Performance Evaluation of ObjectMath on a Partial Differential Equation Application

Deliverable D8.9.3/1

The PREPARE Consortium

Release: 1.6
Date: March 28, 1995
Status: release
Confidentiality: public
Reference: PELAB-8-PartDiffPerformance

Contents

Abstract

This report describes an evaluation of ObjectMath, a language and environment for scientific computing. The application used for the evaluation was a solver for computational fluid dynamics (CFD) problems involving numerical solution of partial differential equations (PDEs). Certain components of the solver have been modeled in ObjectMath and from this model, code has been generated. The performance of different versions of the integrated system has been compared to that of the original solver. The objectives with this approach are to obtain greater flexibility in modeling engineering problems. As engineering models based on PDEs are typically very compute intensive, this report investigates if any performance penalties are present using this approach and defines an approach for using parallel computers via the ObjectMath environment.

1 Introduction

ObjectMath [6], [2], [3] is an environment for object-oriented mathematical modeling and computation. Object-oriented mathematical modeling is a very powerful tool when building engineering applications compared to standard programming languages such as Fortran. ObjectMath is built upon the well known language of Mathematica [7], [4], but primitives for structuring using classes and inheritance have been added, in ObjectMath. The environment also supports generation of efficient code in C++ and Fortran 90.

This task concerns the application of the ObjectMath environment to a mathematical model based on partial differential equations. The goal was to make a performance evaluation of the generated application and also to see how well ObjectMath is suited for modeling this kind of application, and to define a strategy for using parallel computation. Equations and boundary conditions arise from typical engineering models, that are coupled to the solution of the standard PDEs. These equations have been specified in ObjectMath and code has been generated from them. This code has been integrated with an existing solver, written in Fortran, for solving Navier-Stokes and Euler equations [1]. Examples of engineering models that normally are coupled to PDE solvers are models for turbulence, real gas effects and reacting flows etc. Typically, such models need to be tuned extensively as they contain experimental data. Therefore, a powerful and flexible programming environment is of great use for the engineer.

To make the integration possible some minor changes have been made to the solver, but its functionality hasn't changed in any significant ways. The efforts of integration have shown that there is a need for a way to pass multi-dimensional data structures between the solver and ObjectMath. The version of ObjectMath that has been used for this evaluation (2.1) meets this need by the use of the M++ matrix library.

There exists a version of ObjectMath which generates Fortran 90 code, but currently only for ODE equations. When full support for Fortran 90 is available, the engineering models mentioned above will through the ObjectMath environment be able to generate Fortran 90 code. As will be explained below Fortran 90 arrays are well suited to be mapped into different architectures of parallel computers.

2 ObjectMath and Mathematica

The purpose of this project has been to make an evaluation of ObjectMath's usefulness and performance of generated code in handling an application specified by partial differential equations. We will here describe this tool and the language on which it is built, Mathematica.

2.1 Mathematica—A Symbolic Mathematical Language

Since the introduction of the first version of Mathematica in 1988, it has rapidly become one of the standard symbolic formula manipulation environments for scientists, engineers, researchers and many others. Mathematica supports numerical and symbolical computation and graphical visualization. The use of pattern matching and recursion aids implementation in a compact and readable form. Its multitude of libraries with functions for most areas of mathematics, makes it popular and easy to use for a wide range of applications. Mathematica also has a wide range of tools for producing two- and three-dimensional graphics. All this gives Mathematica an advantage compared to standard engineering programming languages such as Fortran.

The simple example below shows how to define and use a function in Mathematica. `In[1] :=` is an input prompt and `Out[2] :=` is an output prompt. The expression can later be referenced by the number between the braces. As seen, the expression is handled symbolically. Mathematica has many built-in functions, like Expand, for symbolic manipulations.

```
In[1] := f[x_] := x^2

In[2] := f[3x + x^2]

                      2 2
Out[2] := (3 x + x )

In[3] := Expand[f[(3x + x^2)]]

                2      3    4
Out[3] := 9 x  + 6 x  + x
```

One major drawback of Mathematica, is the lack of a way to structure the programs well. Especially an object-oriented approach with possibility to reuse parts of models through inheritance would be desired. Another, more important drawback was the lack of facilities for generating efficient sequential or parallel code. These have been the motives behind the development of ObjectMath.

2.2 ObjectMath

ObjectMath was developed to meet the need for generating numerical code from high-level mathematical models. There exist some other environments which try to support this, but they either are too specialized for a specific field of problems or lack capabilities that are important for advanced scientific computing, like good structuring support. ObjectMath was developed to be useful for a wide range of scientific problems, and to have a full support for an object-oriented environment.

The ObjectMath language is essentially Mathematica with added capabilities for classes and inheritance. The system comes with a graphical browser, which makes it easy to see the class structure and make changes to it. It has means of generating C++ code or Fortran 90 from specified functions in Mathematica.

2.2.1 Related Systems

There exist some computer algebra systems which are useful as parts of an environment for scientific computing. Examples of these are Mathematica, Scratchpad, Reduce, Maple and Macsyma. However, they lack the ability to structure the problems. There also exist some tools which have got this object-oriented structuring support, but they are not general enough. Examples of these specialized languages are: Omola which is used for simulation modeling in continuous time systems and FINGER which is used for automatic generation of finite element analysis.

2.2.2 Other Evaluations

The ObjectMath environment has already been used for applications based on ordinary differential equations. One application, which has been used frequently is a rolling bearing model. A model of the mechanical elements has been developed with equations for the contact forces. Each type of mechanical element is represented as a class. There are also abstract classes for coordinate systems and contact between bodies. The object-oriented approach is well suited for solving these kind of problems. See further [2] and [6].

2.3 Parallel Strategy

With later versions of ObjectMath, which generate Fortran 90 for PDE solution, steps can be taken to reach better performance by employing fine-grained parallelization for execution on massively-parallel machines.

For the present numerical method, based on local approximation, the inherent parallelism can be exploited by assigning one mesh cell to one processor. Since the number of points is generally greater than the number of processors, the concept of *virtual processors* can be employed. Each processor is this case undertakes the required operations for more than one mesh cell, with the *VP ratio* defined as the ratio of the number of virtual to physical processors.

It is important to note that the mapping of the computational mesh onto the (virtual) processors is completely transparent to the Fortran programmer; the necessary manipulations are undertaken by the compiler. The same code can therefore be executed on machines with a different number of processors, and on different computer systems, gaining a greatly enhanced performance. See further [5].

On a higher level, there is the possibility to use domain decomposition for coarse-grained parallelization of the problem. This would improve the execution times considerably. However, the subject of interest in this report is the performance effects of integrating ObjectMath generated code with the solver. Such a low-level integration is independent of a coarse-grained parallelization, so the *relative* performance loss would be the same. Thus, since no further information would be gained, this has not had a high priority as yet.

There is also the possibility to take advantage of an appropriate combination of these two programming methodologies. The coarse-grain parallelism on the block level could be handled within a control-parallel framework, while the computation on the fine grain level of the mesh cells within each block could be efficiently undertaken using a data-parallel approach. In such a treatment, the processors of a distributed-memory, massively-parallel MIMD[1] computer could be divided into clusters, each having a (potentially unequal) number of processors. The processors within each cluster would then operate in synchronous mode, while transfer of data (block connectivity) between clusters could be undertaken by asynchronous message-passing or via a virtual shared memory.

For the implementation of this combined control/data parallel approach, suitable hardware and software are required. Fortunately, the next generation of massively-parallel computers that are currently making their appearance (e.g. CM-5 of Thinking Machines, the Cray Research MPP, the Intel Paragon, or the Parsytec GC) provide a variety of programming methodologies. This greater flexibility, combined with a significantly-enhanced computational performance, should enable substantial advances to be made in numerical simulations.

The engineering model developed in the ObjectMath environment involves only local computation. The generated data-parallel Fortran 90 code will therefore not contain any communication. This means that the code will execute efficiently on both SIMD and MIMD type of computers.

3 Computational Fluid Dynamics

3.1 Introduction

Computational fluid dynamics (CFD) has become an important tool when designing vehicles of different kinds. It gives the designer a possibility to try out his ideas at the computer, not having to build expensive prototypes to test in wind tunnels, for example. By computing the flow around the geometry, the engineer can find out factors like air pressure on different parts of the profile, thus getting a measure of lift force for an aircraft wing (see figure 1) or the air resistance on a car. The engineer can find the heating on different areas of a space shuttle that reenters into atmosphere. CFD can guide in choosing a boat profile that has good fluid properties. It is easy to change temperature, pressure, speed, angle of attack, geometry etc. to conduct the tests under different conditions.

The increase of computational power, due to hardware improvements in the last decades and parallelization techniques, has made the method usable for bigger problems and thus more interesting in different fields.

3.2 Partial Differential Equations

When doing CFD, the properties of the fluid is modeled by some kind of differential equations. For fluid dynamics problems in air, the theory is based on the so called Euler equations and the related compressible Navier-Stokes equations. They express the conservation of energy, mass and momentum in each point and the sum of all gas flows to and from each point in a grid.

The methods for solving these equations differ from those used to solve ordinary differential equations. Common is that a Runge-Kutta scheme often is used, to march in time until steady

[1] multiple instruction multiple data, as opposed to SIMD, single instruction multiple data, the other main category of parallel computation

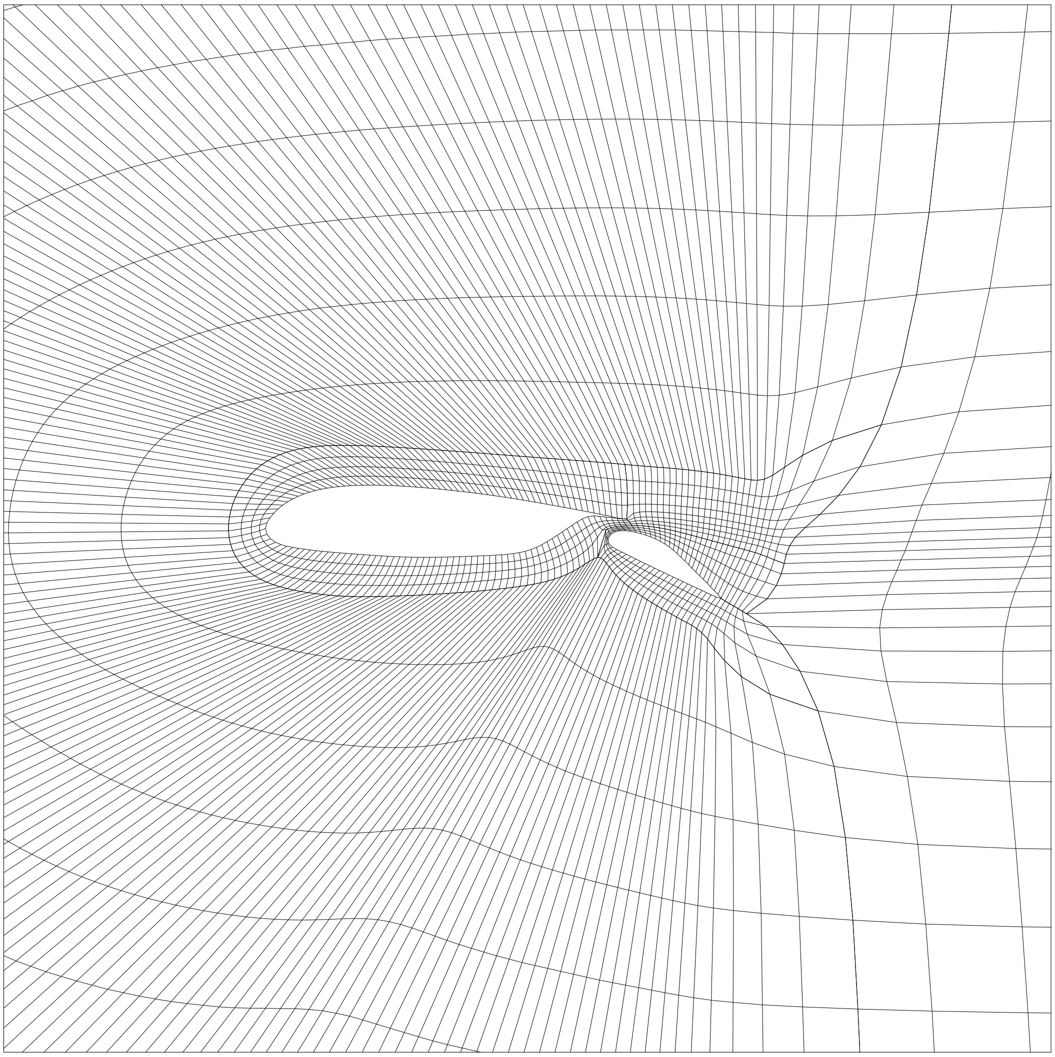

Figure 1: Multi block mesh around a double profile

state is reached.

3.3 Euler's Equations

At high speed flows, the Reynold's numbers become so large, that the viscous terms may be neglected. The compressible Navier-Stokes equations are reduced to the simpler Euler equations. In a 2-dimensional Cartesian coordinate system, the *unsteady Euler equations* can be expressed in conservative form as

$$\frac{\partial W}{\partial t} + \frac{\partial f_q}{\partial x} + \frac{\partial g_q}{\partial y} = 0 \tag{1}$$

where the state vector W is

$$W = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E_{tot} \end{pmatrix} \tag{2}$$

where ρ is the density, u and v are the velocity components, and E_{tot} is the total energy.

The convective fluxes are defined as

$$f_q = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ u(\rho E_{tot} + p) \end{pmatrix} \quad g_q = \begin{pmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ v(\rho E_{tot} + p) \end{pmatrix} \tag{3}$$

where p is the pressure. To close the system of equations, the pressure p must be related to the state vector W. This relation depends on the model used to describe the thermodynamic properties of the gas. For a perfect gas this relation is

$$p = \rho e(\gamma - 1) \tag{4}$$

where e is the internal energy. The internal and total energy are related by

$$e = E_{tot} - \frac{1}{2}(u^2 + v^2) \tag{5}$$

whereby the system of equations is closed. For further details, see [1].

3.4 Numerical Method

The *finite volume method* with a central difference is used for the spatial discretization of the Euler equations, yielding second order accuracy in space for a smooth mesh. To capture shocks properly and to guarantee that no non-physical shocks which violate the entropy condition appear, a second order artificial viscosity term is added. A second numerical problem related to the selected discretization method is caused by the fact that central difference schemes permit odd/even oscillations. This means that a spurious solution in the form of, for example, a sine wave will be undamped. This problem is solved by adding a fourth order artificial dissipation term to the scheme. Since the Euler equations are non-dissipative by themselves it is required that the added artificial dissipation is as small as possible.

Equation 1 is integrated over a domain Ω with the area element dS, edge element ds and outward directed unit surface normal n.

$$\int_\Omega \frac{\partial W}{\partial t} dS + \int_\Omega divH dS = 0 \tag{6}$$

where $H = (f_q, g_q)$ is the flux tensor. The divergence theorem of Gauss yields

$$\int_\Omega \frac{\partial W}{\partial t} dS + \oint_{\partial\Omega} H \cdot nds = 0 \tag{7}$$

Suppose that Ω is a quadrilateral cell i, j and consider $W_{i,j}$ as an approximation to the average value of W over this cell. Hence equation 7 can be approximated as

$$\frac{d}{dt}(S_{i,j}W_{i,j}) + Q_{i,j} = 0 \tag{8}$$

where $S_{i,j}$ is the cell area and $Q_{i,j}$ is the net flux leaving and entering the cell.

$$Q_{i,j} = h_{i+1/2,j} - h_{i-1/2,j} + h_{i,j+1/2} - h_{i,j-1/2} \tag{9}$$

Equation 9 can be interpreted as the inflow subtracted by the outflow of the flux tensor in each direction.

3.4.1 Time Integration

Equation 8 can be written as

$$\frac{d}{dt}(S_{i,j}W_{i,j}) + R_{i,j} = 0 \tag{10}$$

where $R_{i,j}$ is the residual, consisting of the net flux together with the added dissipation terms. When the volume of the cells are constant, 10 can be evaluated as

$$\frac{d}{dt}(W_{i,j}) + \frac{1}{S_{i,j}} R_{i,j} = 0 \tag{11}$$

The main interest here is in the steady state solution. Equation 11 is integrated in time, using an explicit multistage Runge-Kutta time-stepping scheme, until

$$\frac{d}{dt}(W_{i,j}) = 0 \tag{12}$$

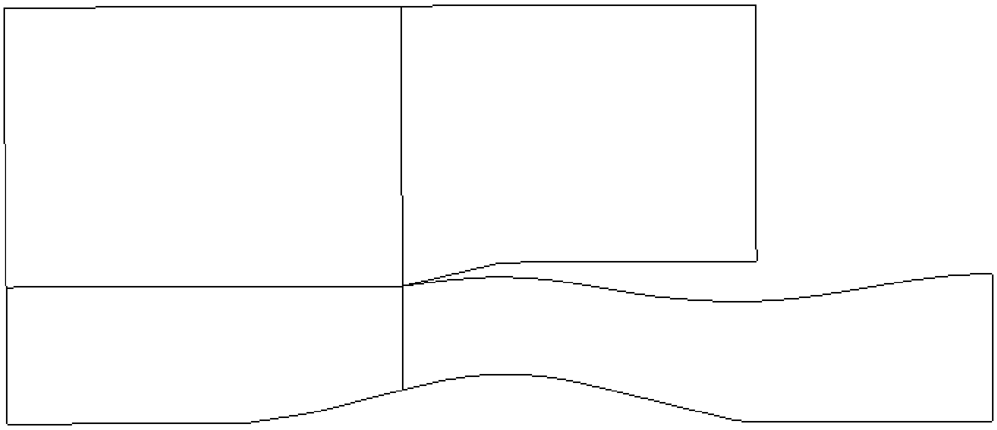

Figure 2: The block partition of the snecma case

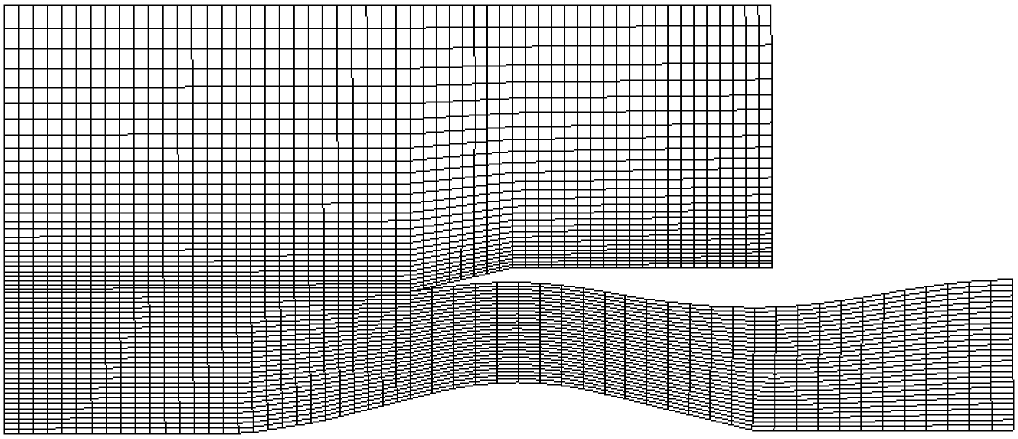

Figure 3: The grid of the snecma case

3.5 Domain Decomposition

The computational domain is split into blocks as in figure 2. Each block is then divided into smaller sections by a mesh. See figure 3. The decomposition gives the following advantages compared to a single block solver.

It is usually necessary to make the computational mesh finer around areas where the geometry is complicated to get sufficient accuracy in the solution. By domain decomposition, it becomes much easier to control the density of the mesh.

There are well established methods for generating meshes for simpler kinds of geometries. By splitting the computational domain into blocks, a higher degree of freedom can be obtained when each block is independently meshed with its own curvilinear coordinate system.

The multi block method can also have computational advantages. It gives a natural way for parallelization by treating the blocks as separate tasks. The amount of data that has to be communicated between the different blocks is reasonably small, so it is often possible to reach a high level of efficiency.

3.6 Boundary Conditions

Here, we distinguish between *external boundary conditions* and *internal boundary conditions*. The external boundary conditions are those that occur due to the splitting of the domain into several blocks. At the border between two neighboring blocks, the state has to be communicated across the border.

The internal boundary conditions are relevant when considering boundaries of physical nature.

The size of the physical domain to be calculated is normally infinite, while the computational domain is limited. Therefore an artificial *far field* boundary has to be introduced. Important is, for example, that outgoing waves are not reflected back again, as this would disturb the convergence. Here the Riemann invariants are used for this purpose. The characteristics are evaluated for a one dimensional flow normal to the boundary. For this purpose, the three first Riemann invariants for the free stream can be derived, one for each eigenvalue of the flux Jacobian in the normal direction given below

$$\begin{array}{ll} \lambda_1 = q_n & R_1 = q_t \\ \lambda_2 = q_n - c & R_2 = q_n - \frac{2c}{\gamma-1} \\ \lambda_3 = q_n + c & R_3 = q_n + \frac{2c}{\gamma-1} \end{array} \tag{13}$$

where q_n is the normal component, q_t the tangent component of the velocity and c the speed of sound. The entropy s, which is the fourth Riemann invariant obtained from the energy equation, where $\frac{Ds}{Dt}$ is constant along a streamline, corresponds to the first eigenvalue in relation 13. The sign of each of the three eigenvalues in 13 determines if the information should be taken from the interior computational domain or be specified as free stream values. This could be performed by first checking the sign of the eigenvalues λ_i, if positive linear extrapolation from the interior is used to form the corresponding R_i, else R_i is formed using specified free stream values. From these Riemann invariants the actual normal velocity and speed of sound can be formed by

$$\begin{aligned} q_n &= \tfrac{1}{2}(R_3 + R_2) \\ c &= \tfrac{\gamma-1}{4}(R_3 - R_2) \end{aligned} \tag{14}$$

The state in the far-field is uniquely defined by q_t, q_n, c and s. If the flow is supersonic in the far-field, hence all eigenvalues are negative, all the flow quantities are specified at an inflow boundary, and they are extrapolated from the interior at an outflow boundary. For the fourth order dissipation the first difference of the state variables is extrapolated by

$$\bar{D}_{-1/2} = \bar{D}_{1/2}$$

representing an zeroth order extrapolation. This extrapolated first difference gives a second difference

$$\bar{E}_0 = \bar{D}_{1/2} - \bar{D}_{-1/2} = 0$$

that in turn gives the third difference as

$$\bar{F}_{1/2} = \bar{E}_1 - \bar{E}_0 = \bar{E}_1 = W_2 - 2W_1 + W_0$$

giving the necessary definition of the numerical dissipation at the far-field.

3.6.1 Solid Surface

The internal boundary conditions are also relevant where the fluid is in contact with a physical body, like the profile investigated or a surrounding wall. Here, the condition is that the normal component of the velocity must vanish, i.e. $(u, v) \cdot n = 0$. From this it is easy to see that the only contribution to the convective flux at the boundary comes from the pressure. Thus it is sufficient to determine the pressure on the boundary. Here the pressure on the surface is found by linear extrapolation from the interior.

3.6.2 Inflow and Outflow

In this work the flow is assumed to be supersonic.

At the inflow all the state variables are fixed at their free stream values (Dirichlet condition) since it is assumed that the inflow is supersonic and thus all characteristics are entering the domain.

At the outflow all characteristics are entering the domain (supersonic outflow) and therefore the state variables are extrapolated from the interior to the exterior ‘ghost points’.

4 Approach

4.1 The Particular Application

The particular application chosen was one for solving Euler and Navier-Stokes equations with the *multi block* method. The domain is divided into several blocks which are solved separately with common boundary values communicated. The program cmb (Concurrent Multi Block Solver) [1], developed by Magnus Bergman, is a Fortran implementation of the solver. It has been tested on several different test cases and on various platforms. The division of the domain into several blocks gives an additional possibility for parallel computation.

The solving consists of a Runge-Kutta march in time, where each point in the domain is updated according to neighboring points, through the Navier-Stokes equations, and boundary conditions.

The solver reads the following files when started: `CASE`, with parameter settings defining physical quantities, `MESH`, which contains a description of how the space is divided into finite volumes, and `BLOCKDATA`, which defines the block interconnectivity.

4.2 Choice of Part to Model in ObjectMath

The task was to make an integration between a CFD solver in Fortran, which was already written and functioning, and ObjectMath. Of interest was to see, how this integration could be done, what modifications to the solver would be needed and if the performance of the application would be affected.

The solver is built around its data structures and its relevant parts are closely coupled to them. This does impose certain limitations of the freedom in expressing the equations in ObjectMath. Some central parts of the solver were to be replaced with calls to functions defined in Mathematica. Their functionality would be the same and they would work on the same data structures, representing states of different entities in the single points of a grid. The functions were defined in a very similar manner in ObjectMath. Then, ObjectMath's facilities for generating code was used to convert the functions and expressions to C++ code. In this process, classes in ObjectMath are represented by standard C++ classes and the functions within the ObjectMath classes by class methods in C++.

4.3 M++ Matrix Class Library

ObjectMath's code generator does generate scalar C++ types such as `int`, `float` and `double`. It does *not*, however, generate arbitrary C++ arrays. The solver, on the other hand, works with arrays of typically 2, 3 and 4 dimensions. In an effort to make the solver more efficient, by significantly reducing the communication between the solver and the generated code, there was a need to transfer and manipulate whole arrays instead of single array elements. There was also a hope that it would enable better structured code. ObjectMath fortunately has a means of dealing with arrays. On the ObjectMath side, they are represented as Mathematica arrays, i.e. lists. The code generator produces code where the arrays are represented as M++ array types. M++ is a C++ library for matrices with associated operations and dynamic memory handling. It supports a wide range of numerical operations on matrices. For this application, however, it was merely used as a means to conduct the communication between the representation of the solver and that of ObjectMath.

A more recent version of ObjectMath generates High Performance Fortran (Fortran 90), but currently only for ODE equations, thus it was not yet possible to use it for the application described in this report. When ObjectMath will be equipped with a general Fortran 90 generator, the code will be more easily integrated with the solver. The syntax and semantics of arrays in ObjectMath are the same in the two cases, though.

5 Results

5.1 Difficulties

Direct transfer of variables of C++ type is limited to scalars, with exception for a few special types of constant size, 1x3 and 3x3 vectors. Larger structures like arrays are dealt with in another way. ObjectMath handles arrays by the data type `doubleArray` from the M++ matrix class library. The code produced is linked with M++. In ObjectMath, the code is represented as lists, which is Mathematica's usual way to deal with arrays.

Sending scalars to ObjectMath and doing all the looping in the Fortran solver introduced some overhead. By adding the pressure there was an introduced delay of about six to nine percent.

The reason was the extra function call in each point, in each time step. For the boundary conditions it would mean even greater disadvantage, because, even the simpler kinds of boundary conditions contains more than one loop construction and thus need to be split up into just as many different functions. This fragmentation of the code makes it non-intuitive, badly structured and hard to read and debug. This alone is probably more than enough to discard this solution.

The solution to this dilemma was to use the M++ class library to represent higher dimension types.

The matrices in question could then be accessed in the same way that Mathematica normally accesses higher dimension structures. ObjectMath has some support for the M++ data type `doubleArray`

Another circumstance that caused problems was that the documentation regarding ObjectMath's code generation capabilities was old and partly inaccurate.

5.2 Performance

Compared to running the Fortran solver alone, some overhead is introduced. Functions evaluated at every point in the grid, such as the pressure equation, cost much more than the boundary conditions, which are only called at the edges of the grid. The overhead is caused by differences between the generated code and the original code, especially when accessing the arrays through the `doubleArray` classes. By using `fastDoubleArray` instead, which has a faster access mechanism, the times were improved.

More interesting performance results will be obtained by parallelization of the pressure equation, but that will need the data-parallel capacity of Fortran 90, which was not available during the part of the project that this report covers.

5.3 Performance Results

Versions with different degrees and types of integration between the Fortran solver and the ObjectMath model have been compiled and evaluated. The performance results should be compared with that of the unaltered solver, *cmbs*. The different versions are described below:

cmbs This is the original version, with no interaction at all with ObjectMath generated code.

cmbs scalar In this version, the pressure function is called for each point, and the values are transferred as scalars, without M++ data structures, to ObjectMath.

cmbs Fp The data for all the points are transferred in a single function call. The M++ data type `fastDoubleArray` is used here.

cmbs Fdew Here, the boundary conditions dirichlet, linear extrapolation and wall are integrated.

cmbs Fpdew Both the pressure equation and the boundary conditions.

cmbs pdew Same as above, but using the M++ type `doubleArray`, as generated by ObjectMath, instead of `fastDoubleArray`.

The performance has been measured on three different problems, called snecma, wedge and naca, executed on a Sun SPARCstation 5. The workstation wasn't loaded with any other time consuming tasks at the time of the measurements. The times listed below are the CPU times, averaged over two runs. The variance is 0.0015 for the snecma and wedge case and 0.020 for the naca case.

version	*execution times*
1 cmbs	98.25
2 cmbs scalar	104.05
3 cmbs pdew	108.10
4 cmbs Fp	106.20
5 cmbs Fdew	99.15
6 cmbs Fpdew	106.55

Results from snecma case.

version	*execution times*
1 cmbs	18.05
2 cmbs scalar	19.15
3 cmbs pdew	20.35
4 cmbs Fp	19.90
5 cmbs Fdew	18.40
6 cmbs Fpdew	19.95

Results from wedge case.

version	*execution times*
1 cmbs	133.80
2 cmbs scalar	146.15
3 cmbs pdew	154.40
4 cmbs Fp	151.35
5 cmbs Fdew	134.85
6 cmbs Fpdew	152.55

Results from naca case.

As expected, the versions where the pressure equation is modeled in ObjectMath show a certain amount of performance degradation, whereas the cost of the boundary conditions is very small. The use of the `fastDoubleArray` type instead of `doubleArray` gives a somewhat better performance. Overall, the performance losses are quite tolerable.

References

[1] Niclas Andersson and Peter Fritzson. Generating parallel code from object oriented mathematical models. In *Proceedings of the Fifth ACM SIGPLAN Symposium om Principles and Practice on Parallel Programming*, pages 48–57, July 1995.

[2] Carl Magnus Bergman. *Development of numerical techniques for inviscid hypersonic flows around re-entry vehicles.* PhD thesis, Institut National Polytechnique Toulouse, 1990. Thesis no. 241.

[3] Dag Fritzson, Peter Fritzson, Lars Viklund, and Johan Herber. Object-oriented mathematical modeling — applied to machine elements. *Computers & Structures*, 51(3):241–253, 1994.

[4] Roman Maeder. *Programming in Mathematica.* Addison-Wesley Publishing Company, 1991.

[5] M.L. Sawley, F. Perrel, C.M. Bergman, and I. Persson. Control-parallel and data-parallel methodologies for flow calculations using supercomputers. *Supercomputing Review*, 1(4):2–8, November 1992.

[6] Lars Viklund and Peter Fritzson. An object-oriented language for symbolic computation — applied to machine element analysis. In Paul S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 397–405. ACM Press, 1992.

[7] Stephen Wolfram. *Mathematica—A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, 1991.

6 Appendix

A ObjectMath Source Code

This is the code in ObjectMath format, which is essentially Mathematica code with added CLASS and INSTANCE declarations. This is the format maintained by the browser. Each class is displayed in a single buffer in the Emacs window. The functions which are related to the code generation, are part of the the ObjectMath environment and are not from the standard Mathematica library.

```
MODEL cmbs2;
  PACKAGES "Codegen`";

  $RecursionLimit=100000;

  CodeGenInit;

CLASS pressure (* Coordinates: 5, 26 *)
END pressure;

CLASS codeGenerator (* Coordinates: 0, 167 *)
END codeGenerator;

INSTANCE c INHERITS codeGenerator
  cc := (
    Print["cmbs2"];
    CodeGenOpenFile["cmbs2"];
    Print["Code generation for variables"];
    CodeGenVars;
    Print["Code generation for functions"];
    perfect`doGenCode;
    dirichlet`doGenCode;
    wall`doGenCode;
    extrapolation`doGenCode;

    CodeGenCloseFile;
  );
END c;

INSTANCE perfect INHERITS pressure
  DefineFunction[ press,
  {"double", gama,
   "int", im1,
```

```
   "int", im2},
  {"doubleArray", w},
  {"double", qr,
   "int", i,
   "int", j,
   "int", c1,
   "int", c2},

  (* Setting coefficients for indexing *)
  c1 = 132;
  c2 = c1*132;

  For[j:=2, j<=im2+1, j++,
    For[i:=2, i<=im1+1, i++,
      qr := .5 (w[[i+ j*c1+ 1*c2]]^2 + w[[i+ j*c1+ 2*c2]]^2) /
w[[i+ j*c1+ 0*c2]];
      w[[i+ j*c1+ 4*c2]] := (gama - 1.) (w[[i+ j*c1+ 3*c2]] - qr);
    ];
  ];
  w
  ];

  doGenCode := (
    DefGen[ press ];
  );
END perfect;

CLASS boundaryCondition (* Coordinates: 19, -16 *)
END boundaryCondition;

INSTANCE dirichlet INHERITS boundaryCondition (* Coordinates: 0, 83 *)
  (* dirichlet condition *)

  DefineFunction[ dirich,
  {"doubleArray", wtmp,
   "doubleArray", wb,
   "doubleArray", xb,
   "int", ns,
   "int", istart,
   "int", istop},
  {"doubleArray", bw,
   "doubleArray", bv,
   "doubleArray", bd,
   "doubleArray", bx},
```

```
  {"int", l,
   "int", io,
   "int", c1,
   "int", c2,
   "int", c3,
   "int", c4},

  (* Indexing coefficients for state arrays *)

  c1 = 132;  (* wb, xb, bw, bv, bd, bx *)
  c2 = c1 * 5; (* wb, bw, bv, bd, bx *)
  c3 = c2 * 2; (* bd *)
  c4 = c1 * 2; (* bx *)
  For[l:=0, l<5, l++,
    For[io:=istart+1, io <= istop+1, io++,
      bw[[io +l*c1 +ns*c2]] :=    wtmp[[l]];
      bv[[io +l*c1 +ns*c2]] :=    bw[[io +l*c1 +ns*c2]];
      bd[[io +l*c1 +0*c2 +ns*c3]] :=
  bw[[io +l*c1 +ns*c2]];
      bd[[io +l*c1 +1*c2 +ns*c3]] :=
  2. bw[[io +l*c1 +ns*c2]] - wb[[io +l*c1 +0*c2]];
    ];
  ];

  For[io:=istart+1, io <= istop+2, io++,
    bx[[io +0*c1 +ns*c4]] := xb[[io +0*c1]];
    bx[[io +1*c1 +ns*c4]] := xb[[io +1*c1]];
  ];

  {bw, bv, bd, bx}
  ];

  doGenCode := (
    DefGen[ dirich ];
  );
END dirichlet;

INSTANCE wall INHERITS boundaryCondition
  (* wall boundary condition *)

  DefineFunction[ wall,
  {"doubleArray", wb,
   "doubleArray", xb,
   "double", twall,
```

```
 "int", ns,
 "int", ins,
 "int", istart,
 "int", istop},
{"doubleArray", bw,
 "doubleArray", bv,
 "doubleArray", bd,
 "doubleArray", bx},
{"int", io,
 "int", l,
 "int", c1,
 "int", c2,
 "int", c3,
 "int", c4},

(* Indexing coefficients for state arrays *)

c1 = 132;  (* wb, xb, bw, bv, bd, bx *)
c2 = c1 * 5; (* wb, bw, bv, bd, bx *)
c3 = c2 * 2; (* bd *)
c4 = c1 * 2; (* bx *)

For[io:=istart+1, io <= istop+1, io++,
  bw[[io +0*c1 +ns*c2]] :=  wb[[io +0*c1 +0*c2]];
  bw[[io +1*c1 +ns*c2]] := -wb[[io +1*c1 +0*c2]];
  bw[[io +2*c1 +ns*c2]] := -wb[[io +2*c1 +0*c2]];
  bw[[io +3*c1 +ns*c2]] := -wb[[io +3*c1 +0*c2]];
  bw[[io +4*c1 +ns*c2]] :=  wb[[io +4*c1 +0*c2]];

  bv[[io +0*c1 +ns*c2]] := wb[[io +0*c1 +0*c2]];
  bv[[io +1*c1 +ns*c2]] := 0.;
  bv[[io +2*c1 +ns*c2]] := 0.;
  bv[[io +3*c1 +ns*c2]] := 0.;
  bv[[io +4*c1 +ns*c2]] := wb[[io +4*c1 +0*c2]];
];
    (* iso-thermal wall for navier-stokes equations *)

If[ins == 1 && twall > 0.0,
  For[io := istart+1, io<= istop+1, io++,
    bw[[io +0*c1 +ns*c2]] := wb[[io +4*c1 +0*c2]] / twall;
    bv[[io +0*c1 +ns*c2]] := wb[[io +4*c1 +0*c2]] / twall;
  ];
];
```

```
  For[l := 0, l<5, l++,
    For[io := istart+1, io <= istop+1, io++,
      bd[io +l*c1 +0*c2 +ns*c3] :=
  2.0 wb[io +l*c1 +0*c2] - wb[io +l*c1 +1*c2];
      bd[io +l*c1 +1*c2 +ns*c3] :=
  3.0 wb[io +l*c1 +0*c2] - 2.0 wb[io +l*c1 +1*c2];
    ];
  ];

  For[l := 0, l<2, l++,
    For[io := istart+1, io <= istop + 2, io++,
      bx[[io +l*c1 +ns*c4]] := xb[[io +l*c1]];
    ];
  ];

  {bw, bv, bd, bx}
  ];

  doGenCode := (
    DefGen[ wall ];
  );
END wall;

INSTANCE extrapolation INHERITS boundaryCondition
  (* linear extrapolation *)

  DefineFunction[ extrap,
  {"doubleArray", wb,
   "doubleArray", xb,
   "int", ns,
   "int", istop,
   "int", istart},
  {"doubleArray", bw,
   "doubleArray", bv,
   "doubleArray", bd,
   "doubleArray", bx},
  {"int", l,
   "int", io,
   "int", c1,
   "int", c2,
   "int", c3,
   "int", c4},

  (* Indexing coefficients for state arrays *)
```

```
  c1 = 132;  (* wb, xb, bw, bv, bd, bx *)
  c2 = c1 * 5; (* wb, bw, bv, bd, bx *)
  c3 = c2 * 2; (* bd *)
  c4 = c1 * 2; (* bx *)

  For[l := 0, l < 5, l++,
    For[io := istart+1, io <= istop+1, io++,
      bw[[io +l*c1 +ns*c2]] :=
  2.0 wb[[io +l*c1 +0*c2]] - wb[[io +l*c1 +1*c2]];
      bv[[io +l*c1 +ns*c2]] := bw[[io +l*c1 +ns*c2]];
      bd[[io +l*c1 +0*c2 +ns*c3]] :=
  bw[[io +l*c1 +ns*c2]];
      bd[[io +l*c1 +1*c2 +ns*c3]] :=
  2.0 bw[[io +l*c1 +ns*c2]] - wb[[io +l*c1 +0*c2]];
    ];
  ];

  For[l := 0, l < 2, l++,
    For[io := istart+1, io <= istop+2, io++,
      bx[[io +l*c1 +ns*c4]] := xb[[io +l*c1]]
    ];
  ];

  {bw, bv, bd, bx}
  ];

  doGenCode := (
    DefGen[ extrap ];
  );
END extrapolation;
```

B Modified C++ Code

Running the Mathematica to C++ translator is done by calling `DefGen` with appropriate arguments. Each DefineFunction will be represented by a C++ class method. Arguments which are of higher dimension will be handled by the M++ types. By using the M++ type `doubleArray` in the `DefineFunction` declaration, the function will be able to accept and perform operations on a matrix of that kind. To make the translated code callable from Fortran, interface functions have to be appended to the code. They accept a call from the Fortran solver and pass it on to the functions defined in ObjectMath.

```
// This C++ file was automatically generated by ObjectMath
// and modified by Erik Ekholm for integration with CFD solver
#include <darray.h>
#include "cmbs2.h"

void cmbs2::initialize()
{ // Initialize the model

  // Allocate and zero arrays
  allocate_dynamic_arrays();
}

void cmbs2::allocate_dynamic_arrays() {
};

void cmbs2::perfect_press(
  // Input parameters
  const double perfect_gama,
  const int perfect_im1,
  const int perfect_im2,
  // Output parameters
  doubleArray& perfect_w
  )
{
  double perfect_qr;
  int perfect_i;
  int perfect_j;
  int perfect_c1;
  int perfect_c2;

// Body of 'perfect_press'
perfect_c1 = 132;
perfect_c2 = perfect_c1*132;
for(perfect_j = 2; perfect_j <= perfect_im2 + 1; perfect_j++) {
for(perfect_i = 2; perfect_i <= perfect_im1 + 1; perfect_i++) {
perfect_qr = 0.5*(perfect_w(perfect_i + perfect_j*perfect_c1 + 1*perfect_c2)
*perfect_w(perfect_i + perfect_j*perfect_c1 + 1*perfect_c2) +
perfect_w(perfect_i + perfect_j*perfect_c1 + 2*perfect_c2)*
perfect_w(perfect_i + perfect_j*perfect_c1 + 2*perfect_c2))/
perfect_w(perfect_i + perfect_j*perfect_c1 + 0*perfect_c2);
perfect_w(perfect_i + perfect_j*perfect_c1 + 4*perfect_c2) =
(perfect_gama - 1.)*(perfect_w(perfect_i + perfect_j*perfect_c1 +
3*perfect_c2) - perfect_qr);
}
```

```
}

perfect_w = perfect_w;
};  // perfect_press

void cmbs2::dirichlet_dirich(
  // Input parameters
  const doubleArray& dirichlet_wtmp,
  const doubleArray& dirichlet_wb,
  const doubleArray& dirichlet_xb,
  const int dirichlet_ns,
  const int dirichlet_istart,
  const int dirichlet_istop,
  // Output parameters
  doubleArray& dirichlet_bw,
  doubleArray& dirichlet_bv,
  doubleArray& dirichlet_bd,
  doubleArray& dirichlet_bx
  )
{
  int dirichlet_l;
  int dirichlet_io;
  int dirichlet_c1;
  int dirichlet_c2;
  int dirichlet_c3;
  int dirichlet_c4;

// Body of 'dirichlet_dirich'
dirichlet_c1 = 132;
dirichlet_c2 = dirichlet_c1*5;
dirichlet_c3 = dirichlet_c2*2;
dirichlet_c4 = dirichlet_c1*2;
for(dirichlet_l = 0; dirichlet_l < 5; dirichlet_l++) {
for(dirichlet_io = dirichlet_istart + 1; dirichlet_io <= dirichlet_istop + 1;
 dirichlet_io++) {
dirichlet_bw(dirichlet_io + dirichlet_l*dirichlet_c1 + dirichlet_ns*
dirichlet_c2) = dirichlet_wtmp(dirichlet_l);
dirichlet_bv(dirichlet_io + dirichlet_l*dirichlet_c1 + dirichlet_ns*
dirichlet_c2) = dirichlet_bw(dirichlet_io + dirichlet_l*dirichlet_c1 +
dirichlet_ns*dirichlet_c2);
dirichlet_bd(dirichlet_io + dirichlet_l*dirichlet_c1 + 0*dirichlet_c2 +
dirichlet_ns*dirichlet_c3) = dirichlet_bw(dirichlet_io + dirichlet_l*
dirichlet_c1 + dirichlet_ns*dirichlet_c2);
dirichlet_bd(dirichlet_io + dirichlet_l*dirichlet_c1 + 1*dirichlet_c2 +
```

```
dirichlet_ns*dirichlet_c3) = 2.0*dirichlet_bw(dirichlet_io + dirichlet_l*
dirichlet_c1 + dirichlet_ns*dirichlet_c2) - dirichlet_wb(dirichlet_io +
dirichlet_l*dirichlet_c1 + 0*dirichlet_c2);
}
}
for(dirichlet_io = dirichlet_istart + 1; dirichlet_io <= dirichlet_istop + 2;
dirichlet_io++) {
dirichlet_bx(dirichlet_io + 0*dirichlet_c1 + dirichlet_ns*dirichlet_c4) =
dirichlet_xb(dirichlet_io + 0*dirichlet_c1);
dirichlet_bx(dirichlet_io + 1*dirichlet_c1 + dirichlet_ns*dirichlet_c4) =
dirichlet_xb(dirichlet_io + 1*dirichlet_c1);
}

dirichlet_bw = dirichlet_bw;
dirichlet_bv = dirichlet_bv;
dirichlet_bd = dirichlet_bd;
dirichlet_bx = dirichlet_bx;
};  // dirichlet_dirich

void cmbs2::wall(
  // Input parameters
  const doubleArray& wall_wb,
  const doubleArray& wall_xb,
  const double wall_twall,
  const int wall_ns,
  const int wall_ins,
  const int wall_istart,
  const int wall_istop,
  // Output parameters
  doubleArray& wall_bw,
  doubleArray& wall_bv,
  doubleArray& wall_bd,
  doubleArray& wall_bx
  )
{
  int wall_io;
  int wall_l;
  int wall_c1;
  int wall_c2;
  int wall_c3;
  int wall_c4;

// Body of 'wall'
wall_c1 = 132;
```

```
wall_c2 = wall_c1*5;
wall_c3 = wall_c2*2;
wall_c4 = wall_c1*2;
for(wall_io = wall_istart + 1; wall_io <= wall_istop + 1; wall_io++) {
wall_bw(wall_io + 0*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
0*wall_c1 + 0*wall_c2);
wall_bw(wall_io + 1*wall_c1 + wall_ns*wall_c2) = -wall_wb(wall_io +
1*wall_c1 + 0*wall_c2);
wall_bw(wall_io + 2*wall_c1 + wall_ns*wall_c2) = -wall_wb(wall_io +
2*wall_c1 + 0*wall_c2);
wall_bw(wall_io + 3*wall_c1 + wall_ns*wall_c2) = -wall_wb(wall_io +
3*wall_c1 + 0*wall_c2);
wall_bw(wall_io + 4*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
4*wall_c1 + 0*wall_c2);
wall_bv(wall_io + 0*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
0*wall_c1 + 0*wall_c2);
wall_bv(wall_io + 1*wall_c1 + wall_ns*wall_c2) = 0.;
wall_bv(wall_io + 2*wall_c1 + wall_ns*wall_c2) = 0.;
wall_bv(wall_io + 3*wall_c1 + wall_ns*wall_c2) = 0.;
wall_bv(wall_io + 4*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
4*wall_c1 + 0*wall_c2);
}
if(wall_ins == 1 && wall_twall > 0.) {
for(wall_io = wall_istart + 1; wall_io <= wall_istop + 1; wall_io++) {
wall_bw(wall_io + 0*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
4*wall_c1 + 0*wall_c2)/wall_twall;
wall_bv(wall_io + 0*wall_c1 + wall_ns*wall_c2) = wall_wb(wall_io +
4*wall_c1 + 0*wall_c2)/wall_twall;
}
}
for(wall_l = 0; wall_l < 5; wall_l++) {
for(wall_io = wall_istart + 1; wall_io <= wall_istop + 1; wall_io++) {
wall_bd(wall_io + wall_l*wall_c1 + 0*wall_c2 + wall_ns*wall_c3) =
2.*wall_wb(wall_io + wall_l*wall_c1 + 0*wall_c2) - wall_wb(wall_io +
wall_l*wall_c1 + 1*wall_c2);
wall_bd(wall_io + wall_l*wall_c1 + 1*wall_c2 + wall_ns*wall_c3) =
3.*wall_wb(wall_io + wall_l*wall_c1 + 0*wall_c2) - 2.*wall_wb(wall_io
+ wall_l*wall_c1 + 1*wall_c2);
}
}
for(wall_l = 0; wall_l < 2; wall_l++) {
for(wall_io = wall_istart + 1; wall_io <= wall_istop + 2; wall_io++) {
wall_bx(wall_io + wall_l*wall_c1 + wall_ns*wall_c4) = wall_xb(wall_io
+ wall_l*wall_c1);
```

```
}
}

wall_bw = wall_bw;
wall_bv = wall_bv;
wall_bd = wall_bd;
wall_bx = wall_bx;
};  // wall

void cmbs2::extrapolation_extrap(
  // Input parameters
  const doubleArray& extrapolation_wb,
  const doubleArray& extrapolation_xb,
  const int extrapolation_ns,
  const int extrapolation_istop,
  const int extrapolation_istart,
  // Output parameters
  doubleArray& extrapolation_bw,
  doubleArray& extrapolation_bv,
  doubleArray& extrapolation_bd,
  doubleArray& extrapolation_bx
  )
{
  int extrapolation_l;
  int extrapolation_io;
  int extrapolation_c1;
  int extrapolation_c2;
  int extrapolation_c3;
  int extrapolation_c4;

// Body of 'extrapolation_extrap'
extrapolation_c1 = 132;
extrapolation_c2 = extrapolation_c1*5;
extrapolation_c3 = extrapolation_c2*2;
extrapolation_c4 = extrapolation_c1*2;
for(extrapolation_l = 0; extrapolation_l < 5; extrapolation_l++) {
for(extrapolation_io = extrapolation_istart + 1; extrapolation_io <=
extrapolation_istop + 1; extrapolation_io++) {
extrapolation_bw(extrapolation_io + extrapolation_l*extrapolation_c1 +
extrapolation_ns*extrapolation_c2) = 2.*extrapolation_wb(extrapolation_io +
extrapolation_l*extrapolation_c1 + 0*extrapolation_c2) - extrapolation_wb(
extrapolation_io + extrapolation_l*extrapolation_c1 + 1*extrapolation_c2);
extrapolation_bv(extrapolation_io + extrapolation_l*extrapolation_c1 +
extrapolation_ns*extrapolation_c2) = extrapolation_bw(extrapolation_io +
```

```
extrapolation_l*extrapolation_c1 + extrapolation_ns*extrapolation_c2);
extrapolation_bd(extrapolation_io + extrapolation_l*extrapolation_c1 +
0*extrapolation_c2 + extrapolation_ns*extrapolation_c3) =
extrapolation_bw(extrapolation_io + extrapolation_l*extrapolation_c1 +
extrapolation_ns*extrapolation_c2);
extrapolation_bd(extrapolation_io + extrapolation_l*extrapolation_c1 +
1*extrapolation_c2 + extrapolation_ns*extrapolation_c3) =
2.*extrapolation_bw(extrapolation_io + extrapolation_l*extrapolation_c1 +
extrapolation_ns*extrapolation_c2) - extrapolation_wb(extrapolation_io +
extrapolation_l*extrapolation_c1 + 0*extrapolation_c2);
}
}
for(extrapolation_l = 0; extrapolation_l < 2; extrapolation_l++) {
for(extrapolation_io = extrapolation_istart + 1; extrapolation_io <=
extrapolation_istop + 2; extrapolation_io++) {
extrapolation_bx(extrapolation_io + extrapolation_l*extrapolation_c1 +
extrapolation_ns*extrapolation_c4) = extrapolation_xb(extrapolation_io +
extrapolation_l*extrapolation_c1);
}
}

extrapolation_bw = extrapolation_bw;
extrapolation_bv = extrapolation_bv;
extrapolation_bd = extrapolation_bd;
extrapolation_bx = extrapolation_bx;
};  // extrapolation_extrap


/********************************************************
*  Interface functions - appended to the generated file    *
********************************************************/

cmbs2 *cmbs = new cmbs2;

// -----  perfec  ---------------------------------------------
extern "C"{

  void perfec_(const double *gama,
       const int *im1,
       const int *im2,
               double *w)
    {
      doubleArray wM(&w[0], 68*130*5);
      cmbs->perfect_press(*gama,
```

```
  *im1,
  *im2,
  wM);
    }
}

// -----  dirich  ---------------------------------------------
extern "C"{

  void dirich_(
    // Input parameters
       double *wtmp,
       double *wb,
       double *xb,
       const int *ns,
       const int *istart,
       const int *istop,
    // Output parameters
       double *bw,
       double *bv,
       double *bd,
       double *bx)
  {
    doubleArray wtmpM(&wtmp[0], 20);
    doubleArray wbM(&wb[0],130*5*2);
    doubleArray xbM(&xb[0],130*2);
    doubleArray bwM(&bw[0],130*5*4);
    doubleArray bvM(&bv[0],130*5*4);
    doubleArray bdM(&bd[0],130*5*2*4);
    doubleArray bxM(&bx[0],130*2*4);
    cmbs->dirichlet_dirich(wtmpM,
   wbM,
   xbM,
   *ns,
   *istart,
   *istop,
   bwM,
   bvM,
   bdM,
   bxM);
  }
}

// -----  extrap  ---------------------------------------------
```

```
extern "C"{
  void extrap_(
  // Input parameters
       double *wb,
       double *xb,
       const int *ns,
       const int *istop,
       const int *istart,
  // Output parameters
       double *bw,
       double *bv,
       double *bd,
       double *bx
       )
  {
    doubleArray wbM(&wb[0],130*5*2);
    doubleArray xbM(&xb[0],130*2);
    doubleArray bwM(&bw[0],130*5*4);
    doubleArray bvM(&bv[0],130*5*4);
    doubleArray bdM(&bd[0],130*5*2*4);
    doubleArray bxM(&bx[0],130*2*4);
    cmbs->extrapolation_extrap(wbM,
       xbM,
       *ns,
       *istop,
       *istart,
       bwM,
       bvM,
       bdM,
       bxM);
  }
}

// -----  wall  ---------------------------------------------
extern "C"{

  void wall_(
   // Input parameters
     double *wb,
     double *xb,
     const double *twall,
     const int *ns,
     const int *ins,
     const int *istart,
```

```
    const int *istop,
  // Output parameters
    double *bw,
    double *bv,
    double *bd,
    double *bx
    )
 {
   doubleArray wbM(&wb[0], 130*5*2);
   doubleArray xbM(&xb[0], 130*2);
   doubleArray bwM(&bw[0], 130*5*4);
   doubleArray bvM(&bv[0], 130*5*4);
   doubleArray bdM(&bd[0], 130*5*2*4);
   doubleArray bxM(&bx[0], 130*2*4);
   cmbs->wall(wbM,
      xbM,
      *twall,
      *ns,
      *ins,
      *istart,
      *istop,
      bwM,
      bvM,
      bdM,
      bxM);
 }
}
```

C Relevant Parts of the PDE Solver (Fortran)

The PDE solver originally consists of one main program, `cmbs.f`, and 24 subroutines. Here, the parts that are relevant for the interaction with the functions defined in Mathematica are shown. `bconin.f` contains code for the different kinds of internal boundary conditions. `tmarch.f` advances the state variable one time step, here the pressure formula resides.

C.1 cmbs.f

```
      program cmb
c
c     by c.m. bergman
c
c     concurrent multi block program
c
```

```
c     solving the euler or navier stokes equations
c
c     type of boundary conditions (bcs) :
c
c     nbpg(1,np,ns,nb) = 1 dirichlet using freestrem values
c                        2 1:st order extr. from the interior
c                        3 solid wall 1:st order extr. of pressure
c                        4 pick from some side in the same block
c                        5 pick from some side in another  block
c                        6 1d riemann invariants
c                        7 inflow for channel
c                        8 outflow for channel
c
c  files opened during execution :
c
c     CASE       defines the case
c     BLOCKDATA  defines bcs and block interconnectivity
c     MESH       contains the mesh in ascii format
c     BINMESH    contains the mesh in binary format
c     SOLUTION   contains the solution in ascii format
c     BINSOL     contains the solution in binary format
c
c  global arrays :
c
c     wg(nb,1,i,j) = density
c     wg(nb,2,i,j) = momentum in x direction
c     wg(nb,3,i,j) = momentum in y direction
c     wg(nb,4,i,j) = total energy
c     wg(nb,5,i,j) = pressure
c
c     xg(nb,1,i,j) = x coordinate
c     xg(nb,2,i,j) = y coordinate
c
c  maximum dimensions:
c
c     mb  = number of blocks
c     il  = grid points in i direction in each block
c     jl  = grid points in j direction in each block
c     nsm = number of states
c     nbl = length of list of bcs
c
c  input parameters:
c
c     CFL      =  cfl number for convective part
```

```
c     CFLVIS   =  cfl number for viscous part
c     DKI2     =  second order dissipation coefficient in i-direction
c     DKJ2     =  second order dissipation coefficient in j-direction
c     DK4      =  fourth order dissipation coefficient
c     NMAX     =  maximum number of timesteps
c     TOL      =  convergence criterion (l2-norm of density change < tol)
c     ITAC     =  0  local  time-steps
c                 1  global time-steps
c     IZOOM    =  SG GL windows size
c     IRBIN    =  0  read ascii format mesh file MESH
c                 1  read binary format mesh file BINMESH
c     IWSOL    =  0 do not write solution
c                 1 write solution
c     NWSTEP   =  print convergence check every NWSTEP steps
c     IRR      =  0 start from free-stream conditions
c                 1 start from solution in file BINSOL
c                -1 start from solution in file SOLUTION
c     MSTATE   =  number of states in list
c     NRK      =  number of Runge Kutta stages
c     ARK      =  coefficients of Runge Kutta stages
c     INS      =  0 euler equations solved
c                 1 navier-stokes equations solved
c     DENSITY  =  denisity
c     PRESSURE =  pressure
c     FMACH    =  mach number
c     ALPHA    =  angle of attack
c     GAMA     =  gamma
c     RE       =  reynolds number
c     PR       =  prandtl number
c     TEMP     =  if > 0.0 temperature of the iso-thermal wall
c                 if < 0.0 adiabatic wall
c
c old settings      parameter (mb=16,m2=-1,il=65,i2=il+1,jl=127,j2=jl+1)
      parameter (mb=4,m2=-1,il=129,i2=il+1,jl=129,j2=jl+1)
c
      parameter (n2=max(i2,j2))
c
      parameter (nsm=9,nbl=100,mobj=100)
c
      real*8 wg(m2:i2,m2:j2,5,mb),xg(m2:i2,m2:j2,2,mb)
c
      real*8 vol(m2:i2,m2:j2    ),w1 (m2:i2,m2:j2,4)
     +,    x  (m2:i2,m2:j2,2  ),w  (m2:i2,m2:j2,5)
     +,    lam(m2:i2,m2:j2,2  ),fnd(m2:i2,m2:j2,4)
```

```
     +,    f  (m2:i2,m2:j2,4,2),dov(m2:i2,m2:j2  )
     +,    s  (m2:i2,m2:j2,2,2),dtl(m2:i2,m2:j2  )
c
      real*8 rmu(m2:i2,m2:j2  )
     +,    re2(m2:i2,m2:j2  ),re4(m2:i2,m2:j2    )
c
      real*8 fs(m2:i2,m2:j2,6),fg(m2:i2,m2:j2,6)
c
      real*8 bd(m2:n2,5,0:1,4),bw(m2:n2,5,4),bx(m2:n2,2,4)
     +,    bv(m2:n2,5,4)
c
      real*8 wb(m2:n2,5,0:1),xb(m2:n2,2)
c
      real*8 dt(2),con(20),dk(3),rms(3),ark(10)
c
      real*8 w0g(20,nsm),fmach(nsm),alpha(nsm),gama(nsm)
     &,    re(nsm),pr(nsm),temp(nsm)
c
      real*8 ustime1(2),ustime2(2)
c
#ifdef DO_GRAPH
c
c     work array for graphics
c
      real*8 cm(m2:i2,m2:j2,2,mb)
c
#endif
c
      integer lobj(mobj)
c
      integer img(-2:2,2,mb),nbpg(8,10,4,mb),nbdg(4,mb)
c
      integer icon(2),nstate(3,mb),info(nbl)
c
c     some open statements
c
      open(1 ,file='CASE',status='unknown')
      open(2 ,file='BLOCKDATA',status='unknown')
      open(4 ,file='SOLUTION',status='unknown')
      open(7 ,file='BINSOL',form='unformatted',status='unknown')
      open(9 ,file='MESH',status='unknown')
      open(19,file='BINMESH',form='unformatted',status='unknown')
c
c     number of blocks on grid and data file
```

```
c
      read(2,*)
      read(2,*)
      read(2,*) nblock
c
c     maximum indexes and orientation for the blocks
c
      read(2,*)
      do 100 nb=1,nblock
        read(2,*) img(0,1,nb),img(0,2,nb),nstate(3,nb)
100   continue
c
c     read input parameters
c
      read(1,*)
      read(1,*) cfl,cflvis,dk(1),dk(2),dk(3)
c
      read(1,*)
      read(1,*) nmax,tol,itac,izoom,irbin
c
      read(1,*)
      read(1,*) iwsol,nwstep,irr,mstate
c
c     runge-kutta stage coefficients
c
      read(1,*)
      read(1,*) nrk,(ark(i),i=1,nrk)
c
c     read list of states
c
      read(1,*)
      do n=1,mstate
        read(1,*) w0g(1,n),w0g(5,n),fmach(n),alpha(n),gama(n)
     +,           re(n),pr(n),temp(n)
      enddo
c
c     should we do graphics or not
c
      dogra = 0
      if( nwstep .lt. 0 ) dogra = 1
      nwstep = abs( nwstep )
c
c     set the freestream values
c
```

```
      do n=1,mstate
        cf       =  sqrt(gama(n)*w0g(5,n)/w0g(1,n))
        pi       =  3.141592e0
        alpha(n) =  alpha(n)*pi/180.e0
        w0g(2,n) =  w0g(1,n)*fmach(n)*cf*cos(alpha(n))
        w0g(3,n) =  w0g(1,n)*fmach(n)*cf*sin(alpha(n))
        qr       = (w0g(2,n)**2 + w0g(3,n)**2)/w0g(1,n)
        ei       =  w0g(5,n)/((gama(n)-1.e0)*w0g(1,n))
        w0g(4,n) =  w0g(1,n)*ei + 0.5e0*qr
      enddo
c
c     copy into w0g
c
      do n=1,mstate
        w0g(6 ,n) = fmach(n)
        w0g(7 ,n) = alpha(n)
        w0g(8 ,n) = gama(n)
        w0g(9 ,n) = re(n)
        w0g(10,n) = pr(n)
        w0g(11,n) = temp(n)
      enddo
c
c     store necessary constans in array con and icon
c
      icon(1) = itac
      icon(2) = nwstep
c
      con(1) = cfl
      con(2) = cflvis
      con(3) = dk(1)
      con(4) = dk(2)
      con(5) = dk(3)
c
      con(6 ) = ark(1)
      con(7 ) = ark(2)
      con(8 ) = ark(3)
      con(9 ) = ark(4)
      con(10) = ark(5)
c
c     read initial states for each block
c
      read(1,*)
      read(1,*) (nstate(1,nb),nb=1,nblock)
c
```

```
c     read equations applied for each block
c
      read(1,*)
      read(1,*) (nstate(2,nb),nb=1,nblock)
c
      if( irbin .eq. 1 ) then
        read(19)  nblgri
      else
        read(9,*) nblgri
      endif
c
c     fill the rest of img
c
      do 110 nb=1,nblock
        do 110 l=1,2
          do 110 lm=-2,2
            img(lm,l,nb)=img(0,l,nb)+lm
110   continue
c
c     read the complete mesh from unit 9
c
      do 120 nb=1,nblock
        call redmsh(irbin,nb,img,xg,mb,i2,j2)
120   continue
c
c  read type of bc for every window on each block side
c
      ib = 0
      do nb=1,nblock
c
c  the number of windows on each side of this block
c
         read(2,*)
         read(2,*) (nbdg(ns,nb), ns=1,4)
c
c  loop over the sides
c
         do ns=1,4
c
c  loop over the number over windows on this side
c
c  read type of bc, first and last index
c
            read(2,*)
```

```
            read(2,*) (nbpg(1,np,ns,nb),np=1,nbdg(ns,nb))
            read(2,*)
            read(2,*) (nbpg(2,np,ns,nb),nbpg(3,np,ns,nb)
     &,                np=1,nbdg(ns,nb))
c
c  one more bc to be read from list?
c
           do np=1,nbdg(ns,nb)
             if( nbpg(1,np,ns,nb) .lt. 0 ) ib = ib + 1
           enddo
         end do
      end do
c
c  loop over the blocks again
c
      read(2,*)
      read(2,*)
c
      do nb=1,nblock
c
         do ns=1,4
c
            do np=1,nbdg(ns,nb)
               if((nbpg(1,np,ns,nb).eq.4) .or.
     &            (nbpg(1,np,ns,nb).eq.5)) then
c
c  block number, side number, index direction, index translation
c
                  read(2,*)
                  read(2,*) nbpg(4,np,ns,nb),nbpg(5,np,ns,nb)
     &,                     nbpg(6,np,ns,nb),nbpg(7,np,ns,nb)
               endif
            end do
         end do
      end do
c
c  read in bc type information
c
      read(2,*)
      read(2,*)
      if( ib .ne. 0 ) then
        read(2,*) (info(i),i=1,ib)
      endif
c
```

```
c  loop over the blocks again
c
      ib = 0
      do nb=1,nblock
        do ns=1,4
          do np=1,nbdg(ns,nb)
c
            if( nbpg(1,np,ns,nb) .lt. 0 ) then
              ib = ib + 1
              nbpg(8,np,ns,nb) = info(ib)
              nbpg(1,np,ns,nb) = abs( nbpg(1,np,ns,nb) )
            else
              nbpg(8,np,ns,nb) = 1
            endif
c
          end do
        end do
      end do
c
c     read old solution for rerun
c
      if( irr  .eq.  1 ) then
c
        read(7)
        do 200 nb=1,nblock
          read(7)
          do 200 j=1,img(-1,2,nb)
            do 200 i=1,img(-1,1,nb)
              read(7) wg(i,j,1,nb),wg(i,j,2,nb),wg(i,j,3,nb)
     +,               wg(i,j,4,nb),wg(i,j,5,nb)
200     continue
c
      else if( irr .eq. -1 ) then
c
        read(4,*)
        do 205 nb=1,nblock
          read(4,*)
          do 205 j=1,img(-1,2,nb)
            read(4,*)
            do 205 i=1,img(-1,1,nb)
              read(4,*) wg(i,j,1,nb),wg(i,j,2,nb),wg(i,j,3,nb)
     +,                 wg(i,j,4,nb),wg(i,j,5,nb)
205     continue
c
```

```
      else
c
c     set the initial condition to the freestream values
c
        do 210 nb=1,nblock
          do 210 l=1,5
            do 210 j=1,img(-1,2,nb)
              do 210 i=1,img(-1,1,nb)
                wg(i,j,l,nb)=w0g(l,nstate(1,nb))
210     continue
c
      endif
c
c     error checking
c
      if (nblock .ne. nblgri) then
        write(*,'(a)') ' Error: nblock not equal to nblgri'
        stop
      end if
c
      if ( ib .ge. nbl ) then
        write(*,'(a)') ' Error: increase nbl'
        stop
      end if
#ifdef DO_GRAPH
c
      if( dogra .eq. 1 ) then
c
c     intialize raster/silicon graphics
c
      call iniras(255,0,izoom,0,izoom,2)
c
c     SGI mouse
c
      call mbound(mb,m2,i2,j2,img,xg,nblock,1)
      call mgrids(mb,m2,i2,j2,img,xg,nblock,3,255)
      lobj(1) = 3
      call mouseman(1,lobj,mobj,1,1)
c
      endif
c
#endif
c
c     timing for alliant
```

```
c
      eti  =etime(ustime1)
c     eti  =secnds(0.0)
c
c     main time integration loop
c
c     first global timestep zero
c
      dt (2) = 0.e0
      do 300 nt=1,nmax
c
c     put global timestep to the minimum from last block sweep
c     and reset dtmin to find minimum for this sweep
c
      dt(1) = dt(2)
      dt(2) = 1.e20
c
c     put residual values to zero
c
      if( nt .eq. 1 .or. mod( nt , 10 ) .eq. 0 ) then
        rms(1) = 0.e0
        rms(2) = 0.e0
        rms(3) = 0.e0
      endif
c
c     loop over the block(s) (can be concurrent on the alliant)
c
ccvd$  cncall
        do 2000 nb=1,nblock
c
c     fill/empty the blocks and andvance w one time step
c
          call cablok(nt,nb,img,nstate,w0g,con,icon,rms
     +,              x,w,lam,w1,vol,s,dtl,wb,xb,bd,bw
     +,              bx,bv,rmu,re2,re4,fs,fg,fnd,f,dov
     +,              dt,nbpg,nbdg,xg,wg,mb,i2,j2,n2)
c
2000    continue
c
c     do the coprocessing graphics every nwstep timestep
c
        if( nt .eq. 1 .or. mod( nt , nwstep ) .eq. 0 ) then
#ifdef DO_GRAPH
c
```

```
            if( dogra .eq. 1 ) then
c
c     draw filled polygons
c
          call msolid(mb,m2,i2,j2,nsm,img,xg,wg,cm,w0g,nblock,255,5,2)
c
c     manipulate the object using the mouse
c
          lobj(2) = 2
          call mouseman(1,lobj,mobj,2,-1)
c
            endif
#endif
c
        endif
c
c     write maximum and rms density correction every 10 timestep
c
        if( nt .eq. 1 .or. mod( nt , 10 ) .eq. 0 ) then
          rms(2) = sqrt( rms(2)/rms(1) )
          write(*,602) nt,rms(2),rms(3),dt(1)
c
c     check if steady state is reached
c
          if( rms(2) .lt. tol .and. nt .gt. 1 ) go to 400
        endif
300   continue
400   continue
c
c     number of floating point operations
c
c     for euler
c
      flop=real(nt)*672.e0*rms(1)
c
c     or some more for navier stokes
c
      if( nstate(2,1) .eq. 1 ) flop=real(nt)*1162.e0*rms(1)
c
c     timing for alliant
c
      eti  =etime(ustime2)
      ttime=ustime2(1) - ustime1(1)
c     ttime=secnds(eti)
```

```
      flops=flop/ttime/1.e6
      write(*,604) ustime1(1),ustime2(1),ttime,flops
c
c     write out solution for rerun
c
      rewind(7)
      write (7) nblock
      do 290 nb=1,nblock
        write(7) img(-1,1,nb),img(-1,2,nb)
        do 290 j=1,img(-1,2,nb)
          do 290 i=1,img(-1,1,nb)
            write(7) wg(i,j,1,nb),wg(i,j,2,nb),wg(i,j,3,nb)
     +,              wg(i,j,4,nb),wg(i,j,5,nb)
290     continue
c
c     write out solution
c
      write(4,*) nblock
      do 5000 nb=1,nblock
        write(4,*) img(-1,1,nb),img(-1,2,nb)
        do 500 j=1,img(-1,2,nb)
          write(4,*) j
          do 500 i=1,img(-1,1,nb)
            write(4,603) wg(i,j,1,nb),wg(i,j,2,nb),wg(i,j,3,nb)
     +,                  wg(i,j,4,nb),wg(i,j,5,nb)
500     continue
5000  continue
c
c     some format statements (to amuse jan)
c
602   format(2x,i4,3g14.6)
603   format(1x,   5g15.7)
604   format(//,2x,'timing on alliant',
     +       //,2x,'starting time : ',f14.6,/,
     +          2x,'ending   time : ',f14.6,/,
     +          2x,'total    time : ',f14.6,//,
     +          2x,'teraflops     : ',f14.6,/)
c
c     close raster graphics
c
#ifdef DO_GRAPH
c
c
      if( dogra .eq. 1 ) then
```

```
        call cloras
      endif
#endif
c
c     end of program
c
      stop
      end
```

C.2 bconin.f

```
      subroutine bconin(iori,im,ins,w0g,w,x,nbp,nbd,bd,bw,bx,bv
     +,                 wb,xb,i2,j2,n2)
c
c     updating bc-arrays formed from the interior of the block
c
      parameter (m2=-1)
c
      parameter(nsm=9)
c
      real*8 x(m2:i2,m2:j2,2),w(m2:i2,m2:j2,5)
c
      real*8 wb(m2:n2,5,0:1),xb(m2:n2,2)
c
      real*8 w0g(20,nsm),wtmp(20)
c
      real*8 bd(m2:n2,5,0:1,4),bw(m2:n2,5,4),bx(m2:n2,2,4)
     +,    bv(m2:n2,5,4)
c
      integer im(-2:2,2),nbp(8,10,4),nbd(4)
c
c     bd = boundary array for dissipation
c     bw = boundary array for convective terms
c     bx = boundary array for coordinates
c     bv = boundary array with state variables for physical viscosity
c
c     size of this block
c
      isize=im(-1,1)
      jsize=im(-1,2)
c
c     loop over the side
c
      do 1000 ns=1,4
```

```
c
c     loop over the windows on this side
c
        do 1000 np=1,nbd(ns)
c
c     pick from some other blocks treated in bconex
c
          if( nbp(1,np,ns) .eq. 5 ) go to 1000
c
c     length of this window
c
          istart=nbp(2,np,ns)
          istop =nbp(3,np,ns)
c
c     go from the given input to adressable indicies
c
c     test if we address on an adjacent side
c
          if( nbp(1,np,ns) .eq. 4 ) then
c
c     side number, index direction, starting adress and full cell
c
            nsid =nbp(5,np,ns)
            nrdi =nbp(6,np,ns)
            nsta =nbp(7,np,ns)-nrdi*istart
            ihalf=0
          else
c
c     only local addressing use half cell
c
            nsid =ns
            nrdi =1
            nsta =0
            ihalf=1
          endif
c
c     find adressable indicies from given input
c
          call finadr(1,ihalf,nsid,nrdi,nsta,isize,iri,iti,idi,ixi)
          call finadr(2,ihalf,nsid,nrdi,nsta,jsize,irj,itj,idj,ixj)
c
c    access the state on adjacent side
c
          do 10 lm=0,1
```

```
          do 10 l=1,5
            do 10 io=istart,istop
              wb(io,l,lm)=
     +        w(iri*io+iti+lm*idi,irj*io+itj+lm*idj,l)
10        continue
c
c     access coordinates on adjacent side
c
          do 20 l=1,2
            do 20 io=istart,istop + 1
              xb(io,l)=
     +        x(iri*io+iti+ixi,irj*io+itj+ixj,l)
20        continue
c
c      get state for this side from list
c
          istate = nbp(8,np,ns)
          do 30 l=1,20
            wtmp(l) = w0g(l,istate)
30        continue
c
c      the boundary conditions can now be selected from a list
c
c
c      dirichlet condition - call an OMath function
c
          if( nbp(1,np,ns) .eq. 1 ) then
             call dirich(wtmp, wb, xb, ns-1
     +, istart, istop, bw, bv, bd, bx)
          endif
c
c
c      linear extrapolation - call an OMath function
c
          if( nbp(1,np,ns) .eq. 2 ) then
             call extrap(wb, xb, ns-1, istop, istart, bw,bv,bd,bx)
          endif
c
c
c      wall - call an OMath function
c
          if( nbp(1,np,ns) .eq. 3 ) then
             call wall(wb, xb, wtmp(11), ns-1
     +, ins, istart, istop, bw, bv, bd, bx)
```

```
          endif
c
c
c     pick from some other interior points
c
          if( nbp(1,np,ns) .eq. 4 ) then
c
            do 400 l=1,5
              do 400 io=istart,istop
                bw(io,l,ns)=wb(io,l,0)
                bv(io,l,ns)=bw(io,l,ns)
400         continue
c
            do 410 lm=0,1
              do 410 l=1,5
                do 410 io=istart,istop
                  bd(io,l,lm,ns)=wb(io,l,lm)
410         continue
c
            do 420 l=1,2
              do 420 io=istart,istop + 1
                bx(io,l,ns)=xb(io,l)
420         continue
c
          endif
c
c     farfield bc using 1d riemann invariants
c
          if( nbp(1,np,ns) .eq. 6 ) then
c
            call farbnd(iori,ns,wtmp,wb,xb,bw,istart,istop,n2)
c
            do 600 l=1,5
              do 600 io=istart,istop
                bv(io,l,ns)=bw(io,l,ns)
600         continue
c
            do 610 l=1,5
              do 610 io=istart,istop
                bd(io,l,0,ns)=     bw(io,l,ns)
                bd(io,l,1,ns)=2.e0*bw(io,l,ns) - wb(io,l,0)
610         continue
c
            do 620 l=1,2
```

```
            do 620 io=istart,istop + 1
              bx(io,l,ns)=xb(io,l)
620       continue
c
        endif
c
c     inflow for channel
c
        if( nbp(1,np,ns) .eq. 7 ) then
c
          call chanin(ns,wtmp,wb,bw,istart,istop,n2)
c
          do 710 l=1,5
            do 710 io=istart,istop
              bv(io,l,ns)=bw(io,l,ns)
710       continue
c
          do 720 l=1,5
            do 720 io=istart,istop
              bd(io,l,0,ns)=     bw(io,l,ns)
              bd(io,l,1,ns)=2.e0*bw(io,l,ns) - wb(io,l,0)
720       continue
c
          do 730 l=1,2
            do 730 io=istart,istop + 1
              bx(io,l,ns)=xb(io,l)
730       continue
c
        endif
c
c     outflow for channel
c
        if( nbp(1,np,ns) .eq. 8 ) then
c
          call chanout(ns,wtmp,wb,bw,istart,istop,n2)
c
          do 810 l=1,5
            do 810 io=istart,istop
              bv(io,l,ns)=bw(io,l,ns)
810       continue
c
          do 820 l=1,5
            do 820 io=istart,istop
              bd(io,l,0,ns)=     bw(io,l,ns)
```

```
              bd(io,l,1,ns)=2.e0*bw(io,l,ns) - wb(io,l,0)
820         continue
c
            do 830 l=1,2
              do 830 io=istart,istop + 1
                bx(io,l,ns)=xb(io,l)
830         continue
c
          endif
c
1000  continue
c
      return
      end
```

C.3 tmarch.f

```
      subroutine tmarch(im,ins,w0,w0g,con,icon,dt,s,bd,bw,bv,bx
     +,                 iori,wb,xb,rmu,re2,re4,fnd,f,dov,fs,fg
     +,                 ic,nbp,nbd,x,w,lam,dtl,w1,vol,i2,j2,n2)
c
c     advances state variable w one time step
c
      parameter (m2=-1)
c
      parameter(nsm=9)
c
      real*8 vol(m2:i2,m2:j2    ),w1 (m2:i2,m2:j2,4)
     +,    x  (m2:i2,m2:j2,2  ),w  (m2:i2,m2:j2,5)
     +,    lam(m2:i2,m2:j2,2  ),fnd(m2:i2,m2:j2,4)
     +,    f  (m2:i2,m2:j2,4,2),dov(m2:i2,m2:j2  )
     +,    s  (m2:i2,m2:j2,2,2),dtl(m2:i2,m2:j2  )
c
      real*8 rmu(m2:i2,m2:j2  )
     +,    re2(m2:i2,m2:j2  ),re4(m2:i2,m2:j2    )
c
      real*8 fs(m2:i2,m2:j2,6),fg(m2:i2,m2:j2,6)
c
      real*8 wb(m2:n2,5,0:1),xb(m2:n2,2)
c
      real*8 bd(m2:n2,5,0:1,4),bw(m2:n2,5,4),bv(m2:n2,5,4)
     +,    bx(m2:n2,2,4)
c
      real*8 w0(20),w0g(20,nsm),ark(5),con(20),dt(2)
```

```
      real*8 gama
c
      integer im(-2:2,2),nbp(8,10,4),nbd(4)
c
      integer icon(2),ic(4)
c
      gama = w0(8)
c
      cfl    = con(1 )
      ark(1) = con(6 )
      ark(2) = con(7 )
      ark(3) = con(8 )
      ark(4) = con(9 )
      ark(5) = con(10)
c
      itac = icon(1)
c
c     save the old state vector
c
      do 110 l=1,4
        do 110 j=1,im(-1,2)
          do 110 i=1,im(-1,1)
            w1(i,j,l)=w(i,j,l)
110   continue
c
c     five stage runge-kutta loop
c
      do 1000 nk=1,5
c
c     updating bc-arrays formed from the interior of the block
c
        if( nk .eq. 2 ) then
          call bconin(iori,im,ins,w0g,w,x,nbp,nbd,bd,bw,bx,bv
     +,               wb,xb,i2,j2,n2)
        endif
c
c     freez the numerical viscosity after the second stage
c
        if( nk .eq. 1 .or. nk .eq. 2 ) then
c
c     compute the numerical dissipation flux
c
          call numflu(im,con,f,w,lam,rmu,re2,re4,bd,i2,j2,n2)
c
```

```
c     net numerical dissipation flux
c
          do 200 l=1,4
            do 200 j=1,im(-1,2)
              do 200 i=1,im(-1,1)
                fnd(i,j,l)=f(i+1,j  ,l,1) - f(i,j,l,1)
     +                    +f(i  ,j+1,l,2) - f(i,j,l,2)
200       continue
c
c     compute physical viscosity
c
          if( ins .eq. 1 ) then
c
c     full navier-stokes physical viscosity
c
            call viscos(im,f,fnd,s,w,bw,bv,vol,w0,ic,fs,fg,i2,j2,n2)
          endif
        endif
c
c     compute convective flux
c
        call conflu(iori,im,f,s,w,bw,i2,j2,n2)
c
c     local time step
c
        if( itac .eq. 0 ) then
          do 220 j=1,im(-1,2)
            do 220 i=1,im(-1,1)
              dov(i,j) = ark(nk)*dtl(i,j)/vol(i,j)
220       continue
        else
c
c     global time step
c
          do 230 j=1,im(-1,2)
            do 230 i=1,im(-1,1)
              dov(i,j) = ark(nk)*dt(1)   /vol(i,j)
230       continue
        endif
c
c     update to the next time level
c
        do 240 l=1,4
          do 240 j=1,im(-1,2)
```

```
            do 240 i=1,im(-1,1)
c
c     net convective flux
c
              q=         f(i+1,j  ,l,1) - f(i,j,l,1)
     +                  +f(i  ,j+1,l,2) - f(i,j,l,2)
c
c     update w explicitly to next time level
c
              w(i,j,l)=w1(i,j,l) - dov(i,j)*( q - fnd(i,j,l))
240     continue
c
c     now form the pressure from the other variables
c
c     replaced by call to Omath function
c
c        w(66,128,5) = 77.77777777
c        write(*, 600) gama
c        write(*, 610) im(-1,1)
c        write(*, 610) im(-1,2)
c        write(*, 600) w(-1,-1,1)
c        write(*, 600) w(0,0,2)
c
c 600    format (1x, f9.3)
c 610    format (1x, i6)
        call perfec(gama, im(-1,1), im(-1,2), w)
c
c     end of runge-kutta step
c
1000  continue
c
      return
      end
```