THE UNIVERSITY of TENNESSEE at CHATTANOOGA

SIMCENTER

NATIONAL CENTER for COMPUTATIONAL ENGINEERING

Implicit Solution Algorithms

W. Kyle Anderson, Li Wang, and James Newman

2014 CFD Summer School

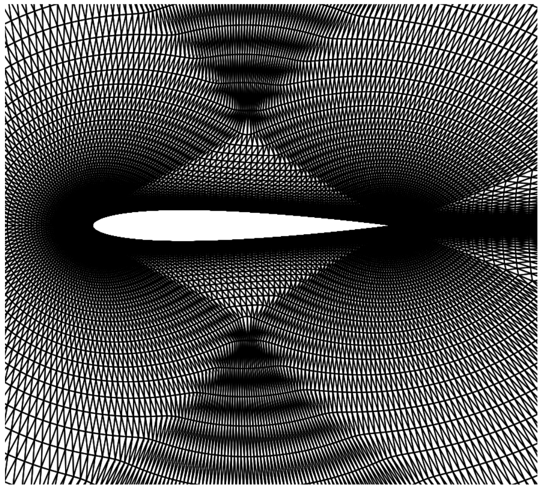
Modern Techniques for Aerodynamic Analysis and Design

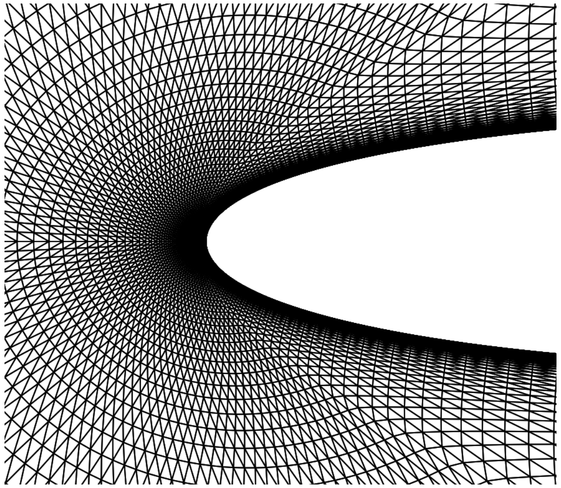
Beijing Computational Sciences Research Center

July 7-11, 2014

Motivation

- Implicit time stepping is required for turbulent flows because of very tight spacing near wall





- Effective schemes often based on Newton’s method
  - Offers fast convergence near root (but not always)
  - Not globally convergent

Newton's Method

- Generic method for solving nonlinear systems of equations

$$R\left(Q\right)=0$$

- Using Taylor series expansion

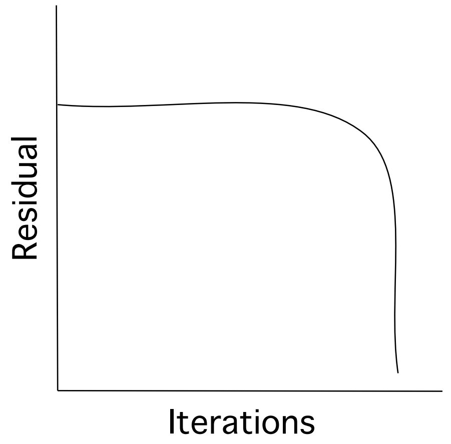
$$\cancelto{0}{R\left(Q^{n+1}\right)}=R\left(Q^{n}\right)+\left[\frac{\partial R\left(Q^{n}\right)}{\partial Q^{n}}\right]\left(Q^{n+1}-Q^{n}\right)+\cdots$$

$$\left[\frac{\partial R\left(Q^{n}\right)}{\partial Q^{n}}\right]\left(Q^{n+1}-Q^{n}\right)=-R\left(Q^{n}\right)$$

- For Euler/Navier-Stokes matrix size is determined by the number of degrees of freedom (depends on mesh and order)

Newton's Method

- Often quadratic convergence near root



- Unpredictable how many iterations required to get near root
- May not converge rapidly (e.g. roots of multiplicity n)

$$f(x) = (x-5)^8$$

- Not globally convergent

Newton's Method

- Fast convergence requires precise linearization of residual
- To improve global convergence adopt methods from optimization
  - Add diagonal term that is large initially but increases as convergence is approached (Levenberg-Marquardt)
  - Line searches
  - Reject non-physical steps

Newton's Method

- Diagonal term added to left-hand side based on time step

$$Q^{n+1} - Q^n = -\Delta t R\left(Q^{n+1}\right) = -\Delta t\left(R^n + \frac{\partial R}{\partial Q}\Delta Q\right)$$

$$\left[\frac{I}{\Delta t} + \frac{\partial R}{\partial Q}\right]\Delta Q = -R\left(Q^n\right)$$

- Essentially "explicit" method with small time step
- Newton's method obtained as time step increases to infinity
- Old technique but regaining popularity
  - Switched Evolution Relaxation (SER) time step begins low and increases strictly based on residual reduction
  - Recently approaches from optimization have been adopted

Newton's Method

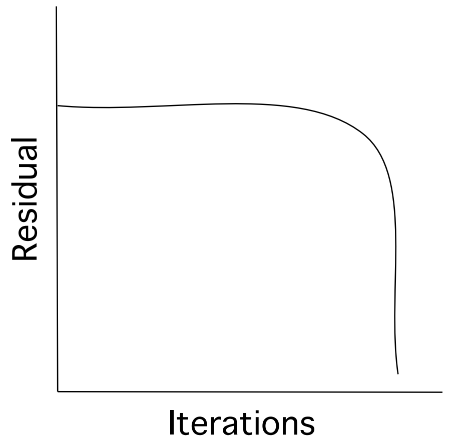
- Many variants on global methods based on optimization
- All have similar essential features
  1. Begin with small time step and solve for initial value of ΔQ
  2. Determine relaxation factor ω_1 to limit changes in physical variables and to ensure for viability (positive pressure)
  3. Without exceeding ω_1 determine ω_2 to reduce residual
  4. If relaxation too small reject step, lower CFL return to 1

$$Q^{new} = Q^{old} + \min\left(\omega_1, \omega_2\right)\Delta Q$$

  5. If relaxation factor is acceptable increase the time step and return to step 1

Newton's Method

- As mentioned previously this technique can often provide very fast convergence



- May not converge rapidly at all (e.g. roots of multiplicity n)
- Number of iterations required to get near root unpredictable so combinations with other algorithms such as multigrid are ongoing subjects of research
- Examples shown later

Linearizing the Residual

- Straight-forward application of Newton's method requires precise linearization of residual
  - Hand differentiation
  - Finite-difference method
  - Complex-variable method
  - Operator overloading

$$\left[\frac{I}{\Delta t} + \frac{\partial R}{\partial Q}\right]\Delta Q = -R\left(Q^n\right)$$

- Matrix-free GMRES only requires the effects of multiplying the linearization matrix with the vector of unknowns but still requires a "preconditioner"
- Matrix-free techniques not helpful if matrix is needed such as adjoint methods for design
- Linearization and matrix-free GMRES presented in detail

Hand Differentiation

- Hand differentiation can be accomplished two ways
- Method 1:
  - Write expressions in terms of dependent variables
  - Linearize by hand
  - Simplify

$$Q = \left(Q_1, Q_2, Q_3\right)^T = \left(\rho, \rho u, \rho E\right)^T$$

$$u = \frac{Q_2}{Q_1} \qquad \frac{\partial u}{\partial \rho} = \frac{1}{\rho^2}\left(Q_1 \frac{\partial Q_2}{\partial Q_1} - Q_2 \frac{\partial Q_1}{\partial Q_1}\right) = -\frac{u}{\rho}$$

- Typically applied for fluxes
- Executes fast but is tedious, error prone, and difficult to maintain

Hand Differentiation

- Hand differentiation can be accomplished two ways
- Method 2: Differentiate the code directly

```
rho = Q1
  rho_Q1 = 1.0
  rho_Q2 = 0.0
  rho_Q3 = 0.0

rhou = Q2
  rhou_Q1 = 0.0
  rhou_Q2 = 1.0
  rhou_Q3 = 0.0

u = rhou/rho
  u_Q1 = (rho*rhou_Q1 - rhou*rho_Q1)/(rho*rho)
  u_Q2 = (rho*rhou_Q2 - rhou*rho_Q2)/(rho*rho)
  u_Q3 = (rho*rhou_Q3 - rhou*rho_Q3)/(rho*rho)
```

- Less tedious/error prone than first method but still takes time
- Not as fast to execute but easy to do while watching TV

Finite Differences

- Finite differences are straight forward to implement
- Can be difficult to find step size to give accurate results
  - Small is better for truncation error
  - Too small and computer cannot distinguish numbers

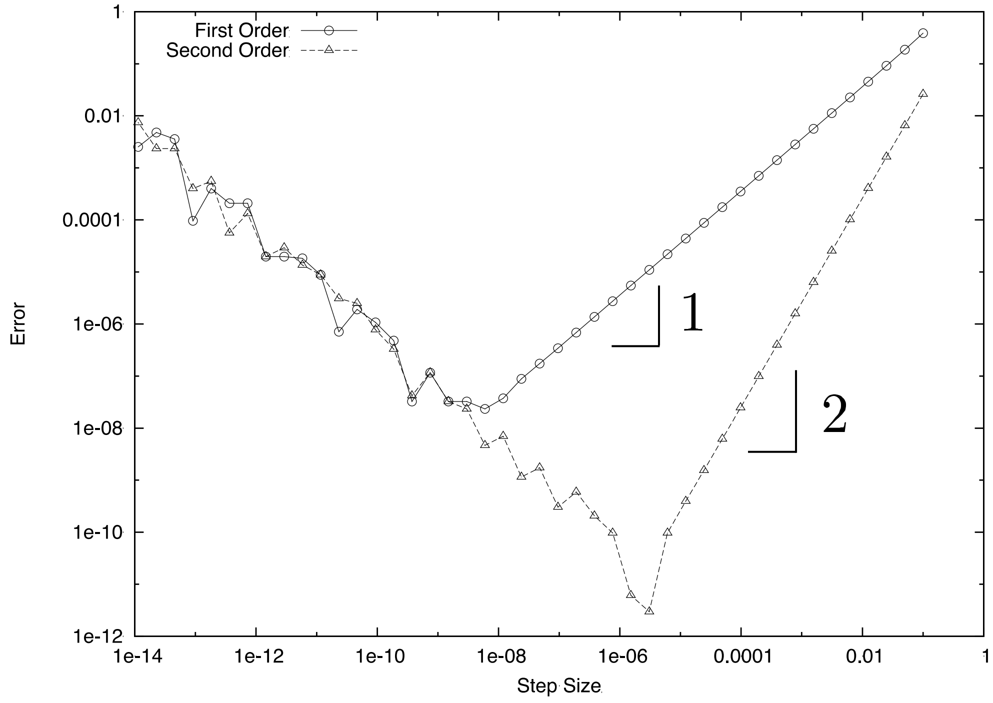
$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

First order

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{\Delta x}$$

Second order

Finite Differences



- Forward difference executes about as fast as second method of hand differentiation
- Central difference is twice as slow

Complex Variable Approach

- Complex variables eliminates subtractive cancelation errors
- Extend Taylor series for complex perturbations

$$f\left(x+ih\right)\approx f\left(x\right)+\frac{\partial f}{\partial x}ih-\frac{1}{2!}\frac{\partial^2 f}{\partial x^2}h^2-\frac{1}{3!}\frac{\partial^3 f}{\partial x^3}ih^3+\ldots$$

- Examine imaginary part

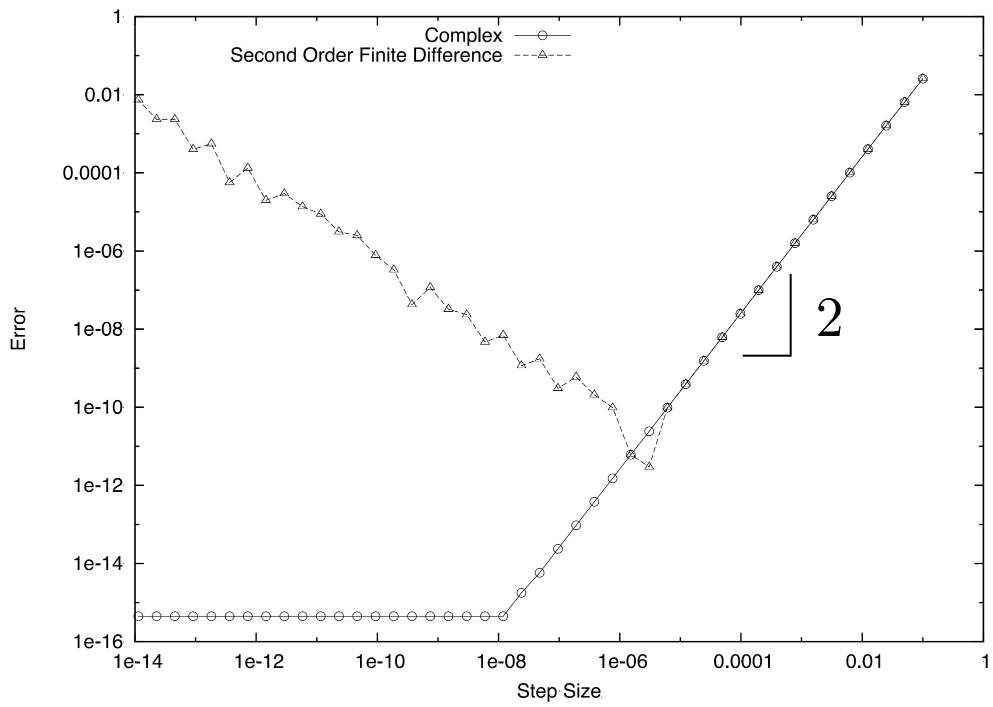
$$\text{Imaginary part }\left(f\left(x+ih\right)\right)\approx\frac{\partial f}{\partial x}h-\frac{1}{3!}\frac{\partial^3 f}{\partial x^3}h^3+\ldots$$

- Solve for derivative

$$\frac{\partial f}{\partial x}\approx\frac{\text{Imaginary part }\left(f\left(x+ih\right)\right)}{h}+\vartheta\left(h^2\right)$$

- Truncation error is second order but does not suffer from subtractive cancelation errors

Complex Variable Approach



•With complex step size accuracy can be achieved without subtractive cancelation error

•Execution time similar to central finite-difference scheme

Operator Overloading

•Modern computer languages (C++, Fortran 95) allow the user to define data types and to “overload” operators

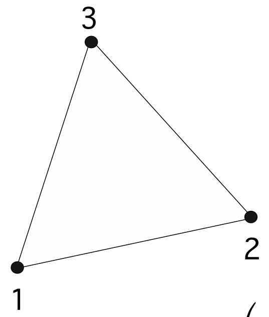
•Data type can be defined to hold the value as well as an array that represents derivatives with respect to variables of interest

•Operations (addition, subtract, multiplication, division, etc.) are all overloaded so that when the residual routine is executed the derivatives are automatically accumulated

•In principle, this is very similar to the second method of hand differentiation but care must be taken to avoid poor performance

Operator Overloading

•Consider the following triangle



•At each grid point 4 variables are stored $\left(Q_1, Q_2, Q_3, Q_4\right)$ so that derivatives of the residuals at nodes 1-3 will be taken with respect to a total of 12 variables

•The residual is typically accumulated by performing operations within the triangle and distributing results so all variables need to be declared as derived data type and all operations in residual routine need to be overloaded

Operator Overloading

- Consider the following derived data type in Fortran95

```
integer, parameter :: maxOverloadLength = 12
integer :: overloadLength = -66

type :: dType
  real(dp) :: value
  real(dp), dimension(maxOverloadLength) :: deriv
end type dType
```

- The values of the derivatives are stored as follows

1	2	3	4	5	6	7	8	9	10	11	12
Node 1				Node 2				Node 3			
$\frac{\partial(\ )}{\partial Q_1}$	$\frac{\partial(\ )}{\partial Q_2}$	$\frac{\partial(\ )}{\partial Q_3}$	$\frac{\partial(\ )}{\partial Q_4}$	$\frac{\partial(\ )}{\partial Q_1}$	$\frac{\partial(\ )}{\partial Q_2}$	$\frac{\partial(\ )}{\partial Q_3}$	$\frac{\partial(\ )}{\partial Q_4}$	$\frac{\partial(\ )}{\partial Q_1}$	$\frac{\partial(\ )}{\partial Q_2}$	$\frac{\partial(\ )}{\partial Q_3}$	$\frac{\partial(\ )}{\partial Q_4}$

Operator Overloading

- Generic interfaces need to be declared for all operations and for a data types used in the operation
- The overloaded functions must be defined for all types

```
interface operator(*)
  module procedure real_real_multiply
  module procedure integer_real_multiply
  module procedure real_integer_multiply
  module procedure const_real_multiply
  module procedure real_const_multiply
end interface

elemental function real_real_multiply(v1,v2)
  type (dtype) :: real_real_multiply
  type (dType), intent(in) :: v1,v2
  real_real_multiply%value = v1%value * v2%value
  real_real_multiply%deriv(1:overloadLength) = &
    (v1%value * v2%deriv(1:overloadLength)) &
  + (v2%value * v1%deriv(1:overloadLength))
end function real_real_multiply
```

Operator Overloading

- Local residual values will need to be initialized to zero
- Local independent variables need to be defined so that their value and derivatives are set properly

```
            overloadLength = 12
            do j = 1,npe
             call initialize(rhs(1,j),0,0.)
             call initialize(rhs(2,j),0,0.)
             call initialize(rhs(3,j),0,0.)
             call initialize(rhs(4,j),0,0.)
           end do
!
! Initialize independent variables
!
           index = 0
           do j = 1,npe
             index = index + 1
             call initialize(local_rho(j), index, node(local_node(j))%q(1))
             index = index + 1
             call initialize(local_u(j), index, node(local_node(j))%q(2))
             index = index + 1
             call initialize(local_v(j), index, node(local_node(j))%q(3))
             index = index + 1
             call initialize(local_T(j),  index, node(local_node(j))%q(4))
           end do
```

Operator Overloading

- Initialization routine sets residual value and derivatives to zero
- Set values for independent variables and initialize derivative with respect to self to unity while others are set to zero

```
      subroutine initialize(v,index,value)
        type (dType) :: v
        integer :: index
        real(dp) :: value
!
! Set the value and the derivatives
!
        v%value = value
        v%deriv = 0.
        if(index.ne.0)then
          v%deriv(index) = 1.0
        end if
      end subroutine initialize
```

Operator Overloading

- After initialization the derivative array for the density at node 2 is seeded as follows

1	2	3	4	5	6	7	8	9	10	11	12
Node 1				Node 2				Node 3			
0	0	0	0	1	0	0	0	0	0	0	0

- At this point computation of residual continues as normal

Operator Overloading

- At this point computation of residual continues as normal

```
! Accumulate values at Gauss points
      rho = 0., u = 0., v = 0., T = 0.
      do k = 1,npe
        rho = rho + local_rho(k)*N(k)
        u   = u   + local_u(k)*N(k)
        v   = v   + local_v(k)*N(k)
        T   = T   + local_T(k)*N(k)
      end do
! Get pressure and energy using function
! that accepts and returns dType data
      therm = linearize_therm(rho,u,v,T)

      p    = therm(1)
      etot = therm(2)

! Continuity
      f = rho*u
      g = rho*v
      do k = 1,npe
        rhs(1,(k)) = rhs(1,(k))
        - weight*area*jacobian*(Nx(k)*f + Ny(k)*g)
      end do
!
! x-momentum similar
! y-momentum similar
! Energy similar
!
```

- Local residual array will be filled with derivatives that can then be distributed into appropriate entries in the matrix

Operator Overloading

- A naive implementation is quite simple once a module is developed that has all the operators defined for all the argument types
- Execution speed can be very poor if not careful
  - Loops may get created for every operation
  - Results of loops may be stored in temporary variables

Operator Overloading

- Optimization report from Intel compiler indicates 17 loops (8 multiplies 9 additions) vectorized for the following line of code

```
delF4 = u*tauxx_x + tauxx*ux &
      + v*tauxy_x + tauxy*vx - qx_x &
      + u*tauxy_y + tauxy*uy &
      + v*tauyy_y + tauyy*vy - qy_y
```

- Note that compiler creates loops and temporary variables in overloaded functions

Loop

```
elemental function real_real_add(v1,v2)
  type (dtype) :: real_real_add
  type (dType), intent(in) :: v1,v2
  real_real_add%value = v1%value + v2%value
  real_real_add%deriv(1:overloadLength) =
    v1%deriv(1:overloadLength) + v2%deriv(1:overloadLength)
end function real_real_add
```

Operator Overloading

- To increase performance consider the following code

```
type (dType) :: result,a,b,c
do i = 1,n
  result = a + b*c
end do
```

- Assuming functions are inlined expect one loop for multiplication and one for addition

```
type (dType) :: result,a,b,c
do i = 1,n
  temp%value = b%value * c%value
  do j = 1,overloadLength
    temp%deriv(j) = b%value * c%deriv(j) &
                  + c%value * b%deriv(j)
  end do
  result%value = a%value + temp%value
  do j = 1,overloadLength
    result%deriv(j) = a%deriv(j) + temp%deriv(j)
  end do
end do
```

Operator Overloading

- Desire is to have one loop and no temporary variables

```
type (dType) :: result,a,b,c
do i = 1,n
  result%value = a%value + b%value * c%value
  do j = 1,overloadLength
    result%deriv(j) = a%deriv(j)         &
                + b%value * c%deriv(j)  &
                + c%value * b%deriv(j)
  end do
end do
```

- To achieve this objective
  - Make sure functions are inlined (compiler directives)
  - “Hardwire” length so loops are unrolled
  - May require more than single pass through outer loop and more complicated to distribute entries into matrix
  - Execution similar to method 2 of hand differentiation

Operator Overloading

- Baseline residual calculation

```
                do i = 1,nElements
!
! Compute residual using existing code
!
                end do
```

- Residual computed inside outer loop over elements

Operator Overloading

- Linearized residual routine

```
                                                          New
         do i = 1,nElements
!
! Initialize rhs(:,:) array of local residuals            ⇦
!
! Initialize independent variables                        ⇦
!
! Compute residual using existing code
!
! Distribute components of rhs(:,:) into matrix           ⇦
!
         end do
```

- Majority of routine remains the same
- Example Fortran95 but obviously can be implemented in C++
- Efficient implementation in C++ using expression templates

Matrix Solution

- At each iteration of the solver must obtain a solution to the matrix problem

$$\left[\frac{I}{\Delta t} + \frac{\partial R}{\partial Q}\right]\Delta Q = -R\left(Q^n\right)$$

- Usually solve with iterative method as direct inversion is too expensive for large problems
  - Point iterative methods
  - Generalized Minimal Residual (GMRES)
- Not necessary to fully converge solution at each step

Point Iterative Method

- Consider solution of general matrix problem

$$\left[A\right]\left\{x\right\}=\left\{b\right\}$$

- For point iterative method decompose matrix into diagonal and off-diagonal contributions and solve for new values

$$\left[D\right]\left\{x\right\}^{n+1}=\left\{b\right\}-\left[O\right]\left\{x\right\}^{n/n+1}$$

- The solution values used on right-hand side may be old or previously updated (Gauss Seidel)

Point Iterative Method

- Easy to implement
- Works well in finite-volume schemes
  - Typically use linearization of first-order accurate residual
  - Relatively low CFL number so diagonally dominant
- Does not work well for non-diagonally dominant systems
  - Full linearization of residual even for second-order finite-volume schemes
  - Large time steps
- Common solution is to use a Krylov-Subspace method

GMRES

- Generalized Minimal Residual Method (GMRES) is very common Krylov-subspace method to solve linear systems
- $$Ax = b$$
- Krylov subspace of dimension k defined as

$$\left\langle b, Ab, A^2 b, A^3 b, \ldots, A^{k-1} b \right\rangle$$

- Heuristic motivation for Krylov subspace
  - Cayley-Hamilton theorem (every matrix satisfies characteristic polynomial)
  - Multiply characteristic polynomial by A^{-1} and simplify
  - Solution to linear system is linear combination of vectors in Krylov subspace

Krylov Subspace

- Cayley-Hamilton for 4x4 matrix

$$\left(A-\lambda_1 I\right)\left(A-\lambda_2 I\right)\left(A-\lambda_3 I\right)\left(A-\lambda_4 I\right)=0$$

- Multiplication by A^{-1} and solving for A^{-1}

$$A^{-1}=\alpha_1 I+\alpha_2 A+\alpha_3 A^2+\alpha_4 A^3$$

- Therefore solution is linear combination of vectors in KS

$$x=A^{-1}b=\alpha_1 b+\alpha_2 Ab+\alpha_3 A^2 b+\alpha_4 A^3 b$$

- General procedure is to generate vectors in Krylov subspace and find best solution possible (least squares problem)

GMRES

- Generate k vectors in Krylov subspace
- Assume solution is linear combination of vectors in KS and solve least squares problem

$$[A]\{x\} \approx [A][V]\{y\} = \{b\}$$

- For example if matrix is 100x100 with 3 vectors in KS

$$\underbrace{[A]}_{\text{100x100}}\ \underbrace{[V]}_{\text{100x3}}\ \underbrace{\{y\}}_{\text{3x1}} = \underbrace{\{b\}}_{\text{100x1}}$$

- Classic least squares problem to find best solution with more equations (100) than unknowns (3). GMRES implementation very clever and numerically robust

GMRES

- Straightforward generation of Krylov subspace is not stable
- Solution: generate orthonormal basis using Arnoldi's method

Choose initial vector v_1 with $\left\|v_1\right\| = 1$

Iterate: for $j = 1, 2, \ldots, k-1$

$$h_{i,j} = \left(Av_j, v_i\right) \quad i = 1, 2, \ldots, j$$ Inner product

$$\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_j$$ Gram Schmidt

$$h_{j+1,j} = \left\|\hat{v}_{j+1}\right\|$$

$$v_{j+1} = \hat{v}_{j+1} \Big/ h_{j+1,j}$$ Normalize

GMRES

- After k steps of the Arnoldi process there are $\left(k+1\right)$ vectors generated that satisfy the following relation

$$A\,V_k = V_{k+1}\bar{H}_k \text{ or } V_{k+1}^T A\,V_k = \bar{H}_k$$

- Here $\bar{H}_k$ is $\left(k+1\right)\mathrm{x}\left(k\right)$ upper Hessenberg matrix generate by Arnoldi's method

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ 0 & h_{32} & h_{33} & h_{34} \\ 0 & 0 & h_{43} & h_{44} \\ 0 & 0 & 0 & h_{54} \end{bmatrix}$$

GMRES

- Consider solution to $Ax = b$ with initial guess x_0

$$A\left(x_0 + z\right) = b \quad \text{or} \quad Az = b - Ax_o = r_0$$

- If z_k is in Krylov subspace then

$$z_k = V_k y_k$$

$$A V_k y_k = r_0$$

- Least squares problem as discussed previously

$$\min_{z_k \in K_k} \left\|r_0 - Az_k\right\|$$

GMRES

- Consider this as minimizing a function of y_k

$$J\left(y_k\right) = \left\|r_0 - AV_k y_k\right\| = \left\|\beta v_1 - AV_k y_k\right\|$$

$$\beta = \left\|r_0\right\| \quad \left(\beta v_1 = \left\|r_0\right\| \frac{r_0}{\left\|r_0\right\|}\right) \qquad v_1 = \frac{r_0}{\left\|r_0\right\|}$$

- Recalling that $AV_k = V_{k+1}\bar{H}_k$

$$J\left(y_k\right) = \left\|V_{k+1}\left(\beta e_1 - \bar{H}_k y_k\right)\right\| = \left\|\left(\beta e_1 - \bar{H}_k y_k\right)\right\|$$

- So now least squares problem is simply

$$\underbrace{\bar{H}_k}_{(k+1)\text{ x }k} \quad \underbrace{y_k}_{(k+1)} = \underbrace{\beta e_1}_{(k+1)\text{ x }k}$$

GMRES

- The implementation in GMRES is done so that the least squares problem is solved every time a vector is added to the Krylov subspace.
- At each step the least squares solution is a $(k+1)$ x k matrix that is solved using Given’s rotations so the error in the linear system is easily determined

$$\begin{array}{r} k \text{ x } k \text{ upper triangular} \Longrightarrow \\[2em] 1 \text{ x } k \Longrightarrow \end{array} \left[\begin{array}{c} R \\ \hline 0 \end{array}\right]\{y\} = \left\{\begin{array}{c} c \\ \hline d \end{array}\right\} \begin{array}{l} \Longleftarrow k \text{ x } 1 \\[2em] \Longleftarrow \text{Error 1x1} \end{array}$$

- The trick in the implementation is that each time a new vector is added to the Krylov subspace, all previous Given’s rotations need to be applied as if new vector were always present

GMRES

- Note that during the Arnoldi process the algorithm breaks down if $h_{j+1,j} = 0$. This occurs when $\hat{v}_{j+1} = 0$ which implies that there are no more independent vectors. In this case V_k spans the space and the exact solution is obtained. This is referred to as "lucky breakdown"
- If initial guess is an eigenvector answer obtained in one step
- For n x n matrix GMRES terminates in at most n steps
- Algorithm converges much faster with a preconditioner

GMRES

- For larger matrices algorithm may take significant number of search directions to converge
- Algorithm accelerated by preconditioner

$$M^{-1}Ax = M^{-1}b$$ Left preconditioning

$$AM^{-1}\left(Mx\right) = AM^{-1}y = b$$ Right preconditioning

Solve for y then $x = M^{-1}y$

- Preconditioner should approximate the inverse of A and should be inexpensive to apply

GMRES

- Note that in using preconditioner M^{-1} is rarely formed

$$r_0 = M^{-1}\left(b - Ax_0\right)$$

$$Mr_0 = \left(b - Ax_0\right)$$

- Examples of preconditioners
  - Diagonal
  - Point iterative scheme (if it will converge)
  - Matrix factorization $\left(M\right) = \left(L + D\right)D^{-1}\left(D + U\right)$
  - Incomplete LU decomposition

GMRES

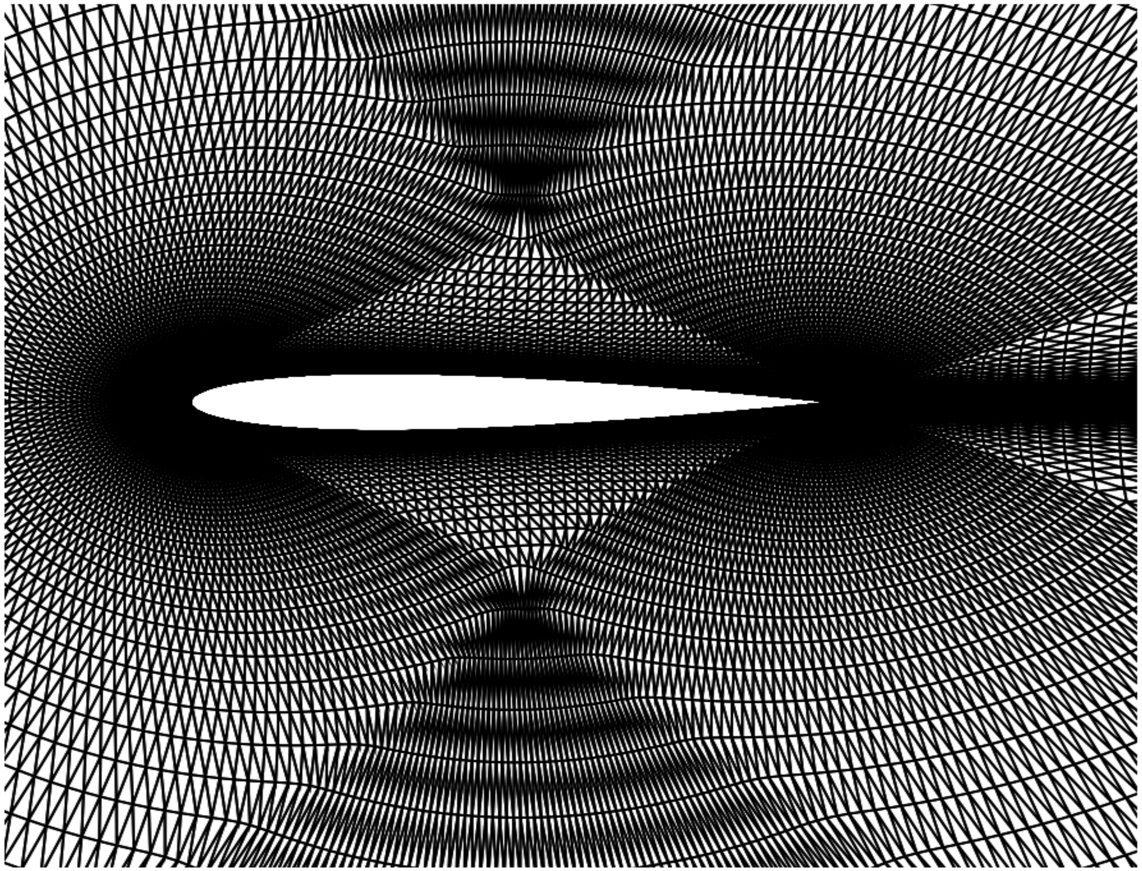
- Note that GMRES does NOT require that the matrix actually be formed
- Only matrix-vector products are required, which can be approximated using finite-difference type formulas and only require evaluation of the residual

$$Av \approx \frac{R\left(Q + \varepsilon v\right) - R\left(Q\right)}{\varepsilon}$$

- Can also be implemented using complex-variable approach
- Matrix-free GMRES still requires a preconditioner and can not be used in adjoint methods for design

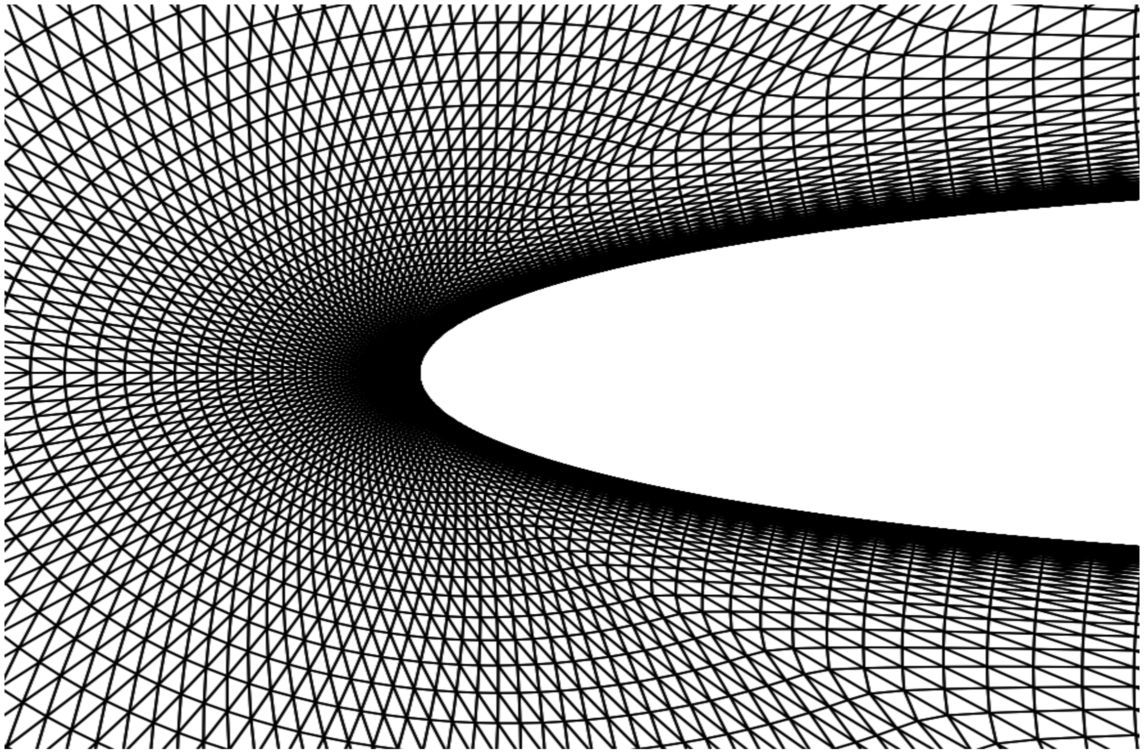
Results

Grid for turbulent flow over NACA 0012 Airfoil (15,576 DOF)



Results

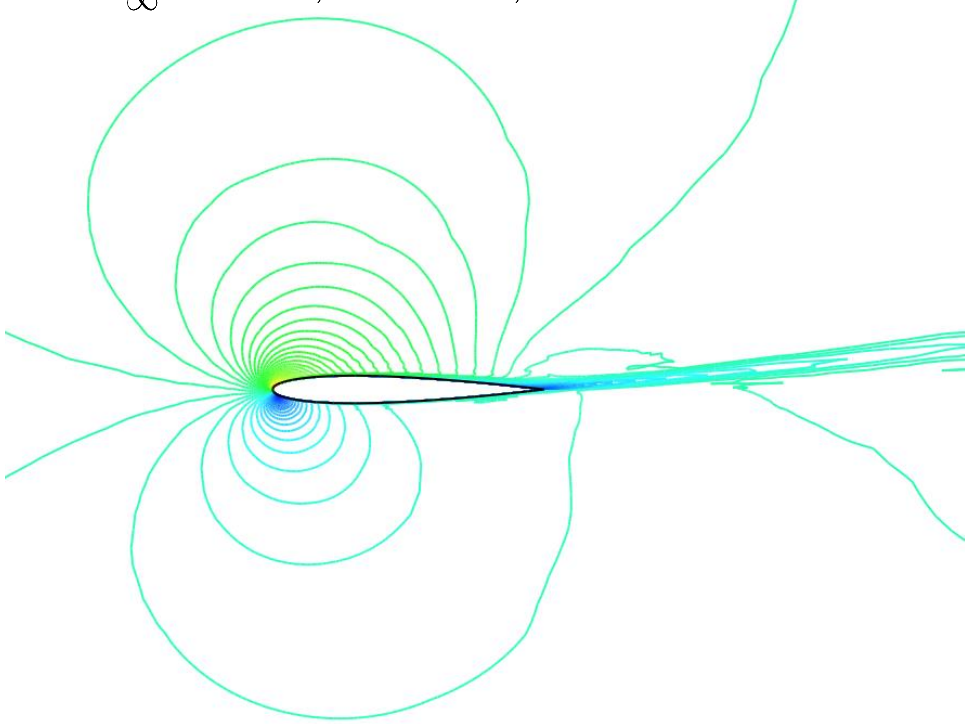
Grid for turbulent flow over NACA 0012 Airfoil (15,576 DOF)



Results

Third-order turbulent flow over NACA 0012 Airfoil

$$M_\infty = 0.15,\ \alpha = 10^0,\ \mathrm{Re} = 6 \text{ x } 10^6$$

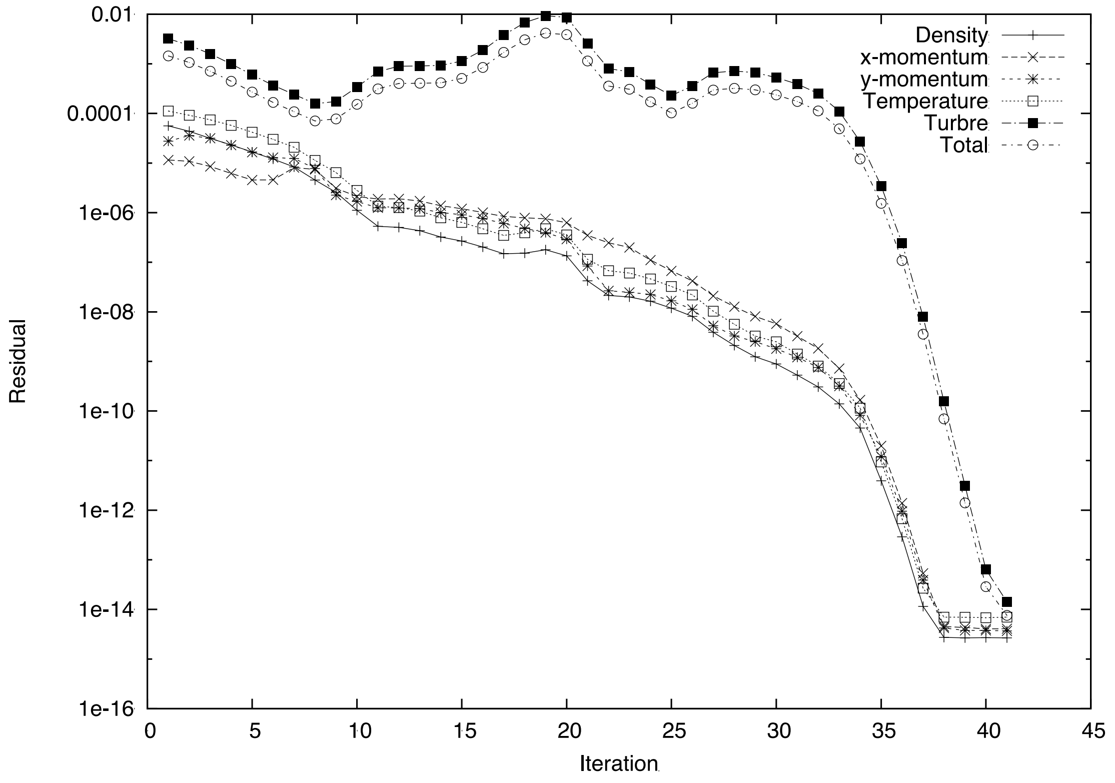


Mach Contours

Results

Third-order turbulent flow over NACA 0012 Airfoil

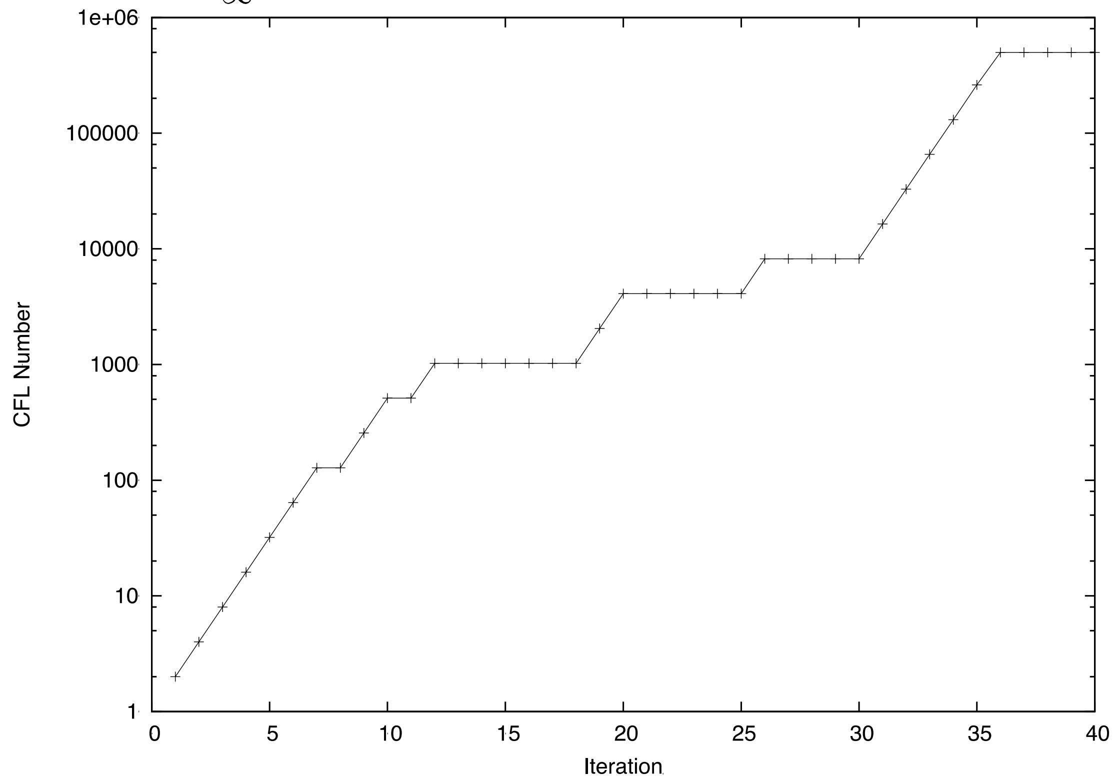
$$M_\infty = 0.15,\ \alpha = 10^0,\ \mathrm{Re} = 6 \text{ x } 10^6$$



Results

Third-order turbulent flow over NACA 0012 Airfoil

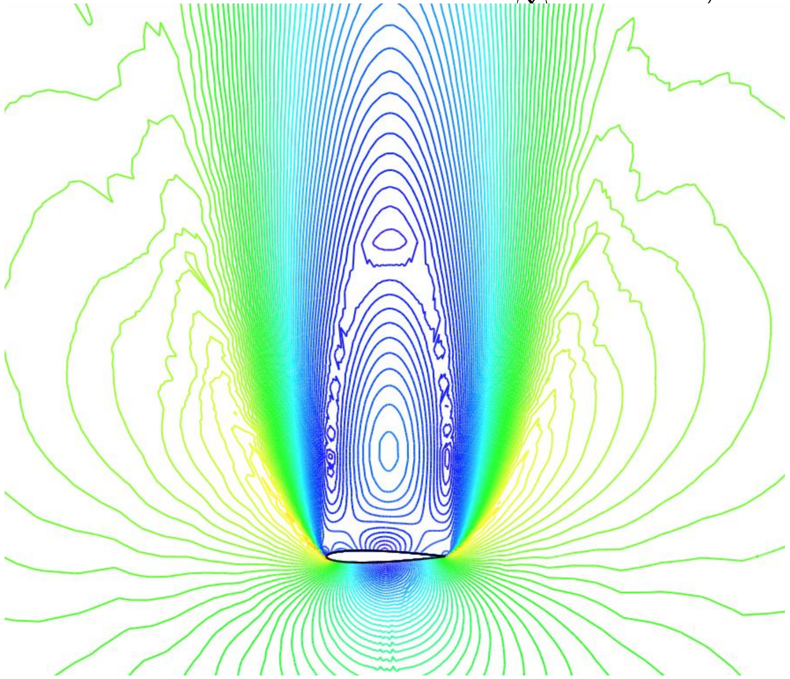
$$M_\infty = 0.15,\ \alpha = 10^0,\ \mathrm{Re} = 6 \text{ x } 10^6$$

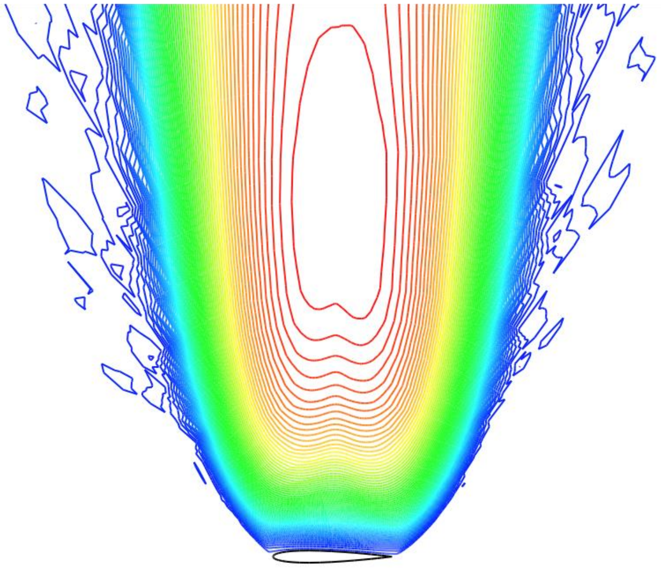


Results

Third-order turbulent flow over NACA 0012 Airfoil

$$M_\infty = 0.15,\ \alpha = 90^0,\ \mathrm{Re} = 6 \text{ x } 10^6$$

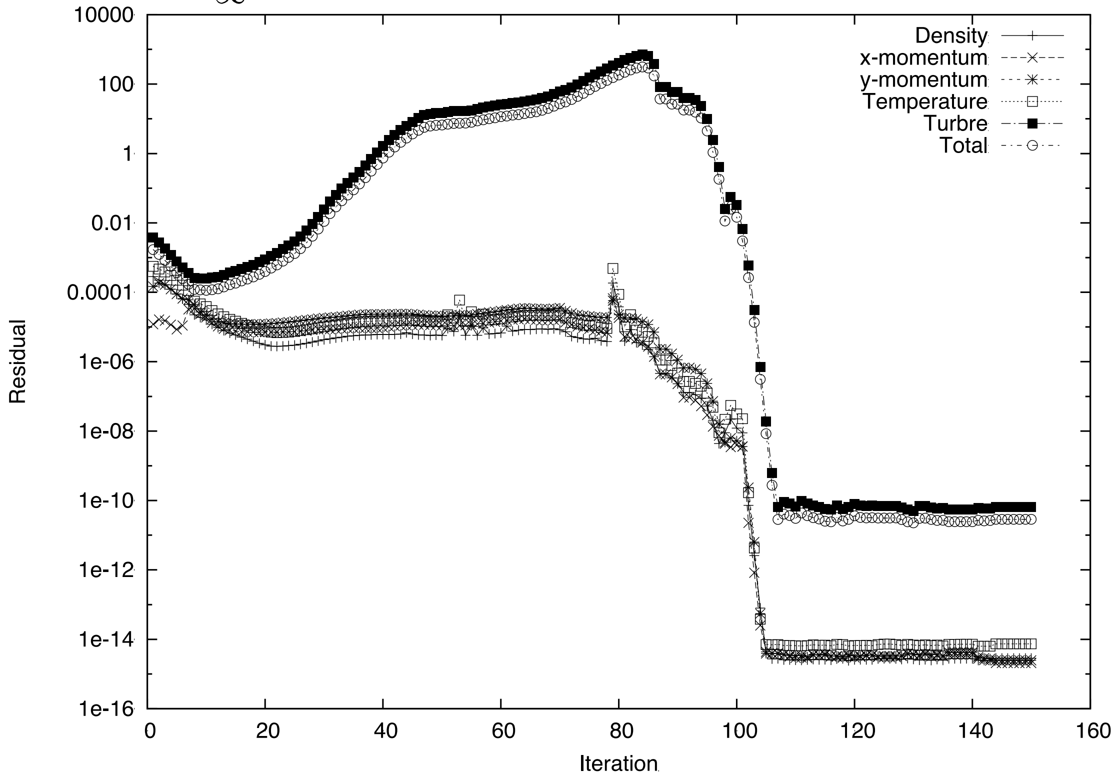

Mach Number


Turbulence Working Variable

Results

Third-order turbulent flow over NACA 0012 Airfoil

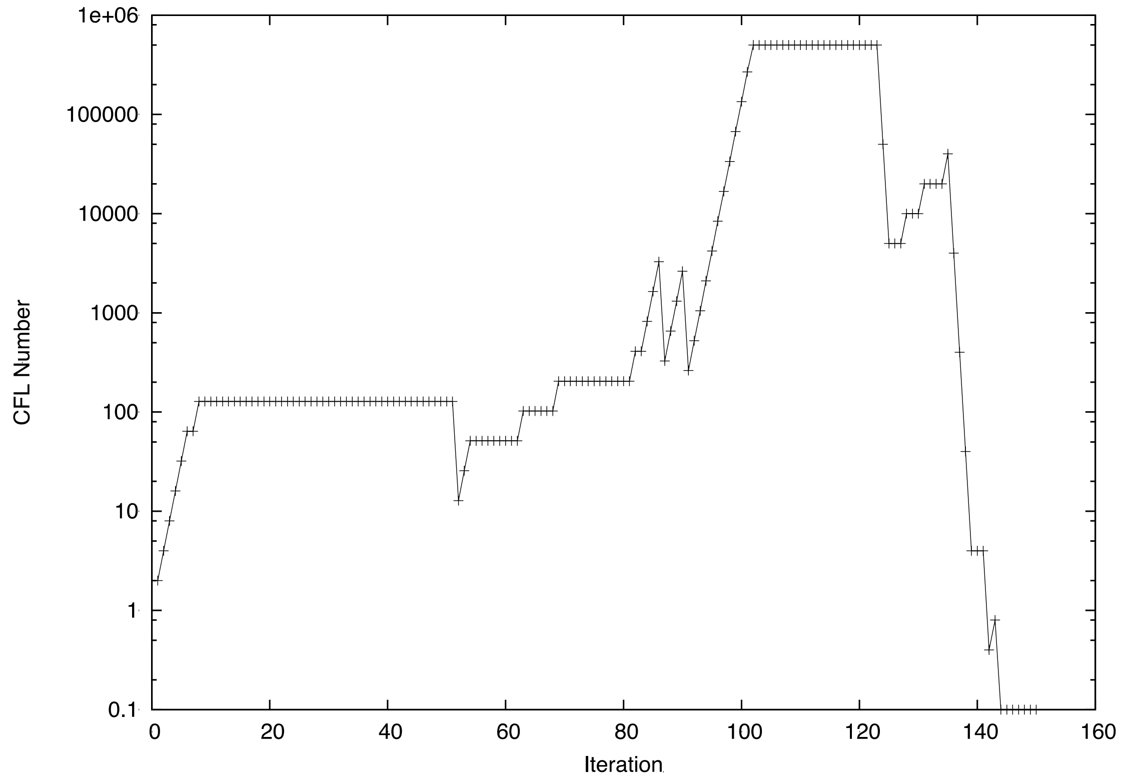
$M_\infty = 0.15,\ \alpha = 90^0,\ \mathrm{Re} = 6 \text{ x } 10^6$



Results

Third-order turbulent flow over NACA 0012 Airfoil

$$M_\infty = 0.15,\ \alpha = 90^0,\ \text{Re} = 6 \text{ x } 10^6$$

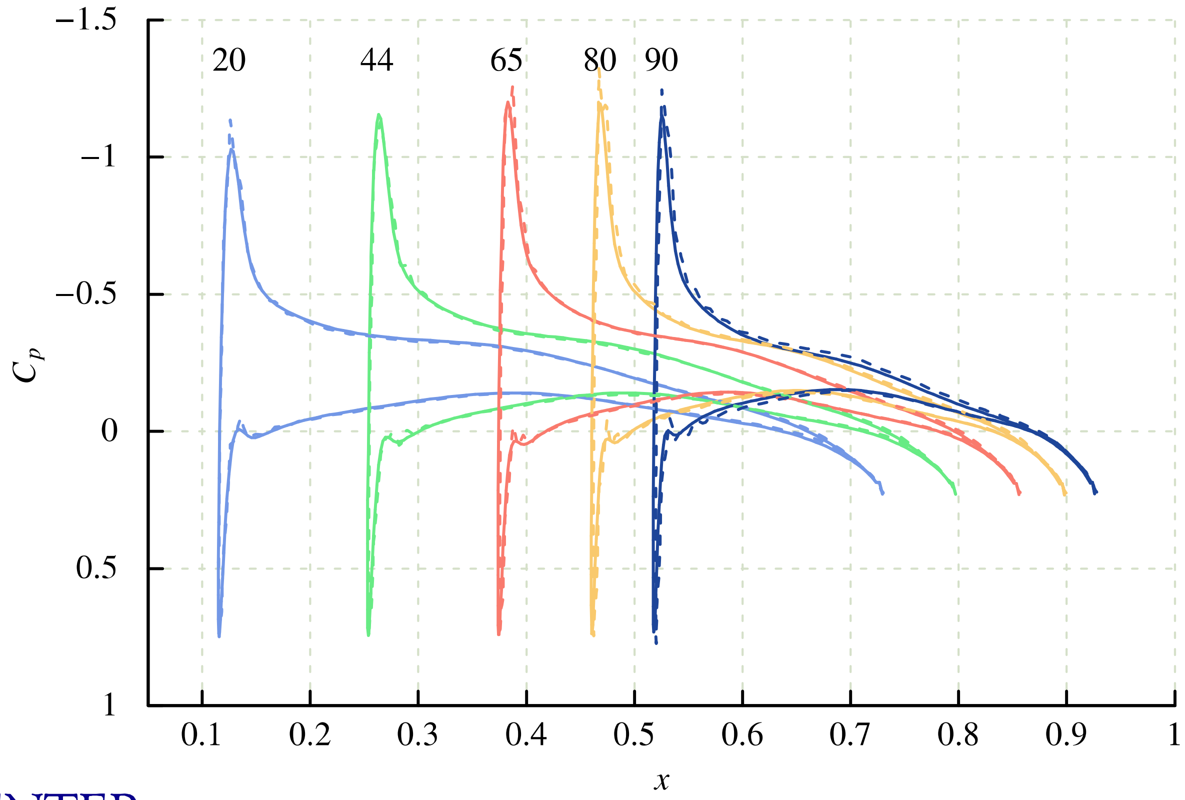


Results

Third-order turbulent flow over ONERA M6 Wing

FUNSAFE compared with CFL3D

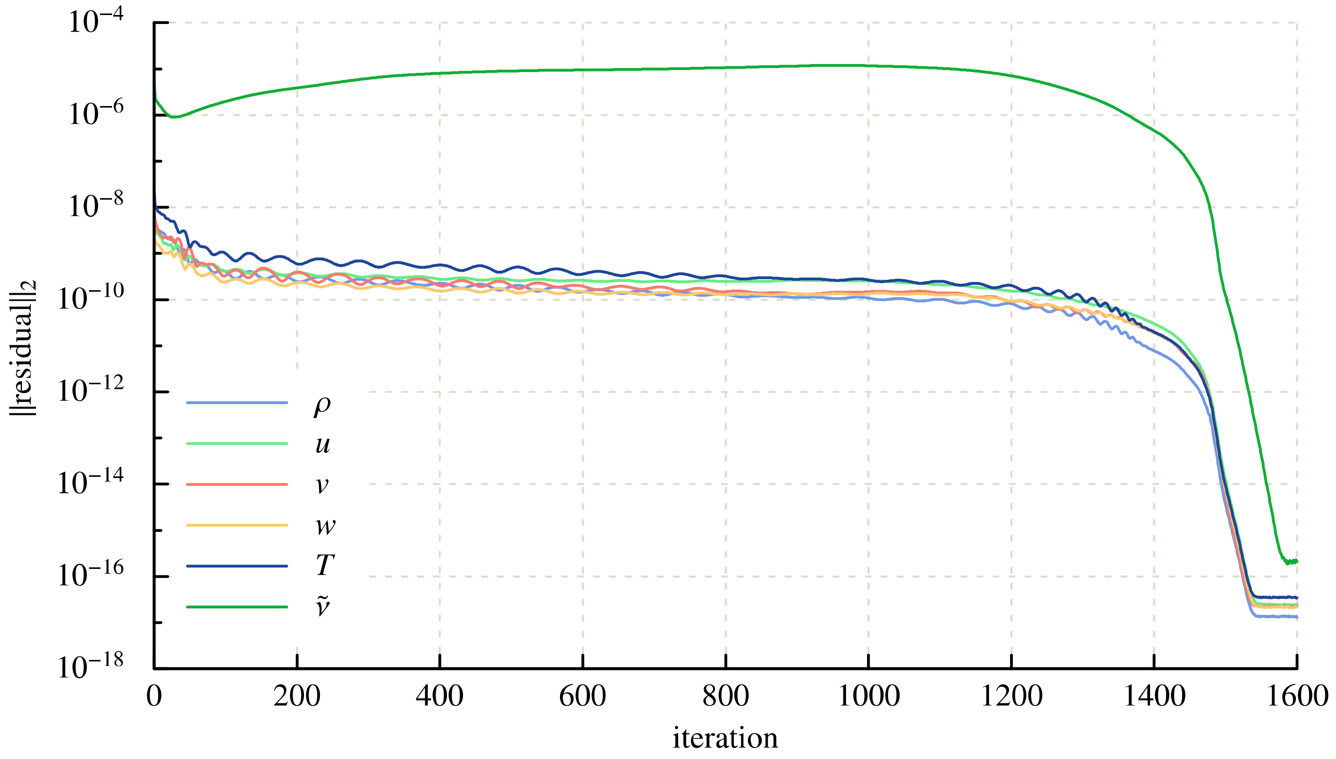
$M_\infty = 0.3,\ \alpha = 3.06^0,\ \mathrm{Re} = 11.27 \text{ x } 10^6$



Results

Third-order turbulent flow over ONERA M6 Wing

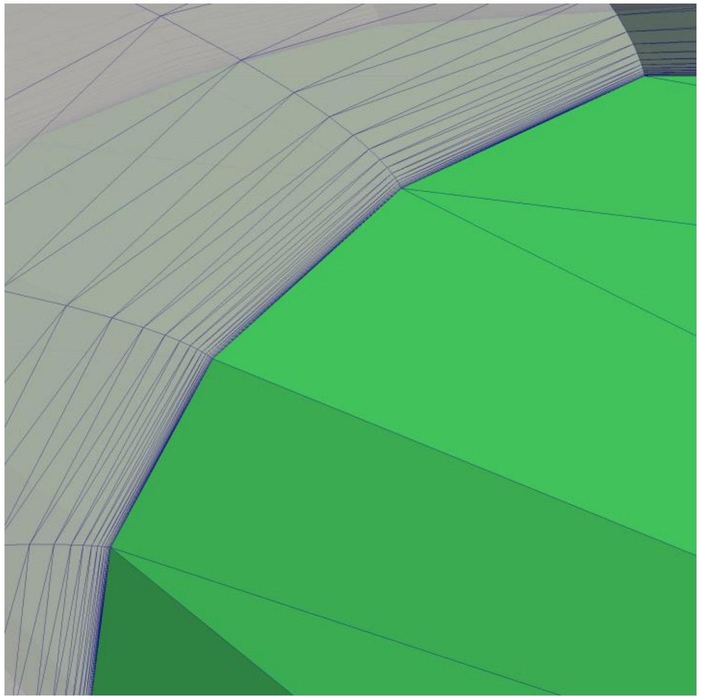
$M_\infty = 0.2,\ \alpha = 3.06^0,\ \mathrm{Re} = 11.27 \text{ x } 10^6$

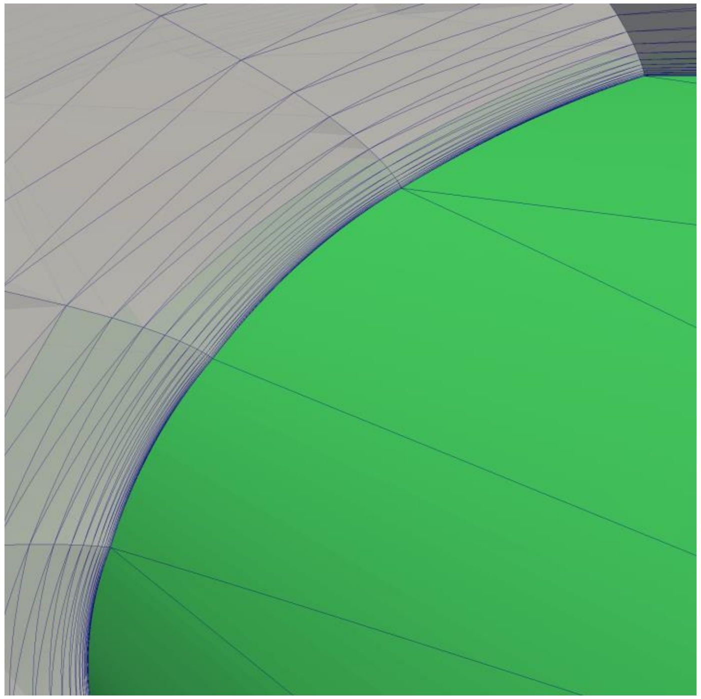


Results

Third-order turbulent flow over ONERA M6 Wing

Effect of curving surface

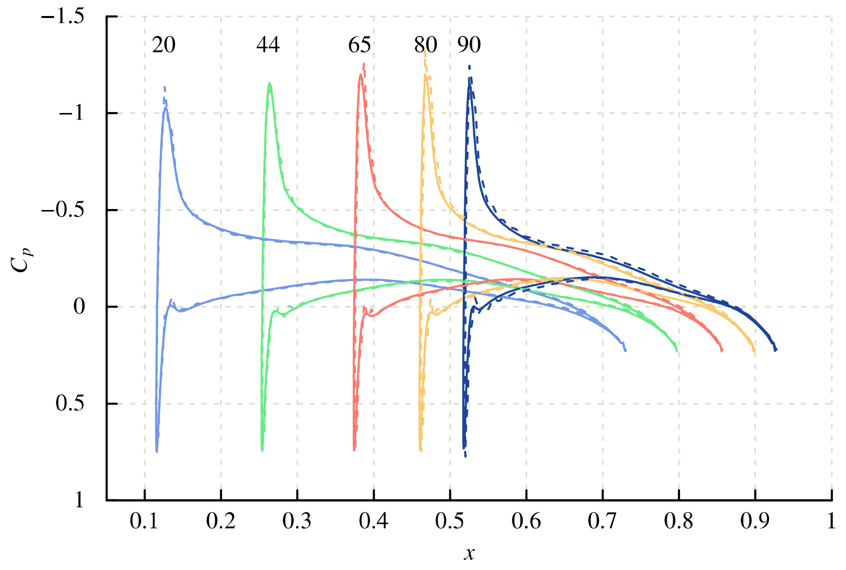

Before curving surface


After curving surface

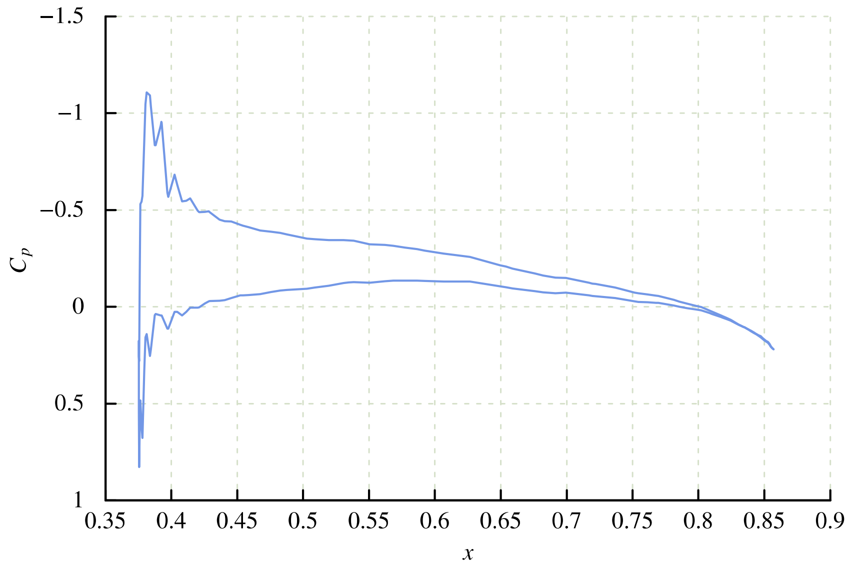
Results

Third-order turbulent flow over ONERA M6 Wing

Effect of curving surface



After curving surface



Before curving surface

Summary

- Implicit algorithm described based on Newton's method
- Various methodologies for obtaining exact linearizations
  - Hand differentiation
  - Finite differences
  - Complex variables
  - Operator overloading
- GMRES algorithm described including matrix-free variants
- Demonstrated results for airfoil and wing
- Future work
  - Further variations on global convergence techniques
  - Acceleration out of "flat" convergence region

Suggested Reading

- Nocedal, J., and Wright, S.J., Numerical Optimization, Springer, 1999.
- Burgess, N.K., and Glasby, R.S., “Advances in Numerical Methods for CREATE-AV Analysis Tools,” AIAA 2014-0417.
- Ceze, M., “A Robust hp- Adaptation Method for Discontinuous Galerkin Discretization Applied to Aerodynamic Flows,” Ph.D. Thesis, University of Michigan, 2013.
- Anderson, W.K., Rausch, R.D., and Bonhaus, D.L., “Implicit/Multigrid Algorithms for Incompressible Turbulent Flows on Unstructured Grids,” J. Comp. Phys., Vol. 128, No. 2 1996.

Suggested Reading

- Mulder, W. A., “Multigrid Relaxation for the Euler Equations,” J. Comp. Phys., Vol. 60, No. 2, 1985.
- Aubert, P., Di Cesare, N., and Pironneau, O., “Automatic Differentiation in C++ using Expression Templates and Application to a Flow Control Problem,” Computing and Visualization in Science, Vol. 3, No. 4, Jan. 2001, pp. 197-208
- Squire, W., and Trapp, G., “Using Complex Variables to Estimate Derivatives of Real Functions,” SIAM Review, Vol. 10, No. 1, March 1998, pp. 110-112
- Lyness, J.N., and Moler, C.B., “Numerical Differentiation of Analytic Functions,” SIAM J. Numer. Anal., Vol. 4, 1967, pp. 202-210.

Suggested Reading

- Newman, J.C., Anderson, W.K., and Whitfield, D.L., "Multidisciplinary Sensitivity Derivatives Using Complex Variables," MSSU-COE-ERC-98-08, June, 1998.
- Anderson, W.K., Newman, J.C., and Whitfield, D.L., "Sensitivity Analysis for the Navier-Stokes Equations on Unstructured Meshes using Complex Variables," AIAA Journal, Vol. 39, No. 1, 2001, pp. 56-63 (also AIAA 99-3294)
- Newman, J.C., Whitfield, D.L., and Anderson, W.K., "A Step-Size Independent Approach for Multidisciplinary Sensitivity Analysis," J. of Aircraft, Vol. 40, No. 3, 2003, pp. 566-573. (also see AIAA 99-3101)

Suggested Reading

- Anderson, W.K., and Bonhaus, D.L., “Airfoil Design on Unstructured Grids for Turbulent Flows,” AIAA Journal, Vol. 37, No. 2, 1999, pp. 185-191.
- Nielsen, E.J., and Anderson, W.K., “Aerodynamic Design Optimization on Unstructured Meshes using the Navier-Stokes Equations,” AIAA Journal, Vol. 37, No. 11, 1999, pp. 1411-1419.
- Saad, Y., and Schultz, M., “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systemsm” SIAM J. Sci. Stat. Comp., Vol. 7, 1986, pp. 856-869.