INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids **28**: 1183–1197 (1998)

A FLEXIBLE APPROACH TO EXPRESSION EVALUATION WITHIN A COMPUTATIONAL ENGINEERING ENVIRONMENT

J.W. JONES* AND N.P. WEATHERILL

Department of Civil Engineering, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK

SUMMARY

In many stages of a typical computational simulation, the user has a requirement to extract data which is not always in a readily available form. Typical examples include mesh quality statistics, where the quality measure is typically defined using an expression involving the co-ordinates of each mesh cell, face, edge or node; solution visualisation, where the quantity to be displayed/analysed is an expression involving the resultant variables of the flow solver; and mesh adaption, where the refinement may be driven by a quantity which could be a combination of flow solution variables and the co-ordinates of the mesh edges. A code developer can readily modify source code to meet such requirements but this is not an option to a typical user and, when additionally, codes are embedded within graphical user interfaces. This paper describes EQUATE, a system designed to allow the user to define their own measures at run-time, and how it can be integrated into general interactive, graphical environments. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: computational simulation; visualisation; CFD; expression parsing; post-processing; compiler techniques

1. INTRODUCTION

Over the past few years the field of computational engineering, in particular computational fluid dynamics (CFD), has reached a certain level of maturity. Almost without exception, the algorithms at the heart of such software systems require various parameters to be specified by a user. Conventionally, these are input by the user through the error-prone process of manually editing text files.

Nowadays many of these codes have reached a level of stability and usability which allows them to be used by design teams who do not want to be involved in file editing, etc. This has prompted the development of a number of interactive, graphical environments which abstract the user away from the details of the actual CFD codes [1–3].

These graphical environments offer a large number of benefits to the user;

1. Geometries can be visualised and analysed as to their suitability for CFD analysis. Areas of the geometry which represent errors or problems, (e.g. unwanted gaps between surfaces or overlapping surfaces) can be highlighted graphically.
2. Mesh generation controls (e.g. point sources) can be placed interactively without the need to edit any files or even know the figures representing the dimensions of the geometry.

* Correspondence to: Department of Civil Engineering, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK.

CCC 0271–2091/98/181183–15$17.50

Received September 1997
Revised November 1997

3. Meshes can be visualised to allow easy determination of their general suitability, e.g. concentrations of points/cells in the required locations.
4. Analysis of the quality of the mesh can be performed using various quality measures. These can be displayed to the user in the form of histograms and/or in a graphical display which highlights particularly bad cells.
5. Areas of the geometry or mesh can be easily selected in order to apply boundary conditions for the solver.
6. The results from flow solvers consisting of, potentially, millions of numbers, can be visualised in a manner which is manageable using methods such as colourful contour plots, streamlines, line graphs, etc.

Two examples of graphical environments presently under development at Swansea, are PROMPT (pre-processing option for military powerplant technology) and the PSUE (parallel simulation user environment). PROMPT, developed under joint funding by Rolls-Royce and DERA, is an interactive environment in which CFD analysis can be performed using structured multiblock, unstructured or hybrid grid technology using a number of flow solvers. To increase flexibility and ease of maintainability PROMPT is split into several modules using a client–server model as shown in Figure 1. The visualisation module integrated within the system allows the user to interactively manipulate and select entities using the mouse and keyboard. It also acts as the server storing all the data currently being used by the system. Each of the other modules are designed to perform one particular step of the CFD analysis process. Only the modules currently in use are initiated at any one time, thus saving memory. Data and commands are passed between these client modules and the server using high-speed communication links.

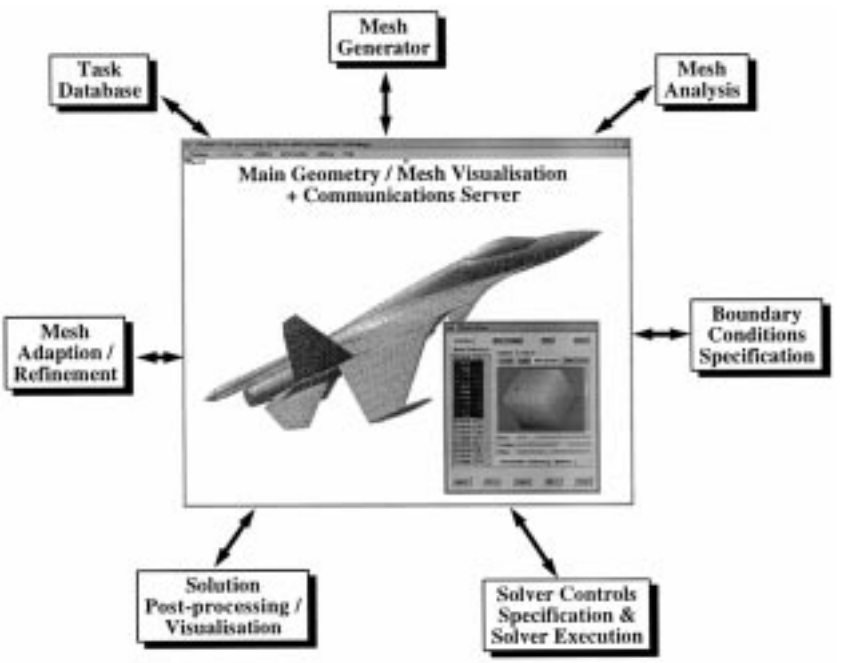


Figure 1. The client–server model used in PROMPT.

The PSUE [1,3], developed under funding from EU (ESPRIT), is a general software environment which is similar to PROMPT but has an architecture which enables arbitrary software modules to be coupled into the environment. Both PROMPT and the PSUE are developments which address the issue of the computer–human interface to increase the ease-of-use, effectiveness and versatility of advanced computational engineering algorithms.

2. THE NEED FOR EQUATE

In the area of solution visualisation and post-processing it is commonplace for commercial packages to allow the definition of new scalar and vector variables by combining several generic flow variables in an expression [9,10].

During the development of the systems PROMPT and the PSUE, it was evident that this facility would also be very useful in two other stages of the analysis, namely mesh quality analysis and mesh adaption/refinement. However most expression evaluation systems in visualisation packages are limited to defining scalar or vector quantities based on nodes. These node-based variables are not general enough to be useful for mesh adaption and mesh quality analysis since these operations require expressions that are based on other types of data, e.g. edge-, face- and cell-based data.

To meet these requirements, a module, called EQUATE (equation editor), has been designed to allow the user to define expressions relating to:

1. Mesh quality analysis [8]. When ascertaining if a mesh is of a suitable quality a whole range of measures are used, including cell skewness, mesh cell expansion ratio, cell aspect ratio, etc. These measures are inherently of different data-types, e.g. cell aspect ratio is a cell-based measure, whereas mesh expansion ratio would be defined as a face-based measure of the ratio of the volumes of the cells on either side of that face. They could not be defined solely in terms of quantities at individual nodes.
2. Solution visualisation/post-processing. As mentioned previously, solution visualisation is usually involved with the calculation and display of node-based data, usually of a scalar type; even vector types can be broken down to three scalar types, each representing a different component of the vector.
3. Mesh adaptation/refinement [6]. The type of variables used for mesh adaption are similar to those used for solution visualisation, since the adaption of the mesh is usually controlled by some aspect of the solution. But rather than being node-based variables, mesh adaption would often use edge-based variables since the quantity of interest is the rate of variation of the node-based values along an edge. Again, these quantities could not be defined solely in terms of a list of nodes.

3. EQUATE (EQUATION EDITOR)

During the design of EQUATE a number of requirements were highlighted;

1. The ability to cope with different data-types. To allow EQUATE to be utilised in these three different stages it must be able to evaluate equations that are based on variables computed at nodes, edges, faces and cells, even though the input data to EQUATE is always node-based. This is achieved by grouping the individual nodal values into tuples to represent the higher level entities. For example, grouping nodes at either end of an edge

into pairs creates a pair (two-tuple) which represents an edge value. This would be useful in mesh adaptation, where the rate of change of solution values along an edge could easily by calculated by dividing the difference between the two end nodal values by the distance between them.

2. Completeness. To enable the user to quickly and easily define new variables EQUATE contains all of the usual mathematical operators (e.g. $+$, $-$, *, etc.) and functions (e.g. log, e^x, $\sin(x)$, etc.). A number of other functions which could be defined using the generic operators and functions are included in EQUATE for efficiency purposes due to their frequent usage (e.g. edge length, face area, cell volume, dot product, etc.).
3. Speed. Evaluating a particular variable in EQUATE on a large mesh will require the same expression to be evaluated possibly 1–5 million times. Obviously the time to evaluate an expression must be as quick as possible in order to provide a reasonable response time to the user.

3.1. Definition of a generic mathematical expression in EQUATE

In EQUATE a mathematical expression consists of classes of constants, variables, operators and functions, i.e. a generic mathematical expression E is expressed as

$$E = f(C, V, O, G) \quad \text{where}$$

$C = (c_1, c_2, \ldots, c_n)$ is the class of constants, e.g. $1, 2, 3.5, \lambda, \pi, -1.5$, etc.

$V = (v_1, v_2, \ldots, v_n)$ is the class of variables, e.g. pressure (p), density (ρ), etc.

$O = (o_1, o_2, \ldots, o_n)$ is the class of operators, e.g. $\alpha + \beta, \alpha - \beta, \alpha \times \beta, \alpha \div \beta, \alpha^{\beta}$, etc.

$G = (g_1, g_2, \ldots, g_n)$ is the class of functions, e.g. $\sin(\phi), \log(\phi), e^{\phi}, \min(\phi, \psi)$, etc.

For example, the equation

$$\left(\frac{\rho^{\lambda} + 2}{p^{\log(\lambda)} - 1}\right)^{3.5}$$

is expressed in EQUATE as

$$E = f(C, V, O, G) \quad \text{with}$$

$$C = \left(\underset{(c_1)}{1}, \underset{(c_2)}{2}, \underset{(c_3)}{3.5}, \underset{(c_4)}{\lambda}\right),$$

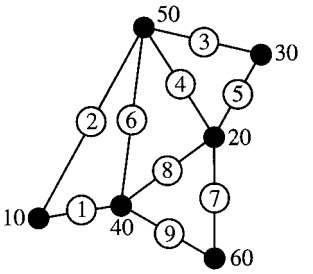
$$V = \left(\underset{(v_1)}{p}, \underset{(v_2)}{\rho}\right),$$

$$O = \left(\underset{(o_1)}{\alpha + \beta}, \underset{(o_2)}{\alpha - \beta}, \underset{(o_3)}{\alpha \div \beta}, \underset{(o_4)}{\alpha^{\beta}}\right),$$

$$G = \left(\underset{(g_1)}{\log(\alpha)}\right),$$

where

$$E = \left(\frac{(v_2^j) \wedge_{o_4} (c_4^j) +_{o_1} (c_2^j)}{(v_1^j) \wedge_{o_4} g_1(c_4^j) -_{o_2} (c_1^j)}\right) \wedge_{o_4} (c_3^j) \quad (j \text{ is the cardinality of the tuple}).$$



Edge	Node 1	Node 2
1	10	40
2	10	50
3	50	30
4	50	20
5	20	30
6	50	40
7	20	60
8	20	40
9	40	60

Figure 2. Example of representing edges by forming node pairs.

3.2. EQUATE syntax and semantics

As mentioned above, EQUATE can evaluate expressions based on data defined at nodes, edges, faces and cells. The input to EQUATE consists entirely of nodal values; either solution values output by the solver or geometric values based on the mesh itself.

In order for EQUATE to evaluate expressions based on entities other than nodes the nodal values are grouped into tuples of varying cardinality to represent the desired entity. An example of the tuples formed to allow edge-based expressions to be evaluated is shown in Figure 2; Figure 3 shows a similar grouping of nodes into three-tuples for triangular faces. Cell and quadrilateral face tuples are constructed in a similar manner.

New variables can be defined using any valid combination of constants, built-in variables (both geometric and generic solver variables), operators and functions.

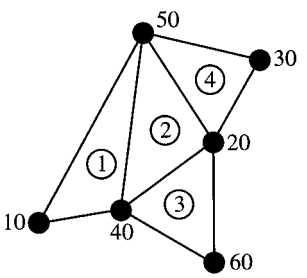
The following sections deal with each component in turn.

3.3. Syntax and semantics of expressions

In EQUATE an expression can be built from a number of sub-expressions separated by semi-colons. Each sub-expression (except for the last) has an assignment section at the front. This causes the value of the sub-expression to be assigned to the specified local variable. The last sub-expression has no assignment section since the result is automatically assigned to the global variable.

The purpose of the sub-expression is to mimic the normal method of writing mathematical equations, with the complete expression split into more manageable parts. It also means that if any sub-expression is used more than once in the global expression then it can be evaluated once only and the values stored to be re-used later, rather than re-calculate it each time it appears.

A simple example of this is shown below. The edge-based expression



Face	Node 1	Node 2	Node 3
1	10	50	40
2	40	50	20
3	60	40	20
4	30	20	50

Figure 3. Example of representing triangular faces by forming three-tuples of nodes.


```
((Density.n2-Density.n1)/Length)
/((Density.n2-Density.n1)/Length-1)
```

can be simplified to

```
a=(Density.n2-Density.n1)/Length;
a/(a-1)
```

An expression containing sub-expressions is semantically valid if, and only if, the expression would be correct if the sub-expressions were all inserted to form one global expression.

3.4. Syntax and semantics of operators

To build expressions out of variables and constants it is necessary to combine them using mathematical operators and functions. The operators included in EQUATE consist of the usual mathematical operators; $a+b$, $a-b$, $a*b$, etc.

Operators are like functions (in fact operators can be replaced with an equivalent function) except that they are restricted, in that the operands must be n-tuples of the same cardinality; the result of the operator will be an n-tuple with that same cardinality.

The operators broadly fall into two categories; unary and binary. The unary operators in EQUATE take one n-tuple as an argument and return one tuple with the same cardinality, n. The only unary operator currently in EQUATE is negation which is defined as

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n),$$

then

$$-(A^n) \rightarrow B^n \quad \text{where} \quad B^n = (-a_1, -a_2, \ldots, -a_n).$$

EQUATE also contains five binary operators, $a+b$, $a-b$, $a*b$, a/b, a^b. These have similar definitions to the unary operators,

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n), \quad \text{and} \quad B^n = (b_1, b_2, \ldots, b_n),$$

then

$$A^n \odot B^n \rightarrow C^n \quad \text{where} \quad C^n = (a_1 \odot b_1, a_2 \odot b_2, \ldots, a_n \odot b_n),$$

where

$$x \odot y = x+y, x-y, x*y, x/y \text{ or } x^y.$$

As shown above, the two arguments of each of these operators must have the same cardinality. The result of the operator is a tuple with the same cardinality as the arguments.

3.5. Syntax and semantics of functions

There are a substantial number of functions built into EQUATE. These include exponential, logarithmic and trigonometric functions, and some miscellaneous functions.

The exponential and logarithmic functions are e^x, $\log_e(x)$ and $\log_{10}(x)$. The trigonometric functions are $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sin^{-1}(x)$, $\cos^{-1}(x)$ and $\tan^{-1}(x)$. The other functions don't really fit into a particular category so are classed as miscellaneous functions; these are $\sqrt{x}$, $\min(x, y)$ and $\max(x, y)$.

These functions are either unary functions or binary functions and thus have the same properties as unary and binary operators, respectively. They take one or two n-tuples as arguments and return a tuple of the same cardinality, n. The unary functions are defined as

Let $A^n = (a_1, a_2, \ldots, a_n)$,

then

$$f(A^n) \to B^n \quad \text{where} \quad B^n = (f(a_1), f(a_2), \ldots, f(a_n)),$$

where

$$f(x) = e^x, \log_e(x), \log_{10}(x), \sin(x), \cos(x), \tan(x), \sin^{-1}(x), \cos^{-1}(x), \tan^{-1}(x) \text{ or } \sqrt{x}.$$

The binary functions are defined as

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n), \quad \text{and} \quad B^n = (b_1, b_2, \ldots, b_n),$$

then

$$f(A^n, B^n) \to C^n \quad \text{where} \quad C^n = (f(a_1, b_1), f(a_2, b_2), \ldots, f(a_n, b_n)),$$

where

$$f(x, y) = \min(x, y) \text{ or } \max(x, y).$$

There are also some functions in EQUATE which operate on an n-tuple and reduce it to a single number (one-tuple). These are $t\min(x)$, $t\max(x)$, $t\text{avg}(x)$ and $t\text{rms}(x)$. These functions could be performed using EQUATE's more basic features but since they are common operations they have been built in both for ease of use and efficiency of execution. They are defined as

Let $A^n = (a_1, a_2, \ldots, a_n)$,

and

$$C^1 = (c_1),$$

then

$$t\min(A^n) \to C^1 \quad \text{where} \quad C^1 = \min_{i=1}^{n}(a_x),$$

$$t\max(A^n) \to C^1 \quad \text{where} \quad C^1 = \max_{i=1}^{n}(a_i),$$

$$t\text{avg}(A^n) \to C^1 \quad \text{where} \quad C^1 = \frac{\sum_{i=1}^{n} a_i}{n},$$

$$t\text{rms}(A^n) \to C^1 \quad \text{where} \quad C^1 = \sqrt{\frac{\sum_{i=1}^{n} a_i^2}{n}}.$$

Finally, for completeness, the last group of functions in EQUATE compress n single numbers (one-tuples) into one n-tuple. These are given the names *tuple*2, *tuple*3, ..., *tuple*8 depending on how many arguments they take. These are defined as

Let $c_1, c_2, \ldots, c_8$ be single numbers (1-tuples),

then

$$tuple2(c_1, c_2) \to A^2 \quad \text{where} \quad A^2 = (c_1, c_2)$$

$$tuple3(c_1, c_2, c_3) \to A^3 \quad \text{where} \quad A^3 = (c_1, c_2, c_3)$$

$$tuple8(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8) \to A^8 \quad \text{where} \quad A^8 = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$$

The cardinality of a tuple that can be created is dependent upon the semantic requirements at that point in the expression. Any tuple created that is not compatible will generate an error.

3.6. Syntax and semantics of variables

In EQUATE there are two different types of variable that can be used in an expression;

1. Built-in solver variables: for node-based expressions, these variables take on the solution value at each mesh node in turn. For edge-based expressions, the variables take on the pair of values of the nodes at either end of each edge in the mesh. Face and cell variables are constructed in a similar fashion.
2. Geometric variables: these variables are constructed in the same way as the built-in solver variables but instead of containing solution values they contain geometric values. For unstructured meshes, there are three such variables:

`X`—The x-co-ordinate of the node
`Y`—The y-co-ordinate of the node
`Z`—The z-co-ordinate of the node

Unlike the built-in solver variables, there are also some variables used specifically with the higher order types. A small number of these were selected to be hard-wired in EQUATE due to their frequent usage to improve speed. These are

`Length`(Edge)—The length of the edge
`Area`(Face)—The area of the face
`Volume`(Cell)—The volume of the cell
`CentreX`(All types)—The x-co-ordinate of the centre
—of the edge, face or cell
`CentreY`(All types)—The y-co-ordinate of the centre
—of the edge, face or cell
`CentreZ`(All types)—The z-co-ordinate of the centre
—of the edge, face or cell

In the present version of EQUATE any variable that is created by the user by defining an expression cannot be used in subsequent expression definitions.

Syntactically the definition of any variable in EQUATE is the same as an identifier in the language 'C', i.e. a lower or upper case letter or an underscore followed by a sequence of zero or more letters, underscores or digits.

Like a constant, the semantic definition of a variable is an n-tuple of numbers. Unlike a constant, the cardinality of this tuple is fixed and if it does not match the requirements of the expression an error is generated.

The cardinality of a variable depends on the data-type it represents, as shown below.

$$V^n = (e_1, e_2, \ldots, e_n) \begin{cases} n = 1 & \text{(node)} \\ n = 2 & \text{(edge)} \\ n = 3 & \text{(triangular face)} \\ n = 4 & \text{(quadrilateral face/tetrahedral cell)} \\ n = 5 & \text{(pyramid cell)} \\ n = 6 & \text{(prismatic cell)} \\ n = 8 & \text{(hexahedral cell)} \end{cases}$$

3.6.1. Accessing variable sub-components. The sub-components of any variable of higher order than a node-based variable can also be accessed and used in expressions. This is achieved by adding a suffix `.n`, `.e` or `.f` followed by the node, edge or face number to the end of the variable name.

When a sub-component is accessed in this manner the type of variable becomes the type of the sub-component being accessed. As with variables the type of this component must satisfy the requirements of the expression.

Examples of accessing sub-components are

Cell-based variables—`Pressure.e3, Density.n1,`
—`Entropy.f4`
Face-based variables—`Temperature.e1, Density.n4`
Edge-based variables—`Pressure.n1`

An example of accessing sub-components of edge-based variables in the context of mesh adaptation could be calculating the density gradient along each edge of the mesh. An edge-expression could be defined,

```
(Density.n2-Density.n1)
/sqrt((XCoord.n2-XCoord.n1)^2+(YCoord.n2-YCoord.n1)^2
+(ZCoord.n2-ZCoord.n1)^2)
```

although this could be simplified by using one of the built-in variables, `Length`,

```
(Density.n2-Density.n1)/Length
```

3.7. Syntax and semantics of constants

To the user, constants are simply floating-point (real) numbers. However, in EQUATE, constants are stored as a tuple consisting of one or more of the same constant. The cardinality of the tuple is automatically determined by EQUATE so as to satisfy the semantic requirements of the expression. For example, when defining a node-based expression, a constant is stored as a single number (one-tuple); in an edge-based expression a constant is stored as a two-tuple consisting of the same constant number.

In general an EQUATE *constant* is defined as an n-tuple,

$$C^n = c, \underbrace{c, \ldots, c}_{n \text{ times}},$$

where c is the constant value, and n is chosen to satisfy the semantic requirements of the expression.

Constants may also be defined directly as a tuple by surrounding a list of comma-separated constants with curly brackets. The number of entries in the defined tuple must match the semantic requirements of the expression or an error will be generated.

Examples of constants are

```
12, 12.45, -1.2e-6, {3.0, -1, 5e2, -1E0.5}
```


3.8. Some implementation details

In order to maximise EQUATE's use throughout the department, it was decided to implement it as a library which would be linked with the users code during compilation. EQUATE obtains its data through a string containing the expression as typed by the user and a number of simple arrays. The string is parsed once using LEX and YACC [4,5,7] to produce a tokenised version which is then easily and efficiently interpreted for each of the (possibly millions of) data items. The arrays contain the generic flow solution variables, the co-ordinates of the nodes of the mesh, and the mapping between the names of the generic flow variables and their actual values. The variables and co-ordinates are stored in groups depending on whether the expression is based on cells, faces, edges or nodes, e.g. an edge-based expression requires its data with the variables and co-ordinates grouped into pairs representing the nodes at either end of each edge.

Since EQUATE is implemented as a library, it is straightforward to construct a simple wrapper around it to allow standalone execution with the data being obtained from a set of data files.

4. PERFORMANCE FIGURES

One of the main requirements for EQUATE was speed, because for a large mesh an expression could be evaluated 1–5 million times (once per cell). Obviously any improvement in speed, however small, will have a significant effect overall.

It was decided that each expression evaluated using EQUATE would be timed and compared with the same expression hard-wired in C source code. Obviously since EQUATE is a form of interpretive compiler it could never match the speed of the native compiled code, but the closer it got to this lower bound the more effective it would be.

Expressions of varying complexity were evaluated using a structured mesh containing 4 million nodes (11.9 million edges). Some expressions were chosen to produce meaningful results whereas others were chosen simply to illustrate the functionality of EQUATE.

The results along with the relative speed comparisons are shown below. The speeds were obtained on a Silicon Graphics Challenge with 512 Mb of memory.

4.1. Simple expressions

The following set of expressions are very simple using only the basic arithmetic operations. The figures are designed to show the speed comparisons between EQUATE and hard-wired C code when the overhead of the various flow constructs (e.g. loops, if-then-else statements, etc.) are responsible for a significant amount of the total time.

	Equation	EQUATE time (s)	Hard-wired C-code time (s)	Relative Speed %
1	`Density+Temperature`	8.6	2.8	3.1
2	`Density.n1*Temperature.n2 +Pressure.n2`	47.3	22.7	2.1
3	`(Density.n2-Density.n1)/Length`	50.6	25.6	2.0

4.2. Complex expressions

These expressions are somewhat more complex than the previous examples in the sense that functions which are more CPU intensive to compute are used, (e.g. $\sqrt{x}$, $\log(x)$, $\sin(x)$, etc.). These expressions are designed to illustrate how EQUATE compares with hard-wired C code when the time needed to calculate the expression outweighs the overhead caused by the various flow constructs.

	Equation	EQUATE time (s)	Hard-wired C-code time (s)	Relative Speed %
1	`sin(Density)^2+cos(Density)^2`	40.9	29.5	1.4
2	`(log(Temperature.n2^4)` `+log(Temperature.n1^2))^2`	201.2	161.2	1.25
3	`Density.n2^Density.n1/` `(Temperature.n1-Temperature.n2)`	128.6	98.6	1.3

4.3. Explanation of figures

As can be seen from the figures, the hard-wired C code was quicker in every case than EQUATE, but this was to be expected. For the simple expressions EQUATE was ≈ 2–3 times slower, but this dropped to only 1.2–1.5 times slower for the more complex expressions. Complex expressions, in this context, signify expressions which use more complex functions/operators (e.g. $\sin(x)$, $\log(x)$, x^y, etc.); whereas simple expressions signify expressions which use quicker, simpler operators and functions (e.g. $x+y$, $x-y$, $\min(x, y)$, etc.).

The reason for this behaviour is easily explained if the total execution time is split into two parts,

$$T = F + C,$$

where T is the total execution time, C is the time spent actually evaluating the operators and functions in the expression, and F is the time spent executing the surrounding flow constructs.

We shall use $T_{\text{Equate}}^{\text{Compute}}$ to denote the total execution time for the complex expressions using EQUATE; similarly for F and C.

We shall represent the relative speed of EQUATE to the hard-wired C code as

$$S = \frac{T_{\text{Hard}}}{T_{\text{Equate}}}.$$

It is obvious that $C_{\text{Equate}} = C_{\text{Hard}}$ for the same expression whether simple or complex, since the same operators and functions are being evaluated. Therefore the relative speed, S, is directly dependent on the difference between F_{Equate} and F_{Hard}. The fact that EQUATE is slower than hard-wired C code (i.e. $F_{\text{Equate}} > F_{\text{Hard}}$) is due to the increased complexity of the

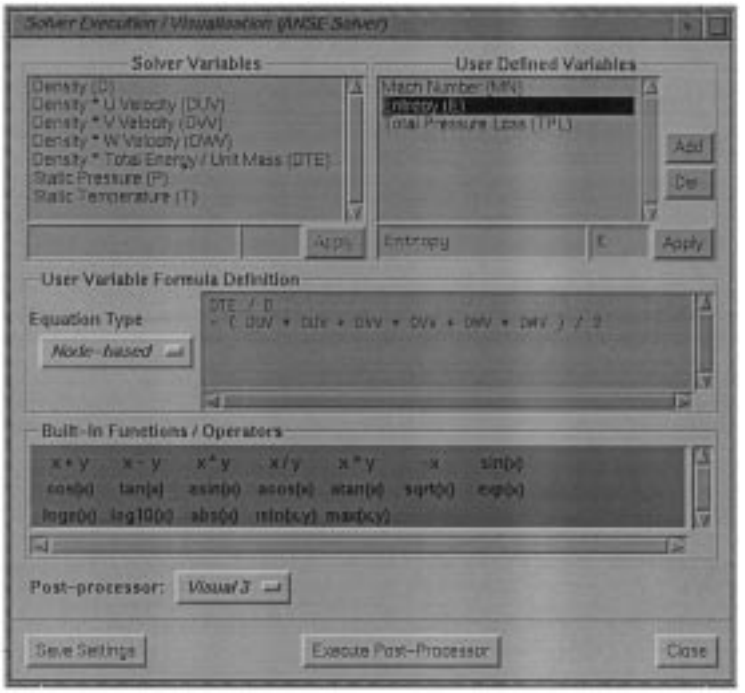


Figure 4. Interface for solution visualisation using EQUATE.

code in EQUATE required to parse the expression tree and decide which operations are to be carried out.

It is also obvious from the results that as the expression gets more complex (but still has the same number of operands) the relative speed of EQUATE to the hard-wired C code increases, i.e. $S^{\text{Complex}} < S^{\text{Simple}}$.

Expanding the above we have

$$\frac{F_{\text{Hard}}^{\text{Complex}} + C_{\text{Hard}}^{\text{Complex}}}{F_{\text{Equate}}^{\text{Complex}} + C_{\text{Equate}}^{\text{Complex}}} < \frac{F_{\text{Hard}}^{\text{Simple}} + C_{\text{Hard}}^{\text{Simple}}}{F_{\text{Equate}}^{\text{Simple}} + C_{\text{Equate}}^{\text{Simple}}}.$$

Now $C_{\text{Equate}} = C_{\text{Hard}}$, therefore

$$\frac{F_{\text{Hard}}^{\text{Complex}} + \lambda^{\text{Complex}}}{F_{\text{Equate}}^{\text{Complex}} + \lambda^{\text{Complex}}} < \frac{F_{\text{Hard}}^{\text{Simple}} + \lambda^{\text{Simple}}}{F_{\text{Equate}}^{\text{Simple}} + \lambda^{\text{Simple}}}.$$

Re-arranging, we have

$$\frac{F_{\text{Hard}}^{\text{Complex}} + \lambda^{\text{Complex}}}{F_{\text{Hard}}^{\text{Simple}} + \lambda^{\text{Simple}}} < \frac{F_{\text{Equate}}^{\text{Complex}} + \lambda^{\text{Complex}}}{F_{\text{Equate}}^{\text{Simple}} + \lambda^{\text{Simple}}}.$$

The expression has the same number of operands, $F_{\text{Simple}} = F_{\text{Complex}}$, therefore

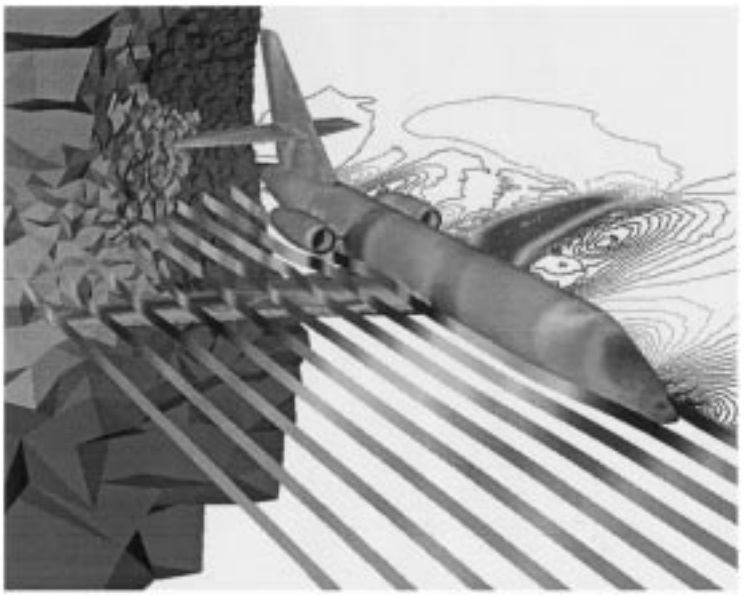


Figure 5. A typical flow visualisation over an aircraft with variables obtained using EQUATE.

$$\frac{\phi_{\text{Hard}} + \lambda^{\text{Complex}}}{\phi_{\text{Hard}} + \lambda^{\text{Simple}}} < \frac{\phi_{\text{Equate}} + \lambda^{\text{Complex}}}{\phi_{\text{Equate}} + \lambda^{\text{Simple}}}.$$

Now ϕ_{Hard} and ϕ_{Equate} are constant, regardless of the complexity of the expression; therefore it can be seen that as the complexity of the expression increases the two sides of the relation converge to equality.

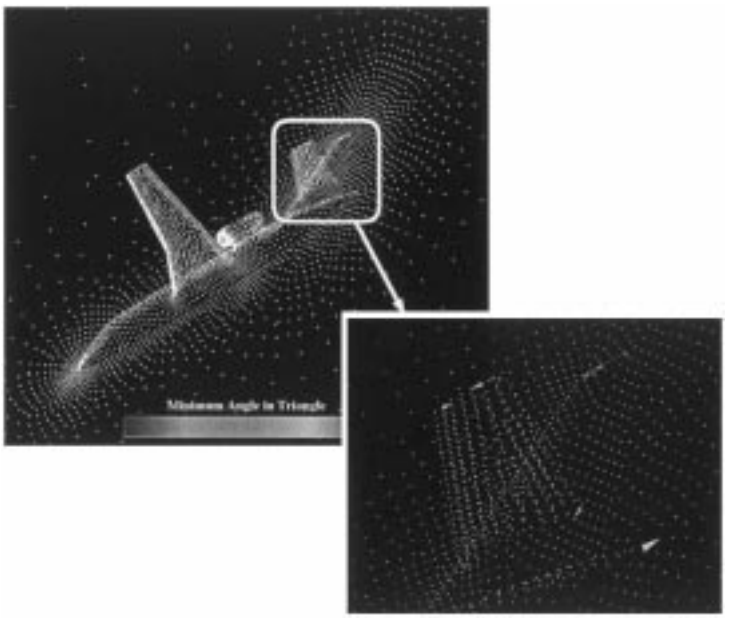


Figure 6. Mesh quality analysis performed using EQUATE on the surface mesh of the above aircraft.

5. SOME TYPICAL APPLICATIONS OF EQUATE

As mentioned earlier, EQUATE is generic and can be applied in many different frameworks, including mesh quality analysis, solution visualisation and mesh adaptation.

The first example concerns solution visualisation, the interface constructed around EQUATE is shown in Figure 4. The list of variables on the left are the generic flow variables produced by the flow solver and the list of variables on the right are the user-defined variables. The box below these two lists contains the equation defining the currently selected user variable. The example shown below in Figure 5 is of a flow analysis over a complete aircraft. The data used for the display of the colour coding of the aircraft surface, the iso-contours and the streamlines, shows typical results obtained with EQUATE.

The second example shows how EQUATE can be used to analyse the quality of a mesh. The interface constructed for this purpose is similar in appearance to that for solution visualisation. Figure 6 shows the surface mesh for the same aircraft as above with cells with small interior angles highlighted.

6. CONCLUSION

As can be seen from the performance figures, EQUATE is, at worst, three times slower than hard-wired C source code. At first glance this may look very significant, but considering, for this worst case, the calculation took only of the order 10 s for 4 million nodes, 3×10 s is not too long a response time for the average user; and as the equation gets more complex the performance penalty EQUATE exhibits reduces significantly.

The extra flexibility that EQUATE gives the user in an interactive CFD environment like PROMPT and the PSUE far outweighs the performance loss incurred through its use. If the performance loss is considered too great by the developer of a CFD package then EQUATE could be used in parallel with some commonly used expressions hard-wired into the package. That way the performance penalty would only be incurred when a user needed a measure not available within the environment.

REFERENCES

1. E. Turner-Smith, M.J. Marchant, Y. Zheng, M. Sotirakos and N.P. Weatherill, 'A parallel simulation user environment for computational engineering', *5th Int. Conf. on Grid Generation in CFD and Related Fields*, Stariville, Mississippi, April 1995.
2. J.A. Murphy, N.P. Weatherill, E.A. Turner-Smith, D.P. Rowse and C. Guichard, 'Computing environments for controlling simulations in computational electro-magnetics and multi-physics', *3rd Int. Conf. in Electro-magnetics*, University of Bath, April 1996.
3. M.J. Marchant, E.A. Turner-Smith, Y. Zheng, M. Sotirakos and N.P. Weatherill, 'The design of a graphical user environment for multi-disciplinary computational engineering', *2nd ECCOMAS Conference on Numerical Methods in Engineering*, Paris, September 1996.
4. A.V. Aho, R Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.
5. J.P. Bennett, I*ntroduction to Compiling Techniques: A First Course using ANSI C, Lex and Yacc*, McGraw Hill, New York, 1996.
6. A.F.E. Home, 'Adaptive mesh generation within a 2D CFD environment using optimisation techniques', in *Notes on Numerical Fluid Mechanics Vol.44: Multiblock Grid Generation, Results of the EC/BRITE- EURAM EU-ROMESH*, 1990–1992.
7. C. Donnelly and R. Stallman, *Bison—The YACC-Compatible Parser Generator*, published by the Free Software Foundation, November 1995.
8. T. Fol and V. Treguer-Katossky, 'Brite Euram "Euromesh" sub task 1.2: basic metrics for mesh quality', *Aérospatiale Division Avions, Note Technique No. 443.558/91*, December 1991.

9. P.P. Walatka, J. Clucas, R.K. McCabe, T. Plessel and R. Potter, *FAST 1.1 User Guide*, NASA Aimes Research Centre: NAS Division, RND Branch, June 1993.
10. *Ensight 5.5 User Manual*, Computational Engineering International, Inc., 1995.

