Report no. 00/10

On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equations

M. B. Giles

This paper considers the solution of steady adjoint equations using a class of iterative methods which includes preconditioned Runge-Kutta time-marching with multigrid. It is shown that, if formulated correctly, equal numbers of iterations of the direct and adjoint iterative solvers will result in the same value for the linear functional being sought. The precise details of the adjoint iteration are formulated for the case of Runge-Kutta time-marching with partial updates, which is commonly used in CFD computations. The theory is supported by numerical results from a MATLAB program for two model problems, and from programs for the solution of the linear and adjoint 3D Navier-Stokes equations.

(This report is an expanded version of a paper presented at the AD2000 Conference in Nice, France on June 19-23, 2000.)

Key words and phrases: Adjoint, Runge-Kutta, multigrid, design

This research was supported by EPSRC research grant GR/L95700.

Oxford University Computing Laboratory
Numerical Analysis Group
Wolfson Building
Parks Road
Oxford, England OX1 3QD
http: //www.comlab.ox.ac.uk
email: giles@comlab.oxford.ac.uk

June, 2000

1 Introduction

In CFD analysis, it is common to use Runge-Kutta time-marching with local timesteps and/or local preconditioning to solve the steady flow equations [11]. The use of multigrid, obtaining fine grid corrections by transferring residuals into a coarser grid and solving the coarse grid equations to obtain a correection which is interpolated onto the fine grid, provides even greater iterative convergence rates and is used extensively in CFD [10]. It is natural then to use similar techniques to solve the steady adjoint flow equations which arise in the context of optimal design and error analysis [7].

In formulating the adjoint equations, if one follows the discrete adjoint approach in which the adjoint operator is the transpose of the corresponding linear operator [1, 5] then the definition and construction of the spatial part of the operator is clear [7]. The implementation of the adjoint code can be aided significantly by the use of Automatic Differentation software (such as ADJIFOR [2] or Odyssee [6]) which will generate adjoint code for the spatial discretisation, given the nonlinear code as input.

However, it is not so clear how one should treat the iterative solution procedure if one wants a procedure which is truly adjoint, in the sense that 1000 iterations of the adjoint solver will give a linear functional which is equal to that given by 1000 iterations of the linearised direct solver. Although not a necessity, since it is only the steady adjoint solution which is needed, it is nevertheless very desirable because it means that the asymptotic rate of convergence of the adjoint solver must be exactly the same as the linear direct solver, and it is this guaranteed success of the discrete adjoint approach which is one of its strengths. It is also very helpful as a check on the correct implementation of the adjoint solver. By contrast, if one follows the so-called 'continuous' adjoint approach in which one discretises the adjoint p.d.e. [9], then it is often very difficult to fully validate the programming implementation given the lack of test cases with analytic adjoint solutions.

The purpose of this paper therefore is to investigate the construction of a proper adjoint treatment of Runge-Kutta time-marching to obtain a steady-state solution. The theory is first derived for equations which are continuous in time, and then for a class of methods which includes Runge-Kutta time-marching. The analysis allows for the possibility of partial updating of some terms, as commonly used by Jameson [8] and others in not updating the viscous fluxes at each stage, as well as the use of matrix preconditioning, as in block-Jacobi [10] or low Mach number preconditioning [11]. The analysis also covers the use of multigrid, showing that the restriction operator for the adjoint solver must be the transpose of the prolongation operator for the linear solver, and *vice versa*.

The paper concludes with a simple Matlab program which illustrates the programming of the direct and adjoint methods and produces results which support the conclusion that the adjoint Runge-Kutta time-marching is a true adjoint in

the sense defined above. The theoretical results are also supported by numerical tests with codes which approximate the linear and adjoint Navier-Stokes equations, using both block-Jacobi preconditioning and multigrid.

The analysis is related to the work of Christianson [3, 4] on fixed point iteration for solving the direct and adjoint equations. However, the present work differs from that of Christianson in restricting attention to linear iterative methods (excluding other methods such as conjugate gradient algorithms) and problems in which the adjoint variables have the same dimension as the primal variables. As a consequence, it is possible to formulate the adjoint iteration using working variables which converge to the steady adjoint variables. This is not possible in Christianson's more general approach. The present work also differs from Christianson's in considering in detail the application of the general theory to preconditioned Runge-Kutta time-marching and multigrid.

2 Continuous equations

In the steady problem, the functional to be evaluated is the inner product

$$I = (g, u),$$

where u is the solution of the linear equations

$$L\,u = f.$$

In steady design problems, f, g, u are all real vectors and L is a real matrix. However, in this paper we want to also allow for the case of complex vectors and matrices, as arises in the adjoint harmonic unsteady equations, and therefore the inner product notation denotes

$$(g, u) \equiv g^H u.$$

where g^H is the Hermitian, the complex conjugate transpose, of g.

The functional can be re-written as

$$I \;=\; (g, u) - (v, L\,u - f) \;=\; (v, f) - (L^H v - g, u) \;=\; (v, f),$$

where v is the solution of the adjoint equations

$$L^H v = g.$$

In the corresponding unsteady problem with the same steady vectors f, g, the unsteady $u(t)$ is given by the differential equation

$$\frac{du}{dt} = P\,(f - L\,u),$$

for some constant matrix P, subject to the initial conditions $u(0) = 0$, and the functional is the inner product

$$I = (g, u(T)),$$

at the final time T which is chosen to be sufficiently large that $\frac{du}{dt}$ is very small and therefore $u(T)$ is very close to being the solution of the steady equations.

The unsteady adjoint problem is given by

$$\begin{aligned} I &= (g, u(T)) - \int_0^T (w, \frac{du}{dt} + P(L\,u - f))\; dt \\ &= (g, u(T)) - \int_0^T (-\frac{dw}{dt} + L^H P^H w, u) - (P^H w, f)\; dt - (w(T), u(T)) \\ &= \int_0^T (P^H w, f)\; dt, \end{aligned}$$

where w is the solution of the differential equation

$$\frac{dw}{dt} = L^H P^H w,$$

which is solved backwards in time subject to the final condition $w(T) = g$.

To obtain the link with the steady adjoint equation, we define

$$v(t) = \int_t^T P^H w\, dt$$

so that the functional is

$$I = (v(0), f)$$

and $v(t)$ satisfies the differential equation

$$\begin{aligned} -\frac{dv}{dt} &= P^H w(t) \\ &= P^H \left(g - \int_t^T \frac{dw}{dt} dt \right) \\ &= P^H \left(g - \int_t^T L^H P^H w\, dt \right) \\ &= P^H \left(g - L^H v \right), \end{aligned}$$

subject to the final condition $v(T) = 0$. In this form, $v(t)$ is seen to correspond to an unsteady evolution towards the solution of the steady adjoint equation, and if T is very large then $v(0)$ will be very close to the steady adjoint solution.

3 Discrete equations

The discrete unsteady equations using the general class of one-step methods (which includes the generalised Runge-Kutta time-marching methods to be discussed in the next section) can be expressed as

$$u^{n+1} = u^n + R\,(f - L\,u^n),$$

where R is a matrix which depends on the details of the one-step method, including the timestep and whether or not any preconditioning is used. Writing the iterative equations in this form emphasises the point that the solution of the steady-state equations is also a steady solution of the unsteady discrete equations. It will be assumed that the iterative procedure is stable, and hence u^n converges exponentially towards the steady-state solution from the initial condition $u^0 = 0$.

The functional is evaluated using the final value u^N, which gives

$$I = (g, u^N).$$

Proceeding as before to find the discrete adjoint formulation yields

$$\begin{aligned} I &= (g, u^N) - \sum_{n=0}^{N-1} (w^{n+1}, u^{n+1} - u^n - R\,(f - L\,u^n)) \\ &= (g, u^N) - (w^N, u^N) - \sum_{n=0}^{N-1} \left\{ (-(w^{n+1} - w^n), u^n) + (L^H R^H w^{n+1}, u^n) - (R^H w^{n+1}, f) \right\} \\ &= (g - w^N, u^N) + \sum_{n=0}^{N-1} \left\{ (w^{n+1} - w^n - L^H R^H w^{n+1}, u^n) + (R^H w^{n+1}, f) \right\}, \end{aligned}$$

in which we have used the following identity which is the discrete equivalent of integration by parts,

$$\sum_{n=0}^{N-1} a^{n+1}\,(b^{n+1} - b^n) = a^N b^N - a^0 b^0 - \sum_{n=0}^{N-1} (a^{n+1} - a^n)\,b^n.$$

Consequently, if w satisfies the difference equation

$$w^n = w^{n+1} - L^H R^H w^{n+1},$$

subject to the final condition $w^N = g$, then the functional is

$$I = \sum_{n=0}^{N-1} (R^H w^{n+1}, f).$$

The above description of the discrete adjoint problem corresponds to what would be generated by Automatic Differentiation, using Odysee or ADJIFOR,

but as with the continuous equations it is preferable to cast the problem as time-marching towards the solution of the steady discrete adjoint equations. To do this we define the variable v^n as

$$v^n = \sum_{m=n}^{N-1} R^H w^{m+1}, \quad n < N$$

with $v^N = 0$, so that the functional can be expressed as

$$I = (v^0, f).$$

Finally, the difference equation for v^n comes from

$$\begin{aligned} v^n - v^{n+1} &= R^H w^{n+1} \\ &= R^H \left(g - \sum_{m=n+1}^{N-1} (w^{m+1} - w^m) \right) \\ &= R^H \left(g - \sum_{m=n+1}^{N-1} L^H R^H w^{m+1} \right) \\ &= R^H \left(g - L^H v^{n+1} \right) \end{aligned}$$

showing that v^n evolves towards the solution of the steady adjoint equations, with exactly the same rate of exponential convergence as the linear direct solution.

4 General Runge-Kutta schemes

In this section we consider a quite general class of Runge-Kutta methods which is used extensively in CFD, and includes both preconditioning and partial updates for viscous and smoothing fluxes. The aim is to first cast the methods into the form used above, finding an expression for the operator R, and hence determine R^H for the adjoint iterative scheme.

Splitting the linear operator L into two parts

$$L\,u \equiv C\,u + D\,u,$$

where Cu represents the inviscid flux terms and Du represents the viscous and smoothing fluxes, a preconditioned version of the partial-update M-stage Runge-

Kutta scheme used by Jameson [8] and others can be expressed as

$$\begin{array}{rcl} d^{(0)} &=& 0 \\ u^{(0)} &=& u^n \\ d^{(m)} &=& \beta_m D\, u^{(m-1)} + (1-\beta_m)\, d^{(m-1)} \\ u^{(m)} &=& u^{(0)} + \alpha_m P \left(f - C\, u^{(m-1)} - d^{(m)} \right) \\ u^{n+1} &=& u^{(M)} \end{array} \quad \left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \quad m = 1, 2, \ldots, M \tag{4.1}$$

where $\beta_1 = \alpha_{_M} = 1$, and P is a preconditioning matrix which in the simplest case is just the identity matrix scaled by a local timestep.

Defining r to be the residual at the beginning of the step,

$$r = f - L\, u^n,$$

and defining the perturbation quantities

$$\tilde{d}^{(m)} \equiv d^{(m)} - Du^n, \qquad \tilde{u}^{(m)} \equiv u^{(m)} - u^n,$$

the algorithm can be re-expressed as

$$\begin{array}{rcl} \tilde{d}^{(0)} &=& -Du^n \\ \tilde{u}^{(0)} &=& 0 \\ \tilde{d}^{(m)} &=& \beta_m D\, \tilde{u}^{(m-1)} + (1-\beta_m)\, \tilde{d}^{(m-1)} \\ \tilde{u}^{(m)} &=& \alpha_m P \left(r - C\, \tilde{u}^{(m-1)} - \tilde{d}^{(m)} \right) \\ u^{n+1} &=& u^n + \tilde{u}^{(M)} \end{array} \quad \left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \quad m = 1, 2, \ldots, M \tag{4.2}$$

The advantage of this form is that it makes it clear that the algorithm fits into the standard form

$$u^{n+1} = u^n + R\,(f - L\, u^n),$$

analysed earlier, with the matrix R defined implicitly by the system of simultaneous equations

$$B \begin{pmatrix} \tilde{u}^{(1)} \\ \tilde{d}^{(2)} \\ \tilde{u}^{(2)} \\ . \\ . \\ \tilde{d}^{(M)} \\ \tilde{u}^{(M)} \end{pmatrix} = \begin{pmatrix} \alpha_1 P \\ 0 \\ \alpha_2 P \\ . \\ . \\ 0 \\ \alpha_M P \end{pmatrix} (f - L\, u^n)$$

where

$$B = \begin{pmatrix} I & & & & & & \\ -\beta_2 D & I & & & & & \\ \alpha_2 PC & \alpha_2 P & I & & & & \\ & \cdot & \cdot & \cdot & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & -(1-\beta_M) & -\beta_M D & I & \\ & & & & \alpha_M PC & \alpha_M P & I \end{pmatrix},$$

from which it follows that

$$R = \begin{pmatrix} 0 & 0 & 0 & . & . & 0 & I \end{pmatrix} B^{-1} \begin{pmatrix} \alpha_1 P \\ 0 \\ \alpha_2 P \\ . \\ . \\ 0 \\ \alpha_M P \end{pmatrix}.$$

Therefore,

$$R^H = \begin{pmatrix} \alpha_1 P^H & 0 & \alpha_2 P^H & . & . & 0 & \alpha_M P^H \end{pmatrix} \left(B^H\right)^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ . \\ . \\ 0 \\ I \end{pmatrix},$$

and so the adjoint time-marching procedure is given by

$$v^n = v^{n+1} + \sum_{m=1}^{M} \alpha_m P^H \tilde{w}^{(m)}$$

where the quantities $\tilde{w}^{(m)}$ are defined by

$$B^H \begin{pmatrix} \tilde{w}^{(1)} \\ \tilde{d}^{(2)} \\ \tilde{w}^{(2)} \\ . \\ . \\ \tilde{d}^{(M)} \\ \tilde{w}^{(M)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ . \\ . \\ 0 \\ I \end{pmatrix} \left(g - L^H v^{n+1}\right),$$

which in turn gives the algorithm

$$\begin{aligned}
\tilde{w}^{(M)} &= g - L^H\, v^{n+1} \\
\tilde{d}^{(M)} &= -\alpha_M P^H \tilde{w}^{(M)} \\
\left.\begin{aligned}
\tilde{w}^{(m)} &= -\alpha_{m+1} C^H P^H \tilde{w}^{(m+1)} + \beta_{m+1} D^H \tilde{d}^{(m+1)} \\
\tilde{d}^{(m)} &= -\alpha_m P^H \tilde{w}^{(m)} + (1-\beta_{m+1})\, \tilde{d}^{(m+1)}
\end{aligned}\right\} & \quad m = M\text{–}1, \ldots, 2, 1. \\
v^n &= v^{n+1} + \sum_{m=1}^{M} \alpha_m P^H \tilde{w}^{(m)}
\end{aligned} \tag{4.3}$$

Changing variables to $\tilde{v}^{(m)} = P^H \tilde{w}^{(m)}$ gives the final form of the algorithm,

$$\begin{aligned}
\tilde{v}^{(M)} &= P^H \left(g - L^H\, v^{n+1}\right) \\
\tilde{d}^{(M)} &= -\alpha_M\, \tilde{v}^{(M)} \\
\left.\begin{aligned}
\tilde{v}^{(m)} &= P^H \left(-\alpha_{m+1} C^H \tilde{v}^{(m+1)} + \beta_{m+1} D^H \tilde{d}^{(m+1)}\right) \\
\tilde{d}^{(m)} &= -\alpha_m \tilde{v}^{(m)} + (1-\beta_{m+1})\, \tilde{d}^{(m+1)}
\end{aligned}\right\} & \quad m = M\text{–}1, \ldots, 2, 1. \\
v^n &= v^{n+1} + \sum_{m=1}^{M} \alpha_m \tilde{v}^{(m)}
\end{aligned} \tag{4.4}$$

5 Multigrid

The general analysis in Section 3 is also applicable to the use of preconditioned multigrid [10]. In this case, the operator R for the updating of the solution on the finest grid using calculations on the coarser grids can be expressed as

$$R \equiv P\ E\ T,$$

where T represents the transfer (or restriction) of the fine grid residual onto the coarser grid, E represents the evolution of the correction on the coarser grid (which may itself involves the use of even coarser grids) and P represents the transfer (or prolongation) of the coarse grid changes onto the fine grid.

For the adjoint iterative process, one therefore has

$$R^H = T^H\ E^H\ P^H.$$

The key observation here is that the restriction for the adjoint equations is the transpose of the prolongation for the direct equations, and *vice versa*.

6 Numerical results

Appendix A contains a MATLAB program which solves either a simple scalar o.d.e. or an upwind approximation to the convection equation with a harmonic source term, depending on the value of the parameter `icase`. In either case, the direct problem is solved with two different but mathematically equivalent forms of the direct solver, corresponding to Equations (4.1) and (4.2), and two equivalent forms of the adjoint solver, corresponding to Equations (4.3) and (4.4). In all cases, the numerical results confirm that the direct and adjoint solvers produce the same value for the functional after the same number of iterations.

The theory in this paper has also been tested with two FORTRAN programs, one of which solves the linearised Navier-Stokes equations with a harmonic source term, and the other of which solves the corresponding discrete adjoint equations. It proved to be an invaluable aid in debugging the two codes; once the bugs had been fixed, identical values (to within machine accuracy) for the functional were obtained after equal number of iterations of each code.

7 Discussion and Conclusions

In this paper we have developed an iterative procedure for solving discrete adjoint equations. It is a true adjoint of a commonly-used preconditioned Runge-Kutta time-marching algorithm for the iterative solution of the original nonlinear equations, in the sense that one obtains exactly the same functional after equal number of iterations of either the adjoint code or the linearised code on which it is based. This guarantees that the iterative convergence rate of the adjoint code is identical to that of the linear code, and the asymptotic convergence rate of the original nonlinear code.

It is hoped that, in conjunction with Automatic Differentiation techniques, this will help the development of adjoint codes for a variety of design optimisation problems which require iterative solvers.

Acknowledgments

Mihai Duta contributed greatly to the programming of the adjoint Navier-Stokes code and carried out most of the verification of its equivalence to the linearised Navier-Stokes code.

References

[1] W.K. Anderson and D.L. Bonhaus. Airfoil design on unstructured grids for turbulent flows. *AIAA J.*, 37(2):185–191, 1999.

[2] A. Carle, M. Fagan, and L.L. Green. Preliminary results from the application of automated code generation to CFL3D. AIAA Paper 98-4807, 1998.

[3] B. Christianson. Reverse accumulation and attractive fixed points. *Opt. Meth. and Software*, 3(4):311–326, 1994.

[4] B. Christianson. Reverse accumulation and implicit functions. *Opt. Meth. and Software*, 9(4):307–322, 1998.

[5] J. Elliott and J. Peraire. Practical 3D aerodynamic design and optimization using unstructured meshes. *AIAA J.*, 35(9):1479–1485, 1997.

[6] C. Faure. Splitting of algebraic expressions for automatic differentiation. Proceedings of the second SIAM Int. Workshop on Computational Differentiation, 1996.

[7] M.B. Giles and N.A. Pierce. An introduction to the adjoint approach to design. *European Journal of Flow, Turbulence and Control*, to appear, 2000.

[8] A. Jameson. Transonic flow calculations for aircraft. In F. Brezzi, editor, *Lecture Notes in Mathematics, Numerical Methods in Fluid Dynamics*, pages 391–404. Springer-Verlag, 1985.

[9] A. Jameson. Aerodynamic design via control theory. *J. Sci. Comput.*, 3:233–260, 1988.

[10] N.A. Pierce and M.B. Giles. Preconditioned multigrid methods for compressible flow calculations on stretched meshes. *J. Comput. Phys.*, 136:425–445, 1997.

[11] J. Weiss and W. Smith. Preconditioning applied to variable and constant density flows. *AIAA J.*, 33(11):2050–2057, 1995.

AppendixA MATLAB program

```
nstep = 5;

alfas(1) = 0.25;
alfas(2) = 0.166666666667;
alfas(3) = 0.375;
alfas(4) = 0.5;
alfas(5) = 1.0;

betas(1) = 1.0;
betas(2) = 0.0;
betas(3) = 0.56;
betas(4) = 0.0;
betas(5) = 0.44;

icase = input('enter icase value');

if icase == 1
  niter = 2;

  D  =  1;
  C  =  i;
  P  = 2.0;

  f = 1;
  g = 1;

elseif icase == 2
  niter = 5;

  N = 10;
  h = 1/N;

  omega = 0.1;
  e = ones(N,1);

  D = spdiags([-e 2*e -e], -1 : 1, N, N) * 0.5/h;
  C = spdiags([-e      e], -1:2:1, N, N) * 0.5/h ...
    + spdiags([    e    ],    0  , N, N) * omega*i;

  D(N,N) = 0.5*D(N,N);
  C(N,N) = - C(N,N-1);

  P  = 2.0*h + 0.01*i;

  f = ones(N,1);
  g = ones(N,1);
end

zero = zeros(size(f));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% direct -- usual                                                       %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

u = zero;

for it = 1:niter
  u0 = u;
  d  = zero;

  for n = 1:nstep
    d = (1-betas(n))*d  + betas(n)*D*u;
    u = u0 + alfas(n)*P*(f - C*u - d);
  end
end

g'*u
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% direct -- new                                                         %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

u = zero;

for it = 1:niter
  du = zero;
  dd = zero;
  r  = f - (C+D)*u;

  for n = 1:nstep
    dd = (1-betas(n))*dd  + betas(n)*D*du;
    du = alfas(n)*P*(r - C*du - dd);
  end;

  u = u + du;
end

g'*u
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% adjoint -- new                                                        %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

v = zero;

for it = 1:niter
  d = zero;
  r = g - (C+D)'*v;

  for n = nstep:-1:1
    d = d - alfas(n)*P'*r;
    v = v + alfas(n)*P'*r;
    r = - alfas(n)*C'*P'*r + betas(n)*D'*d;
    d = (1-betas(n))*d;
  end;
end;

v'*f
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% adjoint -- new (re-arranged)                                          %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

v  = zero;
dd = zero;
betas(nstep+1) = 1;

for it = 1:niter
  for n = nstep:-1:1
    if n == nstep
      dv = P'*(g-(C+D)'*v);
    else
      dv = P'*(-alfas(n+1)*C'*dv + betas(n+1)*D'*dd);
    end
    dd = (1-betas(n+1))*dd - alfas(n)*dv;
    v  =                v  + alfas(n)*dv;
  end;
end;

v'*f
```