–Kutta methods, fourth-order methods have been of particular interest [10,2,9,21]. In Table 1 we present several examples of minimum storage fourth-order methods. We have also found 2S (and 2S*, embedded, etc.) methods of fifth and sixth orders. As far as we know, our sixth order methods are the first two-register methods of sixth order. A description of these methods will be included in a forthcoming work.

It is known that no generally applicable four-stage fourth-order two-register methods of 2N or 2R type exist [21,14]. This is not surprising, since four-stage methods in those classes have seven free parameters, whereas there are 8 conditions for fourth-order accuracy. Four-stage 2S methods, on the other hand, have 9 free parameters, and fourth-order methods of this kind exist. An example of such a method is given in Table 1 as RK4()4[2S]. Its coefficients are provided in Table 2.

By allowing additional stages, methods with improved accuracy or stability are possible. These methods can have very good properties compared to 2R or 2N methods because of the additional degrees of freedom available. As an example, we include in Table 3 a six-stage, fourth-order method RK4()6[2S] with improved real-axis stability. In Table 4 we present a 2S* method of fourth-order, using five stages. In Table 5 we present a 4(3)5 2S pair, and in Table 6 a 4(3)5 3S* pair.

Table 1 summarizes the accuracy and stability properties of these methods. Most of the properties in this table follow the notation of [14]. The quantities S_I, S_R are the size of the largest interval included in the region of absolute stability along the imaginary and real axes, respectively, scaled (divided) by the number of stages of the method. The quantities $\widehat{S}_I, \widehat{S}_R$ are the corresponding values for the embedded scheme. The quantity A^{p+1} is the L_2 principal error norm (i.e., the norm of the vector of leading-order truncation error coefficients). For definitions of the remaining properties, the reader is referred to [14]. The last four columns of the table indicate the number of registers required to implement the method, under the appropriate low-storage assumption for each method. The column labeled "R" indicates the number required if the ability to restart a step is needed; column "EE" indicates the number required in case the embedded error estimator is used; column "R, EE" indicates the number needed if both restarts and error estimation are necessary; column "N" indicates the number required if neither restarts or error estimates are needed.

The classical fourth-order RK method and a recommended 2R method from [14] are included for comparison. The new methods all have reasonably good properties. None of the methods is contractive, however; for contractive low-storage methods see [15].

7. Conclusions

We have proposed a new class of low-storage Runge–Kutta methods and given examples of high order methods in this class requiring fewer stages than existing low-storage methods. The methods include embedded pairs using only two memory registers, as well as embedded pairs that retain the previous solution value and use only three memory registers. Such methods were not previously available. A thorough investigation of 2S methods optimized for various properties, like that done in [14] for 2R methods, would be of great utility.

Acknowledgment

The author thanks Randy LeVeque for the suggestion to consider embedded pairs. This work was funded by a US Dept. of Energy Computational Science Graduate Fellowship.

Appendix A. Coefficients of low-storage methods

See Tables 2–6.

Table 2
Coefficients for RK4()4[2S].

i	γ_{i1}	γ_{i2}	$\beta_{i,i-1}$	δ_i
1	0.000000000000000	0.000000000000000	0.000000000000000	1.000000000000000
2	0.000000000000000	1.000000000000000	1.193743905974738	0.217683334308543
3	0.121098479554482	0.721781678111411	0.099279895495783	1.065841341361089
4	−3.843833699660025	2.121209265338722	1.131678018054042	0.000000000000000
5	0.546370891121863	0.198653035682705	0.310665766509336	

Table 3
Coefficients for RK4()6[2S].

i	γ_{i1}	γ_{i2}	$\beta_{i,i-1}$	δ_i
1	0.000000000000000	0.000000000000000	0.000000000000000	1.000000000000000
2	0.000000000000000	1.000000000000000	0.238829375897678	0.564427596596565
3	0.344088773828091	0.419265952351424	0.467431873315953	1.906950911013704
4	−0.655389499112535	0.476868049820393	0.215210792473781	0.617263698427868
5	0.698092532461612	0.073840520232494	0.205665392762124	0.534245263673355
6	−0.463842390383811	0.316651097387661	0.803800094404076	0.000000000000000
8	0.730367815757090	0.058325491591457	0.076403799554118	

Table 4
Coefficients for RK4()5[2S*].

i	γ_{i1}	γ_{i2}	$\beta_{i,i-1}$
1	0.000000000000000	0.000000000000000	0.000000000000000
2	0.000000000000000	1.000000000000000	0.357534921136978
3	−3.666545952121251	4.666545952121251	2.364680399061355
4	0.035802535958088	0.964197464041912	0.016239790859612
5	4.398279365655791	−3.398279365655790	0.498173799587251
6	0.770411587328417	0.229588412671583	0.433334235669763

Table 5
Coefficients for RK4(3)6[2S].

i	γ_{i1}	γ_{i2}	$\beta_{i,i-1}$	δ_i
1	0.000000000000000	0.000000000000000	0.000000000000000	1.000000000000000
2	0.000000000000000	1.000000000000000	0.653858677151052	−1.662080444041546
3	1.587969352283926	0.888063312510453	0.258675602947738	1.024831293149243
4	1.345849277346560	−0.953407216543495	0.802263873737920	1.000354140638651
5	−0.088819115511932	0.798778614781935	0.104618887237994	0.093878239568257
6	0.206532710491623	0.544596034836750	0.199273700611894	1.695359582053809
7	−3.422331114067989	1.402871254395165	0.318145532666168	0.392860285418747

Table 6
Coefficients for RK4(3)5[3S*].

i	γ_{i1}	γ_{i2}	γ_{i3}	$\beta_{i,i-1}$	δ_i
1	0.000000000000000	0.000000000000000	0.000000000000000	0.000000000000000	1.000000000000000
2	0.000000000000000	1.000000000000000	0.000000000000000	0.075152045700771	0.081252332929194
3	−0.497531095840104	1.384996869124138	0.000000000000000	0.211361016946069	−1.083849060586449
4	1.010070514199942	3.878155713328178	0.000000000000000	1.100713347634329	−1.096110881845602
5	−3.196559004608766	−2.324512951813145	1.642598936063715	0.728537814675568	2.859440022030827
6	1.717835630267259	−0.514633322274467	0.188295940828347	0.393172889823198	−0.655568367959557
7					−0.194421504490852

References

[1] J. Berland, C. Bogey, C. Bailly, Low-dissipation and low-dispersion fourth-order Runge–Kutta algorithm, Computers and Fluids 35 (2006) 1459–1463.
[2] E. Blum, A modification of the Runge–Kutta fourth-order method, Mathematics of Computation 16 (1962) 176–187.

[3] C. Bogey, C. Bailly, A family of low dispersive and low dissipative explicit schemes for flow and noise computations, Journal of Computational Physics 194 (2004) 194–214.
[4] J. Butcher, Numerical Methods for Ordinary Differential Equations, Wiley, 2003.
[5] M. Calvo, J. Franco, L. Rández, Minimum storage Runge–Kutta schemes for computational acoustics, Computers and Mathematics with Applications 45 (2003) 535–545.
[6] M. Calvo, J. Franco, L. Rández, A new minimum storage Runge–Kutta scheme for computational acoustics, Journal of Computational Physics 201 (2004) 1–12.
[7] M.H. Carpenter, C.A. Kennedy, Fourth-order 2N-storage Runge–Kutta schemes, Tech. Report TM 109112, NASA Langley Research Center, June 1994.
[8] M.H. Carpenter, C.A. Kennedy, Third-order 2N-storage Runge–Kutta schemes with error control, Tech. Report, NASA, 1994.
[9] D.J. Fyfe, Economical evaluation of Runge–Kutta formulae, Mathematics of Computation 20 (1966) 392–398.
[10] S. Gill, A process for the step-by-step integration of differential equations in an automatic digital computing machine, Proceedings of the Cambridge Philosophical Society 47 (1950) 96–108.
[11] E. Hairer, S. Norsett, G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, Springer Series in Computational Mathematics, Springer, Berlin, 1993.
[12] R. Hixon, V. Allampalli, M. Nallasamy, S. Sawyer, High-accuracy large-step explicit Runge–Kutta (HALE-RK) schemes for computational aeroacoustics, AIAA paper 2006-797, AIAA, 2006.
[13] F. Hu, M. Hussaini, J. Manthey, Low-dissipation and low-dispersion Runge–Kutta schemes for computational acoustics, Journal of Computational Physics 124 (1996) 177–191.
[14] C.A. Kennedy, M.H. Carpenter, R.M. Lewis, Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations, Applied Numerical Mathematics 35 (2000) 177–219.
[15] D.I. Ketcheson, Highly efficient strong stability preserving Runge–Kutta methods with low-storage implementations, SIAM Journal on Scientific Computing 30 (2008) 2113–2136.
[16] L. Shampine, Storage reduction for Runge–Kutta codes, ACM Transactions on Mathematical Software 5 (1979) 245–250.
[17] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock-capturing schemes, Journal of Computational Physics 77 (1988) 439–471.
[18] D. Stanescu, W. Habashi, 2N-storage low dissipation and dispersion Runge–Kutta schemes for computational acoustics, Journal of Computational Physics 143 (1998) 674–681.
[19] K. Tselios, T. Simos, Optimized Runge–Kutta methods with minimal dispersion and dissipation for problems arising from computational acoustics, Physics Letters A 363 (2007) 38–47.
[20] P. van der Houwen, Explicit Runge–Kutta formulas with increased stability boundaries, Numerische Mathematik 20 (1972) 149–164.
[21] J.H. Williamson, Low-storage Runge–Kutta schemes, Journal of Computational Physics 35 (1980) 48–56.