An Efficient Symbolic-Numeric Environment By Extending Mathematica's Format Rules.

M. SOFRONIOU*

Abstract

A package `Format.m` extending Mathematica's built-in formatting rules is presented. Examples of shortcomings of the standard rules are given, where code generated may not be syntactically correct and of appropriate precision. These issues are addressed and the formatting rules are extended to include lists as data objects and assignments to expressions. The package may be used to considerably enhance and automate code generation. The result is a symbiosis of symbolic-numeric environments built upon the existing `Splice` communication process. Finally, some examples from applied numerical mathematics are given, where the package has been used to establish a generalised formulation to a problem via the use of a template file.

*Dipartimento di Matematica, Universitá degli Studi di Bologna, Piazza Porta San Donato, 40126 Bologna, Italia. Email: sofronio@dm.unibo.it

Contents

1 Introduction

This document is an introduction to the package `Format.m`. The main purpose of the package is to translate Mathematica expressions and assignments into C [10], FORTRAN77 [20], TEX[12] and Maple [17].

Mathematica has its intrinsic formatting rules, `CForm`, `FortranForm` and `TeXForm` etc. These often fail to give accurate and syntactically correct output and are somewhat limited in scope. These are compared and contrasted with output from the `Assign` functions of `Format.m`. It should be emphasised that some of the examples chosen to demonstrate shortcomings in the formatting rules are also valid for other symbolic packages. In fact, some of the features outlined are (as yet) unavailable in alternative symbolic environments.

There are traditional conceptions the roles and capabilities of compiled and symbolic languages. Commonly, symbolic computation programs are regarded as fairly limited in most of the following areas:

1. Solving non-linear ODEs and PDEs
2. Solving non-linear algebraic equations
3. Solving non-linear optimization problems

These problems are the province of numerical computation, because adequate analytic solution methods may not exist and therefore may not be implemented in a symbolic computation package. However symbolic methods can be used to derive expressions necessary for performing numerical computations - such as gradients and Jacobian and Hessian matrices. Thus, the traditional roles of numeric and symbolic computation are not distinct and many benefits arise from merging the two.

Numerical computations in Mathematica (being an interpreted language) can be limited in use for large scale problems [27, 30]. Mathematica is a useful tool for obtaining a quick formulation to a problem, but often involves high computational overhead when compared with compiled languages. Hence there is a need to translate symbolic code into a more efficient computational environment. It is believed that the integration of symbolic-compiled language environments will lead to many new approaches in areas of applied science and numerical mathematics [6, 34, 27].

There are two forms of communication process in Mathematica - structured and unstructured communication. Structured communication is achieved via `MathLink` [18]. The basic idea is to exchange complete Mathematica expressions to external programs which are specially set up to handle such objects. Structured communication via `MathLink` requires strong data typing for all information passed in the communication process and necessarily has a very complex syntax. Often there is no need to establish a two-way communication process, but what is required is the translation of symbolic code for use in an alternative environment.

Unstructured communication consists of sending and receiving ordinary text from external programs. The package `Format.m` is intended primarily for use with the `Splice` form of unstructured communication, but may also be used interactively within a Mathematica session. `Splice` is a one-way communication process which, the author feels, has not achieved it's full potential due to syntactical limitations in Mathematica's built-in formatting code. `Format.m` has been developed to address these issues.

The main ethos behind `Splice` is to set up a (generalised) template file containing a mixture of (active) Mathematica and (passive) compiled language source code. The template file is processed with `Splice`, which evaluates the active parts (the passive parts remain unaffected), producing a file consisting solely of compiled language code. It is important to emphasise the portability of the source code files. In general a template file to a problem is written, Mathematica then processes this file using `Splice` and the resulting file may be transferred to an alternative hardware platform for efficient execution.

By utilising the (numerous) available options, the functions defined in `Format.m` allow a high degree of user control. This enables precise formatting of expressions which eliminates the need for editing of the resulting code. The result facilitates the automation of code generation and the merging of symbolic-numeric environments built upon the existing interface and communication protocols.

The general design philosophy relies on attaching rules to functions in order to reformat them. This will ultimately result in longer execution times when compared with the built-in formatting functions. However, the goal is efficient and accurate compiled language code. It is anticipated that the compiled language code will be used extensively and may be executed many times. The small degradation in performance when translating code is a trade-off that is worth making, in order to provide more accurate and more flexible formatting tools. The current version of `Format.m` consists of over 1500 lines of code (including full on-line documentation) which is an indication of the complexity of the task involved.

There are several intended purposes for this documentation. Firstly it serves as a tutorial introduction to `Format.m`. Secondly it provides a comprehensive test suite which is used extensively in the development stages of the package. Thirdly, the package and examples are updated as new versions of Mathematica appear. The evolution spans several years including development and the documentation refers to the features and implementation strategies of the current version.

In the next section a summary of the main features of the `Assign` functions of `Format.m` is given along with the steps required for initialisation. The construction of the package is then briefly detailed, illustrating design decisions. Examples of interactive usage within a Mathematica session are given throughout and contrasted with the standard formatting rules. The implementation is broken down into several basic components which are outlined in detail in section 4. Extensions to other format types is demonstrated. A function with no current parallel in Mathematica, `MapleAssign`, has been added to translate basic expressions from one symbolic environment to another. The issue of optimization of computational sequences has also been addressed [29]. Optimization of the generated code can yield significant performance benefits and a brief introduction is given. Finally, some examples demonstrating practical usage of the symbolic-numeric environment are provided. The package has been developed and used extensively for research in numerical mathematics [27], and examples are naturally taken from this area. There are many other potential applications.

2 Overview

An overview of the features available in `Format.m` is given in this section. The initialisation steps that are assumed in subsequent sections are then outlined.

2.1 Summary of features

Below is a brief summary of the main features of the `Assign` and related functions of `Format.m`.

- `CAssign` translates to C, `FortranAssign` to FORTRAN77 and `MapleAssign` to Maple.
- The `Assign` function accepts any Mathematica `OutputForm` (e.g. `TeXForm`).
- Translation of and assignments to expressions and lists of expressions is possible. Multiple assignments are also possible.
- Many options are provided for precise control over the formatting of expressions.
- Large FORTRAN expressions can automatically be broken into sub-expressions in order to stay within compiler limits.
- Code translation adheres to ANSI C and FORTRAN standards.
- Optimized computational sequences are possible.
- Many Mathematica rules have been rewritten to produce more efficient executable code.
- The range of integer and floating point numbers can be checked against IEEE standards.
- Syntax checking on all options is performed and full on-line documentation is available.

2.2 Initialisation

In order to reproduce the examples which follow, the package `Format.m` is required, along with the files `driver.f`, `example.mf`, `func.mf`, `linsolv.mf`, `newton.mf`, `rksub.mf`, and `sub.mf`. All Mathematica executable code in subsequent sections is labelled using `In` and `Out` references, initialised in each new section unless otherwise stated.

The next executable statements perform the following three steps. The first (optional) step switches off spelling warning messages for variables with similar names. The second step sets the line width to 73 characters for output. This is required for producing 72 column FORTRAN code (an additional character being required for line-breaking).[1] Note that for the the `NoteBook` front end, the option `Break at` x `character width` (in the `Edit/Preferences/Action` menu) may over-ride with this setting. The third step loads in the package. An optional directory path may also need to be be specified, depending upon where the package resides.

```
In[1]:= Off[General::spell,General::spell1];
In[2]:= SetOptions[$Output,PageWidth->73];
In[3]:= <<Format`;
```

In the sections which follow, input lines begin with `In[4]` if these steps have been assumed.

3 Construction

This section explains some of the details of how the package has been implemented and the general design philosophy.

3.1 Output formats

A print form (or output form) in Mathematica is a printing specification. This is independent of the evaluation of an expression, since evaluation is performed without regard to the print form. For example, reformatting an expression has no effect on the value of the expression itself. In this sense, print forms are referred to as wrappers. The format type specifies how expressions are to be formatted; `FormatValues[f]` specifies the print forms associated with a symbol `f`. A label attached to `Out` (e.g. `//FortranForm`) is used to indicate special formatting associated with an expression.

```
In[1]:= expr = FortranForm[1 + Sin[x]^2]
Out[1]//FortranForm=
1 + Sin(x)**2
```

The result of the previous computation is the expression inside the print form (without `FortranForm` wrapped around it) :

```
In[2]:= %
Out[2]:=
              2
1 + Sin[x]
```

The value of `expr` includes the wrapper:

[1] This document has in fact been typeset with the line width set at 71 characters.

```
In[3]:= expr
Out[3]//FortranForm=
1 + Sin(x)**2
```

Maeder [15] gives an example of the use of `Format` to print arguments as indices and thus mimic subscripted mathematical notation. A list of available print forms may be obtained by issuing the command `$PrintForms`. Other output formats exist, such as `TreeForm[expr]` which returns a tree representation of `expr` with different levels printed at different depths. A list of output forms may be obtained using `$OutputForms`. The formatting functions proved difficult to extend for our purposes and an alternative strategy, outlined in the next section, has been adopted.

3.2 Output as strings

Mathematica can be used to handle pure strings of text. Many facilities for string-conversion and manipulation are provided. It is also possible to use standard programming concepts such as transformation rules.

```
In[4]:= str = ToString[expr];
In[5]:= FullForm[str]
Out[5]//FullForm=
"1 + Sin(x)**2"
```

A string is much more flexible for formatting purposes than an expression. For example, long expressions can be broken into multiple lines at any point, without regard to syntax. `lhs` variables, array indices and assignment operators ("="), can be efficiently attached using `StringJoin`, in order to convert an expression into an assignment. There are also many built-in string manipulation tools available.

```
In[6]:= ToLowerCase[str]
Out[6]= 1 + sin(x)**2
```

When print forms of expressions are converted to strings,the ability to manipulate the evaluated forms (as was possible in section 3.1) is forfeited. The ability to recover expressions using the expression parser `ToExpression` can also be lost. In the following example the expression no longer evaluates to `Power[Sin[x],2]`.

```
In[7]:= ToExpression[str] //FullForm
Out[7]//FullForm=
Plus[1, Times[Sin, NonCommutativeMultiply[x, 2]]]
```

In this way the behaviour of Mathematica's print form wrappers has not been maintained. This is considered a price worth paying in order to provide improved formatting tools.

3.3 Reformatting expressions

Several Mathematica functions have been redefined in order to modify the way that they are formatted. In order to do this a `Block` structure has been used to temporarily set aside current definitions and attributes (such as `Protected`) and assign new definitions. Any redefinitions hold only within the scope of the `Block` and this form of encapsulation mechanism avoids interference with the rest of the system. For example, in some cases it is convenient to attach a rule to a built-in function such as `Power`. In such cases, `Block` ensures that the rule is removed before any result is returned.

3.4 Localised definitions

Rules defined using Mathematica's `Format` hold only in the body of a `Block`. They do not hold for functions executed within the `Block`. This behaviour can be illustrated by the following example:

```
In[1]:= mainformat[expr_,FortranForm]:= FortranForm[expr];
In[2]:= badformat[expr_] :=
          Block[{ArcSin,ASin},
            Format[ArcSin[y_],FortranForm]:= ASin[y];
            mainformat[expr,FortranForm]
         ];
In[3]:= badformat[ArcSin[a+b]]
Out[3]= ArcSin(a + b)
```

The format rule for `ArcSin` does not hold within the body of `mainformat` and is never applied to the function. It is therefore difficult to implement a modular design using this approach which, at the very least, makes code maintenance difficult. A preferred approach is outlined in the next section.

3.5 Renaming functions

It turns out to be efficient and flexible to rename functions by applying a head replacement function. The example below illustrates this. The function `Csc` is redefined in terms of a compiled language equivalent and the result is formatted as a FORTRAN expression. The function `FD` is used for the FORTRAN Definition.

```
In[1]:= mainformat[expr_,FortranForm]:= FortranForm[expr];
In[2]:= myformat[expr_] :=
          Block[{Csc,FD},
            FD[Sin]=sin;
            Csc[x_]:= Evaluate[1/FD[Sin][x]];;
            mainformat[expr,FortranForm]
          ];
```

`Evaluate` is used to avoid head replacement each time the rule is executed, by inserting the redefinition when the rule is created.

```
In[3]:= myformat[y Csc[2 x]]
Out[3]= y/sin(2*x)
```

Using this method, temporary rules defined during execution of the `Assign` functions do not interfere with global definitions. As a cautionary note it is emphasise that this approach may affect the evaluation of expressions. For example, `Csc[2.3]` would not evaluate numerically if `myformat` had the attribute `HoldAll`. Thus numerical approximation (such as using `N`) must be performed before this step.

3.6 Package documentation

Full online documentation for all functions and options is available.

```
In[4]:= ?FortranAssign

FortranAssign[lhs,rhs,options]
 FortranAssign converts the assignment of lhs to rhs into FORTRAN
   compatible strings. Options enable control over the conversion
   process. Expressions are broken up and continuation characters are
   added by default. The precision of real numbers in expressions may
   be specified together with single or double precision exponents.
   Generic FORTRAN functions are used since compilers can infer the
   function type from the precision of the argument. If assignments to
   expressions are not required, the lhs argument may be omitted.
 When used with Splice, the option FormatType->OutputForm should be
   specified.

In[5]:= ?AssignToArray

AssignToArray is used to convert Mathematica arrays and functions into
   arrays in C and FORTRAN. Arguments are protected from N and
   maintained in exact form. AssignToArray may evaluate to any list of
   symbols.
```

A complete list of all Assign related functions and options may be obtained using:

```
In[6]:= ?*Assign*

Out[6]=
Assign              AssignIndent        AssignTemporary
AssignBreak         AssignIndex         AssignToArray
AssignCase          AssignLabel         AssignToFile
```

AssignEnd	AssignMaxSize	AssignTrig
AssignExp	AssignOptimize	AssignZero
AssignFortranNumbers	AssignPrecision	CAssign
AssignFunction	AssignRange	FortranAssign
AssignHyperbolic	AssignReplace	MapleAssign

A complete list of all symbols introduced by the package can be obtained by issuing the command `?@` in a new session after the package has been loaded.

3.7 Parameter and option checking

Mathematica provides a facility for data-typing of function arguments. When a pattern match for an argument fails the unevaluated function is returned.

```
In[4]:= g[x_Integer]:=x^2;
In[5]:= g[2.3]
Out[5]= g[2.3]
```

Since there are many options available in the package, it is preferable to perform parameter type checking for all arguments and options using a conditional test (`/;`). An appropriate error message is then output [16]. This behaviour is demonstrated with an example of incorrect input syntax, showing the close correspondence with Mathematica's own internal error messages.

```
In[6]:= FortranAssign[x,Pi t,AssignIndent->wrongtype]

  FortranAssign::args:
    The option AssignIndent did not evaluate to a string or a
    positive integer.

Out[6]= FortranAssign[x, Pi t, AssignIndent -> wrongtype]
```

The option `AssignIndent` specifies an ASCII string prepended to the output.

4 Implementation

There are many details which need to be considered when translating from one language to another. `f2c` the FORTRAN to C converter from netlib illustrates this [23]. Numerous examples of the limitations and deficiencies of Mathematica's formatting rules are given along with demonstrations of how these are overcome using the `Assign` functions.

4.1 The precedence of operators and operator syntax

Parentheses can be used in order to specify a precedence for the order of evaluation of operations in a computation. For example, the expression `!a==b` parses as `Not[Equal[a,b]]` which is converted to `Unequal[a,b]`. Whereas, `(!a)==b` parses as `Equal[Not[a],b]` which is entirely different. A code translation package is practically useless if it does not produce code that obeys the operator precedence of the target language. Fortunately, most cases are dealt with correctly by Mathematica's format functions, since they simply correspond to the operator precedence in that language. This is not always immediately obvious. For example, what is the order of precedence in the expression `a^b^c`? In fact this is evaluated as `a^(b^c)` and correctly interpreted in FORTRAN without parentheses. Unfortunately, there are cases where Mathematica's syntax for operators does not produce correct target language code.

For example, logical operators can be specified with multiple arguments in Mathematica. `FortranForm` is unable to translate these and returns syntactically incorrect output.

```
In[4]:= FortranForm[ a>b>c ]
Out[4]//FortranForm=
Greater(a,b,c)
```

In contrast, `FortranAssign` knows about such operators and performs some transformations to manipulate the expressions into appropriate form.

```
In[5]:= FortranAssign[ a>b>c ]
Out[5]//OutputForm=
        (a.gt.b).and.(b.gt.c)
```

Some cases are not so obvious and require additional subtlety, because they do not obey the law of transitivity: If $a \to b$ and $b \to c$ then $a \to c$.

```
In[6]:= FortranAssign[a!=b!=c]
Out[6]//OutputForm=
        (a.ne.b).and.(a.ne.c).and.(b.ne.c)
```

In fact there are other examples when translation of relatively simple relational statements is inaccurate. The following example illustrates this for the `Inequality` construct.

```
In[7]:= FortranForm[ a>b<c ]

Out[7]//FortranForm=
Inequality(a,Greater,b,Less,c)
```

Translation of expressions involving `Inequality` is possible with the `Assign` functions.

```
In[8]:= FortranAssign[ a>b<c ]

Out[8]//OutputForm=
        (a.gt.b).and.(b.lt.c)
```

4.2 Rules for code translation

When translating symbolic expressions, certain functions should be restricted according to the type of their arguments (integer, real, double, complex etc.). For example, all the arguments of the FORTRAN function `max` must be of the same type. To ensure ease of use and improved efficiency, the issue of data typing of function arguments has not been addressed. The tools provided are assumed to have been used appropriately.

4.3 Conforming to ANSI standards

Both `FortranAssign` and `CAssign` convert to the full set of compiled language functions according to the American National Standards Institute guidelines. Some examples of translating to compiled language intrinsics are given in this section. `CAssign` and `FortranAssign` automatically test for the presence of non-standard ANSI C and FORTRAN functions. An appropriate warning message is issued the first time a function which does not conform to the standard is encountered.

```
In[4]:= FortranAssign[Sin[f[x]]+f[x]]

AssignFunction::undef:
   Expression contains the function f which is not part of the ANSI
    FORTRAN standard.

Out[4]//OutputForm=
        sin(f(x))+f(x)
```

Only generic functions are used. These are functions which are not restricted by their arguments. For example, `sin` is generic whereas `dsin` is not, since it only accepts double precision arguments.

However, in some cases it is desirable to mix Mathematica with target language code. The ANSI test function also knows about type-specific variants of functions, which are assumed to be in lower case form.

```
In[5]:= FortranAssign[ Cos[x]+dsin[y] ]

Out[5]//OutputForm=
        cos(x)+dsin(y)
```

Examples in C include functions such as `fabs` and `fmod` which are real argument versions of `abs` and `mod`.

This concept can be taken further to produce code fragments. This example uses the FORTRAN function dim(x,y), which returns the maximum of `x-y` and `0`. First the ANSI compatibility message is suppressed because programming language constructs are not implemented in the test.

```
In[6]:= Off[AssignFunction::undef];
In[7]:= FortranAssign[ if[dim[x,y]<10,y,x] ]

Out[7]//OutputForm=
        if(dim(x,y).lt.1.d1,y,x)
```

The user should take care when using Mathematica constructs with the `Hold` attribute.

4.4 Utilising intrinsic compiled language functions

It is desirable to convert Mathematica functions into intrinsic (built-in) compiled language functions whenever possible. The reason for this adherence is not merely syntactic. In many cases the code produced is more efficient.

Mathematica converts `Exp[x]` to `Power[E,x]` which is then formatted in a way that does make use of the FORTRAN `exp` function. It is inefficient to calculate the exponential of a function as a power of a constant `E` in this way.

```
In[4]:= FortranForm[Exp[x]]

Out[4]//FortranForm= E**x
```

FortranAssign converts to the exponential function in FORTRAN.

```
In[5]:= FortranAssign[x Exp[y],AssignIndent->""]

Out[5]//OutputForm= x*exp(y)
```

Consider the inverse trigonometric and inverse hyperbolic family of functions.

```
In[6]:= FortranForm[ArcSin[x]]

Out[6]//FortranForm= ArcSin(x)
```

`ArcSin` is not a legitimate intrinsic FORTRAN function. The corresponding FORTRAN function is `asin`.

```
In[7]:= FortranAssign[ArcSin[x],AssignIndent->""]

Out[7]//OutputForm= asin(x)
```

Some related functions have no compiled language analogy.

```
In[8]:= FortranForm[ArcCot[x]]

Out[8]//OutputForm= ArcCot(x)
```

ArcCot is not a legitimate FORTRAN function (and neither is acot). FortranAssign converts such functions by default, controlled via the option AssignTrig.

```
In[9]:= FortranAssign[ArcCot[x],AssignIndent->""]

Out[9]//OutputForm= atan(1.d0/x)
```

Notice that ArcCot is not uniquely defined at $x = 0$, despite the fact that Mathematica returns the result $Pi/2$ (perhaps it should return $\pm Pi/2$). There is therefore no loss of generality in using the above definition. The handling of branch cuts and choice of principal values is a common criticism of computer algebra systems [33]. If principal values are not used in compiled languages, substantial loss of precision can result.

The existence of inverse hyperbolic functions is compiler dependent since they are not part of the ANSI standard. Some compilers provide intrinsic functions for these. When the compiler does not support such functions, an option for rewriting in terms of Log and Sqrt is provided which return the unique principal value.

```
In[10]:= FortranAssign[ArcTanh[x],AssignHyperbolic->True]

Out[10]//OutputForm=
        5.d-1*log((1.d0+x)/(1.d0-x))
```

Moreover, Mathematica's representation of the two argument function ArcTan is somewhat unusual in reversing the standard convention.

```
In[11]:= FortranAssign[ArcTan[y,x]]

Out[11]//OutputForm=
        atan2(x,y)
```

In summary the options AssignTrig and AssignHyperbolic may be used to format trigonometric and hyperbolic functions, which may not be supported by an ANSI C or FORTRAN conforming compiler. A method of quantifying the magnification of errors in expression sequences is the condition number [25]. Transformations such as those performed above should also be justified on this basis.

4.5 Maintaining appropriate precision

FortranForm will not always produce code of the correct precision, typically only 6 digits are printed. This behaviour was improved in version 2.2 where floats were (sometimes unnecessarily) printed to full machine precision. However, some compilers do not keep the correct precision unless floats are specified with single or double precision exponents.

```
In[4]:= FortranForm[100 Pi t //N]

Out[4]//FortranForm= 314.159*t
```

The default setting for AssignPrecision produces $MachinePrecision digit, double precision C and FORTRAN code for efficiency. In FortranAssign, floating point numbers are coerced into exponent form by default, using an appropriate exponent which is determined by the requested precision.

```
In[5]:= FortranAssign[100 Pi t, AssignIndent->""]

Out[5]//OutputForm= 3.141592653589793239d2*t
```

If the requested precision is less than or equal to 8 digits, then single precision floating point numbers are output.

```
In[6]:= FortranAssign[100 Pi t, AssignPrecision->8,
          AssignIndent->""]

Out[6]//OutputForm= 3.14159265e2*t
```

Considerable time may be required to coerce numbers into exponent form. The option AssignFortranNumbers can be used to suppress this form if it is not required by the compiler.

Constants and function arguments can be maintained in exact form if required. This is advantageous if, for example, a symbol occurs many times - it need only be assigned a value once at the beginning of a file. This is also useful if integer arithmetic is required.

```
In[7]:= FortranAssign[100 Pi t, AssignPrecision->Infinity,
          AssignIndent->""]

Out[7]//OutputForm= 100*Pi*t
```

As demonstrated, when converting symbolic expressions (a variant of) the function N is applied to convert all numerical constants to reals, with the precision controlled by the option AssignPrecision. Evaluation can be delayed until the code is compiled and executed, by specifying analogous lower case functions (these are all documented and declared in the package).

```
In[8]:= FortranAssign[sin[1],AssignIndent->""]

Out[8]//OutputForm= sin(1.d0)
```

Function types can be inferred from the precision of the argument. Thus, explicit use of the dsin/DSIN function is unnecessary. Care should be exercised when using coercion in an expression containing a mixture of single and double precision values. In such cases, the appropriate precision may not be maintained.

This section is concluded with some issues concerning the range of floating point and integer numbers. Mathematica utilises so-called arbitrary precision (or BigNum) arithmetic. Therefore some numbers contained in symbolic expressions may not be admissible

in a corresponding compiled language statement. The option `AssignRange` has been provided to test the range of numbers occurring in the input. Single or double precision real and float ranges are selected for the test according to the setting for `AssignPrecision`. For details of the IEEE standards see [25, 31] (the former reference contains a useful section on rounding strategies, while the latter contains an interesting section on machine representation of numbers). A comprehensive survey of the pitfals of floating point arithmetic is given in [9].
Integer numbers which are outside the permissible range are coerced to floats.

```
In[9]:= FortranAssign[10^20 t,AssignPrecision->Infinity,
          AssignRange->True]

AssignRange::integer:
                                                                            63
   Expression contains integers outside the permissible range -2
         63
     to 2   - 1 which cannot be represented in IEEE double
     precision and have been converted to floating point numbers.

Out[9]//OutputForm= 1.d20*t
```

A warning message is output if floating point numbers lie outside the permissible range.

```
In[10]:= FortranAssign[10.^40 t,AssignPrecision->8,AssignRange->True]

AssignRange::float:
   Expression contains machine numbers outside the permissible range
              -38              38
    1.17549 10    to 1.70141 10   for IEEE single precision.

Out[10]//OutputForm= 1.e40*t
```

4.6 Converting powers

`FortranForm` incorrectly maintains fractional powers as rationals - these will be truncated by a compiler.

```
In[4]:= FortranForm[x^(-2/7)]

Out[4]//FortranForm= x**(-2/7)
```

Using `FortranAssign` most rational powers are converted to floating point numbers.

```
In[5]:= FortranAssign[x^(-2/7),AssignIndent->""]

Out[5]//OutputForm= x**(-2.d0/7.d0)
```

This representation is generally more compact than carrying out the division, which will be handled by a compiler when creating the executable file. Where possible, Mathematica functions are converted to compiled language functions (see section 4.2). An example is the square root function in FORTRAN.

```
In[6]:= FortranAssign[y x^(-1/2),AssignIndent->""]

Out[6]//OutputForm= y/sqrt(x)
```

The only exception to the conversion of rational exponents to floats concerns expressions of the form $x^{\pm 1/2}$. Expressions generally translate into more lines of code, but this is outweighed by improvements in performance.

Unlike C, FORTRAN compilers recognise integer powers and use special heuristics to evaluate them more efficiently than floats. Therefore `FortranAssign` maintains integer powers as integers.

```
In[7]:= FortranAssign[x^7,AssignIndent->""]

Out[7]//OutputForm= x**7
```

A strategy for efficient factorisation of powers is implemented in a separate package for peforming optimizations (see section 5 and [29]).

4.7 Array arguments and the negative sign

There is nothing to distinguish between arrays and functions in Mathematica. Array and function arguments may therefore not be handled appropriately when translating expressions.

On an alternative note, the internal representation of a symbolic expression is often translated verbatim and this can lead to inefficiencies. An example is negation. Mathematica stores the expression `-x` internally as `Times[-1,x]`. The following expression is translated into appropriate form in FORTRAN.

```
In[4]:= FortranForm[ -x ]
Out[4]//FortranForm= -x
```

However, the function `N` is often applied to obtain numerical approximants, and convert constants, often simplifies results. This is not accounted for by Mathematica's formatting functions.

```
In[5]:= FortranForm[ N[a[1] - b[1]] ]

Out[5]//FortranForm= a(1.) - 1.*b(1.)
```

Two issues are raised here. There is a need for some method of protecting array or function arguments from numerical approximation. We also have to overcome the display of `N[-x]` as `-1.*x`, which results in unnecessary multiplication. A partial fix is to explicitly use `NProtectedAll` to protect specified arguments from `N`.

```
In[6]:= SetAttributes[{a,b},NProtectedAll];

In[7]:= FortranForm[a[1] - b[1]] //N

Out[7]//FortranForm= a(1) - 1.*b(1)
```

These attributes remain unless they are cleared.

```
In[8]:= Attributes[{a,b}]

Out[8]= {{NProtectedAll}, {NProtectedAll}}
```

Array conversion in the `Assign` functions is handled using the option `AssignToArray`. Conversion to the negative sign is accomplished by using a pattern matching rule to recover expressions of the form `Times[-1.,_]`.

```
In[9]:= ClearAttributes[{a,b},NProtectedAll];

In[10]:= FortranAssign[a[1] - b[1], AssignToArray->{a,b},
         AssignIndent->""]

Out[10]//OutputForm= a(1)-b(1)
```

Any attributes assigned by `Assign` functions are removed when execution is complete.

```
In[11]:= Attributes[{a,b}]

Out[11]= {{}, {}}
```

As illustrated above, when an array is specified explicitly using `AssignToArray`, it is exempt from the test for ANSI compatible functions.

4.8 String manipulation

Since expressions are converted to strings, various options are provided for for string manipulation purposes.

4.8.1 String replacement

Character replacement is possible using standard transformation rules. As an example, output strings can be compacted using `AssignReplace->{" "->""}` and this is the default for `FortranAssign`. The advantage is a reduction in the number of lines of code needed to represent an expression, which makes line-breaking more efficient. This technique can be used to substitute for other strings, such as to replace generic function types.

```
In[4]:= FortranAssign[Sin[x],AssignReplace->{"sin"->"dsin"}]

Out[4]//OutputForm=
        dsin(x)
```

The replacement is literal in the sense that exact character sequences are substituted and should be used with caution. In this example an ANSI FORTRAN function is not obtained.

```
In[5]:= FortranAssign[ArcSin[x],AssignReplace->{"sin"->"dsin"}]

Out[5]//OutputForm=
        adsin(x)
```

4.8.2 Upper and lower case

Case conversion is used for translating expressions to lower case in C (see section 4.11). Also, certain FORTRAN compilers (specifically MSDOS) require that all characters be specified in upper case.

```
In[4]:= FortranAssign[y,Sin[2 Cos[x + Log[y]]],AssignCase->Upper]

Out[4]//OutputForm=
          Y=SIN(2.D0*COS(X+LOG(Y)))
```

4.8.3 Controlling line breaking

Some editors (such as vi in UNIX) limit the number of lines a single expression may occupy. It is therefore useful to be able to specify how to break long expressions explicitly. This option may also be useful when cutting and pasting expressions into different sized windows.

```
In[4]:= expr = Expand[(w-Sin[x]+y+Exp[z])^2];

In[5]:= FortranAssign[expr,expr,AssignBreak->{45,"\n     &  "}]

Out[5]//OutputForm=
```

```
        expr=w**2+2.*w*y+y**2+2.*w*exp(z)+2.*
     &  y*exp(z)+exp(2.*z)-2.*w*sin(x)-2.*y*s
     &  in(x)-2.*exp(z)*sin(x)+sin(x)**2
```

ANSI standard conforming FORTRAN code occupies columns 7 to 72 (the others are reserved for special tokens such as continuation characters and statement labels). As already mentioned, the default output of `FortranAssign` uses columns 8 to 72 since this aligns with the tab character, commonly used in programs to save on file space. Many modern compilers allow variants on line formatting which are not part of the FORTRAN standard. In such cases, the default setting can be modified using the `AssignBreak` option as above.

4.9 Assignments to expressions

Mathematica's output formats have only been implemented to translate expressions. As a result, it is not possible to produce assignment statements. `Assign` and related functions enable four categories of assignments.

1. Translation of an expression or list of expressions (`lhs` omitted or `Null` assignment).

2. Assignment of an expression to a symbol (or string).

3. Array assignment. Assignment of a list of expressions (`rhs`) to a symbol (`lhs`) resulting in a sequence of assignments to an array. The array index is determined by the shape of the array.

4. Assignment of a list of expressions to a list of symbols - the `lhs` and `rhs` lists must have the same length, but can contain a mixture of lists and symbols.

If the `lhs` is a list, it may not contain sublists (it must pass the test `VectorQ`).
The following example illustrates the inability of Mathematica's output forms to translate assignments.

```
In[4]:= expans = Expand[(b+c)^6];

In[5]:= FortranForm[a=expans]

Out[5]//FortranForm=
b**6 + 6*b**5*c + 15*b**4*c**2 + 20*b**3*c**3 + 15*b**2*c**4 +
     -  6*b*c**5 + c**6
```

The result of the previous computation evaluates the expression `expans` and sets the symbol `a` to the result. In contrast what is required is to produce a translated assignment statement in FORTRAN without affecting the definition of the symbol. Other attempts, such as `FortranForm[HoldForm[a=b]]` also fail to give the desired result. Before proceeding, the previous definition must be cleared.

```
In[6]:= a=.;
```

By specifying a `lhs` argument to our `Assign` function the desired assignment can be obtained.

```
In[7]:= FortranAssign[a,expans]

Out[7]//OutputForm=
        a=b**6+6.d0*b**5*c+1.5d1*b**4*c**2+2.d1*b**3*c**3+1.5d1*b**2*c*
     &  *4+6.d0*b*c**5+c**6
```

Since `FortranAssign` has attribute `HoldFirst`, the definition of `expans` does not interfere with the assignment in the following example.

```
In[8]:= FortranAssign[expans,expans]

Out[8]//OutputForm=
        expans=b**6+6.*b**5*c+15.*b**4*c**2+20.*b**3*c**3+15.*b**2*c*
     &  *4+6.*b*c**5+c**6
```

This enables us to use `lhs` names which would normally evaluate to expressions and maintain closer correspondence with our Mathematica definitions.
Another deficiency of `FortranForm` is that it does not always produce the desired results when applied to lists (in this case, no continuation lines).

```
In[9]:= Map[ FortranForm, {expans,expans} ]

Out[9]//FortranForm=
{b**6 + 6*b**5*c + 15*b**4*c**2 + 20*b**3*c**3 + 15*b**2*c**4 +
   6*b*c**5 + c**6, b**6 + 6*b**5*c + 15*b**4*c**2 + 20*b**3*c**3 +
   15*b**2*c**4 + 6*b*c**5 + c**6}

In[10]:= FortranForm[{expans,expans}]

Out[10]//FortranForm=
List(b**6 + 6*b**5*c + 15*b**4*c**2 + 20*b**3*c**3 + 15*b**2*c**4 +
     -   6*b*c**5 + c**6,b**6 + 6*b**5*c + 15*b**4*c**2 +
     -   20*b**3*c**3 + 15*b**2*c**4 + 6*b*c**5 + c**6)
```

With `FortranAssign` a sequence of array elements are assigned to when the `lhs` is a symbol or string and the `rhs` is a list.

```
In[11]:= FortranAssign[a,{expans,expans}]

Out[11]//OutputForm=
```

```
        a(1)=b**6+6.d0*b**5*c+1.5d1*b**4*c**2+2.d1*b**3*c**3+1.5d1*b**2
     &  *c**4+6.d0*b*c**5+c**6
        a(2)=b**6+6.d0*b**5*c+1.5d1*b**4*c**2+2.d1*b**3*c**3+1.5d1*b**2
     &  *c**4+6.d0*b*c**5+c**6
```

Alternatively, different symbols can be assigned in a single statement, by constructing a list of `lhs` values.

```
In[12]:= lhs = {a1,a2};

In[13]:= rhs = {b1,b2};

In[14]:= FortranAssign[Evaluate[lhs],rhs]

Out[14]//OutputForm=
        a1=b1
        a2=b2
```

In this way, a mixture of lists and symbols can be assigned to. The `Assign` functions automatically detect the type of the assigned object and generate an appropriate `lhs`, including array arguments if necessary. In this way blocks of assignments can be collected and issued simultaneously. Block assignments are more efficient than a sequence of single assignments since rewrite rules are generated only once and option type tests do not need to be repeated.

```
In[15]:= lhs = {a,b,c};

In[16]:= rhs = {d,{e1,e2},f};

In[17]:= FortranAssign[Evaluate[lhs],rhs]

Out[17]//OutputForm=
        a=d
        b(1)=e1
        b(2)=e2
        c=f
```

Any shape list can be assigned and the assignment variable assumes the appropriate index. Labels can also be attached to expressions along with terminating strings. Thus expressions can be separated by a carriage return as `AssignEnd->"\n"`.

```
In[18]:= FortranAssign[a,{{ArcCos[w],x^(1/2)},{Exp[y],Log[z]}},
         AssignLabel->55]

Out[18]//OutputForm=
```

```
55      a(1,1)=acos(w)
        a(1,2)=sqrt(x)
        a(2,1)=exp(y)
        a(2,2)=log(z)
```

Individual array elements can be assigned to by using the list assignment form.

```
In[19]:= FortranAssign[{a[2,3],a[3,2]},{b,c}]

Out[19]//OutputForm=
        a(2,3)=b
        a(3,2)=c
```

The `lhs` may also be specified as a string or list of strings.

```
In[20]:= FortranAssign["a(2,1)",b+c d]

Out[20]//OutputForm=
        a(2,1)=b+c*d
```

4.10 FORTRAN compiler limitations

Many FORTRAN compilers impose a limit on the number of continuation lines a given expression may contain. There are two limitations of Mathematica's `FortranForm` in this sense. The first is the way that `FortranForm` breaks expressions into multiple lines. `FortranForm` attempts to format aesthetically by avoiding breaking sub-expressions where possible, thus producing many more lines of code than is necessary. Compilers do not impose restrictions on where expressions are broken (other than the ANSI standard formatting in columns 7 to 72).[2] Secondly, `FortranForm` imposes no limit on the number of lines of code produced. Typically a restriction of 19 continuation lines by a compiler makes `FortranForm` useless for translating large expressions.

Code has been developed to automatically extract sub-expressions and introduce temporary variables when expressions exceed a specified tolerance. The syntax of the original expression is maintained, by ensuring only syntactically correct sub-expressions are extracted. This was the most difficult feature to implement. After attempting several different strategies, the following approach was adopted because of its efficiency, whilst at the same time it provides a degree of user control.

Sub-expression extraction is performed recursively. A user specified tolerance is input using the option `AssignMaxSize`. This relates to the maximum number of bytes in memory a sub-expression may occupy. The test is based on Mathematica's `ByteCount`, because evaluation is very efficient. Starting at the root of the expression tree, each level

[2] A lot of modern compilers allow more freedom over the formatting of expressions using compiler flags.

in an expression is traversed. Branches in the expression tree are tested to see if they exceed bounds. If the bound is exceeded, a sub-expression within tolerance is found (and extracted) by recursing further down the expression tree. The sub-expression is assigned to a new temporary variable. The position occupied by the sub-expression is replaced by the temporary variable. This process is repeated until the expression is fully traversed and what remains is within bounds.

The test used to estimate the size of sub-expressions is similar to the current strategy implemented by M. Monagan in Maple [17], however, the recursive extraction is quite different. There are several deficiencies in `ByteCount` for our purposes, which should be mentioned. Firstly, it takes no account of the length of a variable name.

```
In[4]:= Map[ByteCount,{a,aa,aaa}]

Out[4]= {0, 0, 0}
```

Secondly, `ByteCount` resolves very little information concerning numbers and their precision.

```
In[5]:= Map[ByteCount,{1, 123456789, 123.456}]

Out[5]= {12, 12, 20}
```

Whilst the strategy based upon `ByteCount` is only a rough heuristic, it works well in practise and is justified on this basis. Before proceeding with an example, it seems appropriate to explain the above behaviour by giving some more detailed information concerning how `ByteCount` is implemented.

`ByteCount[<machine integer>]` for machines with 32-bit integers returns 4 bytes plus an 8 byte expression overhead. `ByteCount[<machine real>]` returns the 8 byte expression overhead plus the memory used for a machine real plus whatever is needed for memory alignment. On computers with 64 bit reals, this works out as 16 bytes. On computers like the Apple Macintosh which typically use 80 bit reals and 4 byte memory alignment, the result is 20 bytes.

`ByteCount[<symbol>]` always returns zero. The reasoning behind this is that symbols are always shared, so multiple copies of the same symbol in an expression are not counted twice. As was illustrated above, they aren't even counted once. Allowance for shared expressions disagrees with the documentation for `ByteCount` [35].

The overhead for a normal expression is essentially 12 bytes. The rest of the memory is normally 4-byte pointers for the head and each of the elements. Since symbols are taken to use zero memory, `ByteCount[f[]]` uses 16 bytes, `ByteCount[f[x]]` uses 20, and `ByteCount[f[x,y]]` uses 24.

A more complicated example is `f[f[x],f[x]]`, which consists of an outer normal expression with two elements. The overhead of this expression is 12 bytes, plus a 4 byte pointer for the head and each of the two elements, yielding a total of 24 bytes. The first element, $f[x]$, is a normal expression with one element, and uses 20 bytes, as does the second element. The total for the entire expression is 64 bytes.

Some comments also need to be made concerning the temporary variable introduced during recursive decomposition of an expression. The option `AssignTemporary` specifies a symbol or string name `temp` for the temporary assignment variable introduced during sub-expression extraction. The temporary variables are introduced sequentially as `temp1`, `temp2`, ... etc. The user should take care not to assign to symbols with the same name.

In FORTRAN, `implicit` data statement declarations may be used to avoid individual variable type declarations. For example when data typing a FORTRAN program, the code below would be preceeded by a declaration such as:

```
implicit double precision(a-h,o-z)
```

The following example illustrates the use of the auto-extraction code.

```
In[6]:= example = {Sin[Expand[(a+Exp[(b-c-d)])^3]]};

In[7]:= FortranAssign[x,example,AssignMaxSize->400]

Out[7]//OutputForm=
        t1=a**3+exp(3.d0*b-3.d0*c-3.d0*d)
        t2=3.d0*a*exp(2.d0*b-2.d0*c-2.d0*d)
        t3=3.d0*a**2*exp(b-c-d)
        x(1)=sin(t1+t2+t3)
```

The default setting of `AssignMaxSize` should be adjusted by the user if overly-long statements are produced. Experimentation with `ByteCount` (utilising the previous information) should give a more precise indication. There is a minimum setting of 200, which prevents the user-specification of unattainable bounds.

4.11 Formatting C Expressions

A function `CAssign` has also been implemented for producing assignments in C. Code resulting from Mathematica's `CForm` requires pre-processor statement, `#include mdefs.h` and the related include file to be accessible. If this statement is not specified, the compiler will be unable to correctly interpret the code produced by Mathematica. The include file `mdefs.h` is currently only distributed with Unix versions of Mathematica. `CAssign` makes the file redundant and adds many extra features.

Array indices in C start from zero. The option `AssignIndex` may be used to modify the starting index of arrays. Here is the default behaviour.

```
In[4]:= CAssign[a,{{x1,x2},{x3,x4}}]

Out[4]//OutputForm=
a[0][0]=x1;
a[0][1]=x2;
a[1][0]=x3;
a[1][1]=x4;
```

Each statement is terminated using a semi-colon via the option `AssignEnd`.
Powers in C are converted according to the following rules:

1. Rational powers $x^{1/2}$ and $x^{-1/2}$ are converted to $sqrt(x)$ and $1/sqrt(x)$ respectively.
2. Remaining rational exponents and all integer exponents and are converted to floats.

The following example illustrates these.

```
In[5]:= CAssign[x^5-1/Sqrt[y]+z^(3/2)]

Out[5]//OutputForm= pow(x,5.)+pow(z,3./2.)-1./sqrt(y);
```

An array in Mathematica `CForm` can thus be incorrectly converted into a C function.

```
In[6]:= expr = Expand[(ArcCos[a[1]]-Sin[b[1]])^2];

In[7]:= CForm[expr]

Out[7]//CForm=
Power(ArcCos(a(1)),2) - 2*ArcCos(a(1))*Sin(b(1)) + Power(Sin(b(1)),2)
```

`CForm` actually converts `Part` to an array. In this example, the list `a` needs to be wrapped in `HoldForm` to prevent an evaluation warning message.

```
In[8]:= CForm[ HoldForm[ a[[3,2]] ] ]

Out[8]//CForm= a[3][2]

In[9]:= CForm[ a[[3,2]] ]

Part::partd: Part specification a[[3,2]]
     is longer than depth of object.

Out[9]//CForm= a[3][2]
```

What is actually required is to convert Mathematica arrays to C arrays. Since there is no distinction for arrays in Mathematica, it is not possible to perform this conversion automatically. Instead this is left up to the user via the option `AssignToArray`.

```
In[10]:= CAssign[expr,expr,AssignToArray->{a,b}]

Out[10]//OutputForm=
expr=pow(acos(a[1]),2.)+pow(sin(b[1]),2.)-2.*acos(a[1])*sin(b[1]);
```

Notice the line continuation character which is added.
Since the `lhs` may be specified as a string, assignment to array elements in C is possible.

```
In[11]:= CAssign["a[2][1]",Exp[b+ArcTan[c]]]

Out[11]//OutputForm=
a[2][1]=exp(b+atan(c));
```

When arrays are initialised in C, array elements that are not formally defined are defaulted to zero by a compiler [10]. Therefore, for the purposes of efficiency, it is desirable to remove unnecessary assignments and have them assigned by the compiler.

```
In[12]:= CAssign[a,{{0,b},{c,0}},AssignZero->False]

Out[12]//OutputForm=
a[0][1]=b;
a[1][0]=c;
```

Notice how the array indices correspond to the input.
There is a potential conflict with this option, when all assignments are zero-valued. In such a case, assignments to zeros are performed and an appropriate warning message is output.

```
In[13]:= CAssign[a,{0,0},AssignZero->False]

AssignZero::continue:
   Expression encountered with no non-zero elements. Continuing with
     zero assignments.

Out[13]//OutputForm=
a[0]=0;
a[1]=0;
```

In some cases, C statement delimiters may not be required. For example if an expression is translated for use inside an `if` statement.

```
In[14]:= CAssign[a<=6||a>=-6,AssignEnd->""]

Out[14]//OutputForm=
a<=6.||a>=-6.
```

When writing C code, it is often useful to control the precision of data types in the following manner.

- Define all float variables as double throughout.

- Conversion to single precision is possible by including the pre-processor definition, `#define double float`, in the header of the C source file. Mathematica may be used to choose whether to specify this if the `Splice` form of communication is used.
- Floating point math library functions in C are implemented in double precision. In ANSI C there are compiler options to ensure that functions accept and return arguments of type float. This may greatly improve efficiency for numerical computations, if the extra precision is not required.

Some other points of note. The default Mathematica exponentiation is used for floating point numbers. Unlike FORTRAN, C does not possess complex data types. A `complex` structure must be declared in order to extend library functions (see section 5.6 of [10]). Code is converted to lower case by default as described in sub-section 4.8.

4.12 Conversion to Maple

A basic expression translator from Mathematica to Maple has been implemented. The `MapleAssign` command works slightly differently from the other `Assign`-related functions in that the `rhs` is maintained in unevaluated form. The following example illustrates how lists and matrices (lists of lists) of expressions can be converted.

```
In[4]:= MapleAssign[ z, {NSolve[x^7+x^5-x==0,x],
          Map[f,Sin[x^2] y CosIntegral[2]]}]

Out[4]//OutputForm=
z:=[fsolve(-x+x^5+x^7=0,x),map(f,y*Ci(2)*sin(x^2))];
```

In contrast, all other `Assign` functions output a sequence of statements, with `lhs` indices in correspondence with the shape of the list.

Most Mathematica operators are translated by `MapleAssign` without evaluation. However, it is convenient to allow some basic arithmetic operations to be performed during the conversion process. The operators `Plus`, `Power` and `Times` are evaluated during conversion. Consequently some numeric operations are performed.

```
In[5]:= MapleAssign[ intex, Integrate[Sin[x^2 + 1/3 - 2/11 ],x] ]

Out[5]//OutputForm=
intex:=int(sin(5/33 + x^2),x);
```

Some more complicated, but commonly used, functions have also been implemented.

```
In[6]:= MapleAssign[ diffex, D[AiryAi[1.] Sin[x y],{x,2},y] ]

Out[6]//OutputForm=
diffex := diff(diff(Ai(1.)*sin(x*y),x$2),y);
```

A few Maple library procedures have also been implemented. These must be read in using `with(libname)`, before execution in Maple. Because of syntactical differences between the two languages, some decisions need to be made during conversion. In the following example the higher of the two truncation orders specified in `Series`, is passed to the Maple equivalent `mtaylor`.

```
In[7]:= MapleAssign[seriesex,Series[Sin[x y],{x,1,3},{y,2,2}]]

Out[7]//OutputForm=
seriesex := mtaylor(sin(x*y),[x=1,y=2],3);
```

Substitutions analogous to Mathematica's replacement rules can also be performed in Maple.

```
In[8]:= MapleAssign[y,f[g,h] /. {g->1,h->2}]

Out[8]//OutputForm=
y:=subs([g=1,h=2],f(g,h));
```

`MapleAssign` is a less complete implementation than the related `Assign` functions. It is useful for comparison purposes and cross-checking of results. While computer algebra systems contain bugs, it is far less likely that the same bug exists in two different systems. Almost all of the basic functions in Maple (those not utilising external libraries) have direct equivalents in Mathematica. These have been implemented. User-defined procedures and functions are not yet supported.

The command `MapleAssign` also illustrates the difference in syntax between Mathematica and Maple. The design of the programming languages and differences in the evaluation processes, make it questionable as to whether more extensive developments are worthwhile.

4.13 Other Assign forms

Other types of Mathematica output formats may be specified as arguments to the `Assign` function. The following shows how TEX expressions may be produced:

```
In[4]:= Assign[ "\alpha", x Log[Subscripted[y[i]]], TeXForm ]

Out[4]//OutputForm=
\alpha = x\,\log (y_{i})
```

Which prints as $\alpha = x\,\log(y_i)$. This facility can be particularly useful for producing TEX and LATEX documents from Mathematica notebooks. Smith suggests establishing template TEX and LATEX files for use with `Splice` [26]. The Mathematica input and output lines are evaluated and placed at appropriate points in the text, thus ensuring accurate proofs. In LATEX the `inverbatim` environment is especially useful and has been used throughout the preparation of this document.

4.14 Interactive output

Interactive output can be used in a session to send results to files. A change in definition followed by re-execution results in modified contents of the file. Here the text file `temp.f` is created and the contents of the file are displayed.

```
In[4]:= example1 = {Expand[(a+b-c-d)^2],
                   Sin[Expand[(a+Exp[(b-c-d)])^3]]};
In[5]:= channel = OpenWrite["temp.f",FormatType->OutputForm];
In[6]:= Write[channel,FortranAssign[a,example1]];
In[7]:= Close[channel];

In[8]:= !!temp.f

        a(1)=a**2+2.*a*b+b**2-2.*a*c-2.*b*c+c**2-2.*a*d-2.*b*d+2.*c*d
     &  +d**2
        a(2)=sin(a**3+exp(3.*b-3.*c-3.*d)+3.*a*exp(2.*b-2.*c-2.*d)+3.
     &  *a**2*exp(b-c-d))
```

The process of writing to an external file has been simplified, by inclusion of the option `AssignToFile`.

```
In[9]:= FortranAssign[a,example1,AssignToFile->"temp.f"];

In[10]:= !!temp.f

        a(1)=a**2+2.*a*b+b**2-2.*a*c-2.*b*c+c**2-2.*a*d-2.*b*d+2.*c*d
     &  +d**2
        a(2)=sin(a**3+exp(3.*b-3.*c-3.*d)+3.*a*exp(2.*b-2.*c-2.*d)+3.
     &  *a**2*exp(b-c-d))
```

As has already been mentioned, the output produced by the `Assign` functions is a sequence of strings. The strings are wrapped in `OutputForm`, which avoids printing of string delimiters (") for example. Cutting and pasting of results using an editor is then straightforward. This process also facilitates the use of the `Copy Output From Above` command when using the `NoteBook` interface.

5 Optimized computational sequences

The package `Optimize.m` performs syntactic code optimization on Mathematica expressions [29, 30]. The term 'optimization' is used in the sense of lowering the overall computational cost and not the best possible. Identical common sub-expressions are matched in linear time (O(n) operations) and the strategy is thus very efficient for large expressions. For example, the `x+y` in `(x+y)*f[x+y]` is matched, but not in `x+y+f[x+y]`. Some

heuristics such as extraction of numeric coefficients are also performed (see [29] for a fuller discussion).
The following operations are distinguished.

- Specified operators to be ignored during the optimization process, controlled using the option `OptimizeNull`.
- Sub-expressions with head `Power` (option `OptimizePower`).
- Sub-expressions with head `Plus` (option `OptimizePlus`).
- Sub-expressions with head `Times` (option `OptimizeTimes`).
- Remaining functions (option `OptimizeFunction`).

Additionally the option `OptimizeCoefficients` can be used to include numerical coefficients in `Plus` and `Times` as candidates for optimization (the default setting does not to do this since it is less efficient). Distinguishing the above operations enables more efficient optimization of certain types of expressions, by allowing the user some control over the optimization process. The example below illustrates several additional issues. Options of `Optimize` may be passed to `CAssign` and `FortranAssign`. Efficient factoring of exponents in binary form is also possible.

```
In[4]:= <<Optimize`;
In[5]:= expr = Tanh[b^5] Sin[a^2] + Tanh[1/b^3] Cos[a^4]/a^5;
In[6]:= FortranAssign[expr,expr,AssignOptimize->True,
          OptimizePower->Binary]
Out[6]//OutputForm=
        o1=a**2
        o2=o1**2
        o3=b**2
        o4=o6**2
        expr=sin(o1)*tanh(b*o4)+cos(o2)*tanh(1/(b*o3))/(a*o2)
```

The example also demonstrates the relation between reciprocals and powers (`1/x^k` and `x^k`) which share the same set of binary factorisations. If `Optimize` related messages are generated, or optimization fails for some reason, code translation proceeds using the original (unoptimized) expression.

When expressions are polynomials, special factorisations can yield more efficient evaluation. `PolynomialQ[poly,var]` is a logical test in Mathematica for verification that the expression `poly` is a polynomial in the variable `var`. Horner form is the most efficient form of factorisation, when issues of numerical stability are not considered. The factorisation involves n additions and n multiplications for a polynomial of degree n. The procedure `Horner` is provided in the package `Optimize.m` for factoring uni-variate and multi-variate polynomials in Horner form. The following example illustrates that when the coefficients are numeric, the routine is able to determine the variable(s) to be factored.

```
In[5]:= Horner[ 3 x^2 + 2 x - 1]
Out[5]= -1 + x (2 + 3 x)
```

The routine is able to determine the variables (using `Variables[poly]`) whenever the coefficients are numeric. The default ordering is lexicographic, but the user can over-ride this by specifying the variable list explicitly. Because of the highly recursive nature of the nesting of factors, Horner form yields a fairly severe test for code translation routines. The following example shows the behaviour of Mathematica's output formatter, simulating a nested Horner form using `Fold`.

```
In[6]:= nestpoly = Fold[ (#2 + x #1)&, 0 , Range[20] ]
Out[6]=
20 + x (19 + x (18 + x (17 +
           x (16 + x (15 + x
                  (14 + x (13 +
                      x (12 +
                        x (11 +
                        x (10 +
                        x (9 +
                        x (8 +
                        x (7 +
                        x (6 + x (5 + x (4 + x (3 + x (2 + x))))))))))
                      )))))))
```

The output is not space-efficient. Furthermore, `CForm` and `FortranForm` exhibit similar behaviour since they utilise the same internal code. In contrast, our implementation is able to deal with such difficult examples, even when sub-expression extraction is requested.

```
In[7]:= FortranAssign[nestpoly, nestpoly, AssignMaxSize->200]
Out[7]//OutputForm=
        t1=4.d0+x*(3.d0+x*(2.d0+x))
        t2=x*(6.d0+x*(5.d0+t1*x))
        t3=9.d0+x*(8.d0+(7.d0+t2)*x)
        t4=x*(1.1d1+x*(1.d1+t3*x))
        t5=1.4d1+x*(1.3d1+(1.2d1+t4)*x)
        t6=x*(1.6d1+x*(1.5d1+t5*x))
        t7=1.9d1+x*(1.8d1+(1.7d1+t6)*x)
        nestpoly=2.d1+t7*x
```

Further examples of the use of the code optimization option are provided in the applications section 6.

6 Splice and template files

The use of `FortranAssign` is illustrated with some examples of template files and code generation using `Splice`. Firstly, an overview of the concepts involved is given using some relatively simple examples.

6.1 Introduction to unstructured communication using Splice

Unstructured communication provides a one-way link for sending ordinary data and text. This process can be used to format Mathematica expressions and pass the results to an external file. This is much more flexible than translating code fragments and avoids the tedious (and sometimes error prone) step of manipulation using a text editor. Furthermore, Mathematica can be used to specify additional program information, ensuring a more efficient and flexible construction than is otherwise possible.

The Mathematica command `Splice` to processes a template file consisting of a mixture of (active) Mathematica and (passive) compiled language source code and produces an object-code output file. The active code is distinguished by statement delimiters (usually `<*` and `*>`) which is taken as Mathematica input. The file is processed by `Splice` which passes over the file, evaluates the active parts and replaces them with Mathematica output, before moving onto the next active block of code. The passive code is translated verbatim. The beauty of this approach is that results are substituted at the exact point in the file which was previously occupied by the active code.

The default format for the output file is determined by the extension of the input file. As an example, the command `Splice[" myfile.mtex "]`, processes the file `myfile.mtex` with `TeXForm` and puts the result in the file `myfile.tex`. An output file name may be specified as a second argument to `Splice`.
E.g. `Splice["myfile.mtex ","outfile.tex "]`.
Here is a simple example of the use of `Splice`.

```
In[1]:= !!example.mf (* Display the template file. *)

        main example
        real x,y
        x = 1.
        y = <* D[x^x Sin[x],x] *>
        write(*,*) y
        stop
        end


In[2]:= Splice["example.mf"]; (* Generate the source code file. *)

In[3]:= !!example.f (* Display the source code file. *)
```

```
        main example
        real x,y
        x = 1.
        y = x**x*Cos(x) + (x**x + x**x*Log(x))*Sin(x)
        write(*,*) y
        stop
        end
```

The `Splice` option `FormatType` can be used to override default output styles. It is emphasised that the `Assign` functions require the setting `FormatType->OutputForm` as an option of `Splice` in order to ensure correct formatting of the output file.

6.2 Defining a subroutine

Here is an example of a subroutine template file.

```
In[4]:= !!sub.mf

        subroutine sub(x,y,z)
        double precision x,y(<* yarray = Array[y,2];
ylength = Length[yarray] *>),z(<* ylength *>)
<*
    (***** Mathematica definition *****)

    expr = yarray^2 - f /@ Table[x^i,{i,ylength}];

    (***** FORTRAN Translation *****)

    FortranAssign[z,expr,AssignToArray->{y}]  *>
        return
        end
```

Arrays are best declared inside the type declaration statement at the beginning of the file, when dimensioning information is required for the first time. The source code file is now generated.

```
In[5]:= Splice["sub.mf",FormatType->OutputForm];

In[6]:= !!sub.f

        subroutine sub(x,y,z)
        double precision x,y(2),z(2)
        z(1)=-f(x)+y(1)**2
        z(2)=-f(x**2)+y(2)**2
```

```
        return
        end
```

The function `f` in this subroutine is as yet undefined and an ANSI compatibility warning message is output to the screen. An example definition is given in the next section.

6.3 Combining code

The adherence to ANSI compiled language functions is illustrated with an example of a function template file for use with the subroutine of the previous section.

```
In[4]:= !!func.mf

        double precision function f(x)
        double precision x
<*  (***** Function Definition *****)

    f = D[(Sin[x] + Cos[x]) Cot[x],x];

    (***** FORTRAN Translation *****)

    FortranAssign[f,f]  *>
        return
        end
```

In addition, Mathematica could be used to specify the type declaration (`real`, `complex` etc.) depending on the requested precision.

```
In[5]:= Splice["func.mf",FormatType->OutputForm];

In[6]:= !!func.f

        double precision function f(x)
        double precision x
        f=-((cos(x)+sin(x))/sin(x)**2)+1.d0*(cos(x)-sin(x))/tan(x)
        return
        end
```

Any definitions made in an external file will remain unless cleared.

```
In[7]:= Clear[f];
```

A simple main routine, `driver.f`, is provided for driving the processed function `func.f` and subroutine `sub.f`. For example, in a UNIX environment, link the driver, subroutine and function files by issuing the command:

```
f77 driver.f func.f sub.f
```

and execute the resulting file (the default is `a.out` on UNIX systems). Various options enable renaming of the executable file (`-o`) and compiler optimization (`-O`) and the user should consult their compiler documentation for more information.

The driver routine can also be generalised using a template file approach. Often it is more convenient to merge the routines in a single template file. It is also good practice to keep all information required for the problem in the template file (and not in additional Mathematica files). This ensures that the program is self-contained and eases the burden of code maintenance.

6.4 A generalised Runge-Kutta subroutine

Consider the problem of writing a general subroutine for a Runge-Kutta scheme [13] for a system of ordinary differential equations. The convention of writing an m-dimensional vector in a bold typeface is adopted.

The general s-stage Runge-Kutta method for the initial-value problem

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y}(\alpha) = \boldsymbol{\eta}, \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^m \,, \tag{1}$$

is defined by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \sum_{i=1}^{s} b_i \, \mathbf{k}_i \,,$$

where

$$\mathbf{k}_i = \mathbf{f}(x_n + c_i \, h, \mathbf{y}_n + h \sum_{j=1}^{s} a_{ij} \, \mathbf{k}_j), \quad i = 1, \ldots, s \,,$$

and the row-sum conditions

$$c_i = \sum_{j=1}^{s} a_{ij} \,,$$

are usually assumed to hold. Furthermore, if $a_{ij} = 0$ for $j \geq i$, $i = 1, \ldots, s$ then the method is said to be an *explicit* or *classical* Runge-Kutta scheme and the $\mathbf{k}_i$ can be calculated by a straightforward recursion.

Structural information for a problem, should invariably be exploited in a symbolic system, since the resulting performance gains can be significant [4]. The same techniques should be applied when creating a general template file. A lot of the tedium can be removed from the analysis of a large number of problems and greater throughput is enabled. The next section contains such an example.

6.4.1 Hamiltonian systems

A Hamiltonian system with d degrees of freedom satisfies Hamilton's equations of motion [2]:

$$\mathbf{q}' = \nabla_{\mathbf{p}} H \quad , \quad \mathbf{p}' = -\nabla_{\mathbf{q}} H$$

$\mathbf{q}$ and $\mathbf{p}$ represent the position and momentum respectively and are d-dimensional vectors. $\nabla_{\mathbf{x}}$ is the gradient operator taken w.r.t. $\mathbf{x}$, $\mathbf{x}'$ denotes the derivative of $\mathbf{x}$ with respect to time, t. $H = H(\mathbf{q}(t), \mathbf{p}(t)) = H(\mathbf{q}, \mathbf{p})$ is the scalar-valued, autonomous (time-independent) Hamiltonian function. A Hamiltonian system is thus (necessarily) of even dimension (with $m = 2d$ in the autonomous form of (1)).

The Hamiltonian $H(\mathbf{q}, \mathbf{p})$ is time-invariant, i.e. it is a constant of the motion. When the Hamiltonian is interpreted as the energy of the system, time-invariance is equivalent to conservation of energy. To see this consider the chain rule:

$$\begin{aligned} H' &= \mathbf{p}'\nabla_{\mathbf{p}}H + \mathbf{q}'\nabla_{\mathbf{q}}H + \tfrac{\partial H}{\partial t} & \\ &= \tfrac{\partial H}{\partial t} & \text{(by Hamilton's equations of motion)} \\ &= 0 & \text{(if H is autonomous)} \end{aligned}$$

6.4.2 The gravitational n-body problem

For the classical n-body problem in dynamics [2], the Hamiltonian is of potential or separable form:

$$H(\mathbf{q}, \mathbf{p}) = T(\mathbf{p}) + V(\mathbf{q})$$

with

$$T(\mathbf{p}) = \frac{\mathbf{p}.\mathbf{p}}{2} \quad , \quad V(\mathbf{q}) = \frac{1}{\sqrt{\mathbf{q}.\mathbf{q}}}$$

where $\mathbf{x}.\mathbf{x}$ denotes the scalar product of $\mathbf{x}$. For example, in 2 degrees of freedom,

$$H(\mathbf{q}, \mathbf{p}) = (p_1{}^2 + p_2{}^2)/2 + 1/\sqrt{(q_1{}^2 + q_2{}^2)}$$

where subscripts denote vector components.

6.4.3 Creating a template file

A general template file for a Runge-Kutta function subroutine for the n body problem is detailed in this section. This involves evaluating the first derivatives of the Hamiltonian. The subroutine below is written so that derivatives with respect to components of position are evaluated first, followed by derivatives of components of momentum.

```
In[4]:= !!rksub.mf

        subroutine evalf(f,q,p)
        implicit double precision (a-h,o-z)
        dimension q(<*

(***** Input degrees of freedom (dimension/2) *****)
```

```
                  d = 3;

      (***** End of required input *****)

(* Ensure that arrays are protected form N during translation. *)

SetOptions[FortranAssign,AssignToArray->{q,p}];

(* Automatically evaluate variables and n-body Hamiltonian *)

qvars = Array[q,d];    pvars = Array[p,d];
H = pvars.pvars/2 + 1/Sqrt[qvars.qvars];
d *>),p(<* d *>),f(<* 2 d *>)
<* (***** Calculate derivatives of the Hamiltonian. *****)

FortranAssign[
   f,
   Flatten[{Outer[D,{H},pvars],- Outer[D,{H},qvars]}]
] *>
      return
      end
```

Appropriate array dimensions are specified by Mathematica. This means that precise control can be exercised over the generated FORTRAN subroutine, by allocating only as much stack space as the problem dictates. This is useful for large scale problems such as molecular dynamics simulations.

An attraction of this approach is that it may be generalised to account for any functional form of Hamiltonian. In order to do so, the Hamiltonian must be specified by the user in the template file, along with the number of degrees of freedom. Mathematica can then be used to `Splice` the results into a file containing FORTRAN source code for the problem.

```
In[5]:= Splice["rksub.mf",FormatType->OutputForm];

In[6]:= !!rksub.f

      subroutine evalf(f,q,p)
      implicit double precision (a-h,o-z)
      dimension q(3),p(3),f(6)
      f(1)=p(1)
      f(2)=p(2)
      f(3)=p(3)
      f(4)=q(1)/sqrt((q(1)**2+q(2)**2+q(3)**2)**3)
      f(5)=q(2)/sqrt((q(1)**2+q(2)**2+q(3)**2)**3)
```

```
f(6)=q(3)/sqrt((q(1)**2+q(2)**2+q(3)**2)**3)
return
end
```

6.4.4 Optimising the code

Computationally, the most expensive part in evaluating the function subroutine comes from the computation of the derivative of the potential term:

$$\frac{\partial V(\mathbf{q})}{\partial q_i} = \frac{q_i}{\sqrt{(q_1^2 + q_2^2 + q_3^2)}} \quad , \quad i = 1, \ldots, 3$$

In particular, evaluation of the fractional power is common to all components. This problem is found in many areas of molecular dynamics simulations ($-\nabla_{\mathbf{q}} V(\mathbf{q})$ is analogous to the force).

Common sub-expressions can be extracted using Mathematica's pattern matching rules before code translation is performed - in an attempt to optimize the resulting FORTRAN code. In general any structural knowledge of the problem in hand should been used to optimize the code. However, some computer algebra systems often use syntactic optimization to extract common sub-expressions and factor powers. The option `AssignOptimize` makes use of the package `Optimize.m` to automatically produce an optimized computational sequence, as outlined in section 5. The user is encouraged to try the optimization option on the example of the previous section. For example, after loading the optimization package, the following statement produces an optimized sequence for the problem.

```
FortranAssign[
    f,
    Flatten[{Outer[D,{H},pvars],- Outer[D,{H},qvars]}],
    AssignOptimize->True,AssignToArray->{p,q},
    OptimizeNull->{p,q},OptimizePower->Binary
]
```

The option `OptimizeNull` effectively tells the optimization procedure that `p` and `q` are arrays (and not functions) and should not be considered as candidates for optimization. The option `OptimizePower` is used to binary factor powers, which has the added benefit of making use of the efficient FORTRAN intrinsic `sqrt`. The resulting output is given below.

```
o1=q(1)**2
o2=q(2)**2
o3=q(3)**2
o4=o1+o2+o3
o5=sqrt(o4)
```

```
o6=o4*o5
o7=1/o6
f(1)=p(1)
f(2)=p(2)
f(3)=p(3)
f(4)=o7*q(1)
f(5)=o7*q(2)
f(6)=o7*q(3)
```

In practice, better numerical schemes exist for Hamiltonian systems than the standard explicit 4th order Runge-Kutta method [27].

6.4.5 Extension to non-autonomous Hamiltonian systems

Assume that the previous problem is extended to non-autonomous Hamiltonians. It is straightforward to modify the above subroutine for this case. In the template file `rksub.mf`, modify the first line to include time-dependence:

```
subroutine evalf(f,q,p,t)
```

The definition of **f** needs to be modified (the vector of first derivatives of H) to account for time-dependence of the Hamiltonian:

```
<* FortranAssign[ f,
     Flatten[{Outer[D,{H},pvars],- Outer[D,{H},qvars],D[H,t]}]
  ] *>
```

The dimension of **f** must also be increased:

```
f(<* 2 d + 1 *>)
```

The main routine should now includes a line to increment the time value at each intermediary stage:

```
t = t + c(i)*h
```

for $i = 1(1)s$. Finally a modified call to `rksub.f` is required

```
call rksub(f,q,p,t)
```

To take the template file approach to an extreme, we could even select an implicit or explicit Runge-Kutta method, depending upon some stiffness criteria implemented in Mathematica.

6.5 A generalised Newton iteration scheme

As another practical example of the use of the template file approach, consider writing a general program for solving a system of equations of the form:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{2}$$

using Newton's method [32]. A bold typeface is again used to denote a vector.

The system of iterates can be written as:

$$\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} - (J^{(i-1)})^{-1}\mathbf{f}^{(i-1)} \quad , \quad i = 1, \ldots, itmax \tag{3}$$

where $J = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}$ denotes the Jacobian matrix and $\mathbf{x}^0$ is a given set of initial values used to start the iteration process. $(J^k)^{-1}$ denotes the inverse of $J^{(k)}$ and the superscript k denotes evaluation at iterate $\mathbf{x} = \mathbf{x}^{(k)}$. The iterative process is continued until some stopping criteria are satisfied at iteration `itmax`. When the system to be solved is scalar valued, $(J^k)^{-1}$ is simply the reciprocal of J^k. For practical implementation, (3) is rewritten at each iteration as:

$$\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} - \mathbf{y}^{(i-1)} \tag{4}$$

where $\mathbf{y}^{(i-1)}$ satisfies

$$J^{(i-1)}\mathbf{y}^{(i-1)} = \mathbf{f}^{(i-1)}. \tag{5}$$

The linear system (5) is typically solved using LU decomposition of the Jacobian matrix at each iteration. The value of $\mathbf{y}^{(i-1)}$ thus obtained is substituted into (4) to obtain $\mathbf{x}^{(i-1)}$.

6.5.1 Evaluating the Jacobian matrix analytically

Mathematica can be used to calculate the Jacobian matrix in *exact* form for any given system of equations. There is currently no built-in definition, but a simple construction using the partial differentiation operator `D` and the generalised outer product `Outer` is possible [15]:

```
JacobianMatrix[f_List,x_List] := Outer[D,f,x]
```

6.5.2 Performance issues

In implementing Newton's method using a template file formulation, greater control over the convergence of the scheme (and convergence properties) is possible than using Mathematica's `FindRoot`. At the same time, much of the generality of the built-in function is retained.

The majority of computational time in Newton's method is spent on LU decomposition and updating of the Jacobian matrix at each iteration. If the matrix is not updated at each iteration quadratic convergence is lost, but similar convergence properties are still retained in the neighbourhood of the solution.

Convergence of the iteration scheme can be controlled by:

1. Moving the definition of the Jacobian matrix and the call to the LU decomposition routine outside the main iteration loop (i.e. to before the do loop labelled 20 in the code below). This means that a fixed Jacobian matrix is used throughout the iteration process (sometimes referred to as modified Newton-Raphson iteration). This approach is not recommended if the starting value is known to be far from the true solution.

2. Adding a clause to ensure that the Jacobian matrix is not updated after a specified number of iterations. If the problem is well conditioned, the Jacobian matrix will not change significantly in value when the scheme is near convergence.

3. Taking account of any special structure arising in the problem - such as a tri-diagonal, banded, sparse or symmetric Jacobian matrix. This decreases the complexity of the linear solver and allows for greater scope in parellisation. For special case linear solution routines see for example [1, 22].

For more information about the linear solution routines used in our example, see [1]. These are a set of machine independent codes in the public domain [23] for linear algebra known as LAPACK.[3] The file `linsolv.f` contains the LAPACK general linear solver and LU decomposition routine which does not utilise property 3 above. Special purpose routines are available for other matrix structures. LAPACK uses Basic Linear Algebra Subprograms (BLAS) with machine specific implementations, thus providing near optimal efficiency for each platform. The necessary BLAS are given as FORTRAN code in the file `linsolv.f` and should be removed if machine specific versions are available.

The file `linsolv.f` should be linked with `newton.f` (the file which results from the application of `Splice` to `newton.mf`) at compile time. When the system of equations becomes large (typically greater than 50x50) iterative refinement of the solution may become necessary. It is straightforward to implement iterative refinement using the LAPACK routine `dgerfs.f`.

6.5.3 Stopping criteria

3 convergence criteria for the scheme are specified in the template file: `relerr`, `abserr` and `fmaxerr`.
The absolute error at each iteration is given by

$$||\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}|| < abserr.$$

The absolute error is necessary when the sequence of iterates is converging to zero, but is sensitive to changes in the scaling of $\mathbf{x}^{(i)}$.
The relative error is given by

$$\frac{||\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}||}{||\mathbf{x}^{(i)}||} < relerr.$$

[3] LAPACK supersedes the previous LINPACK and EISPACK libraries

Combining these we have

$$||\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}|| < ||\mathbf{x}^{(i)}||\, relerr + abserr$$

which provides a reasonable stopping criteria for the Newton scheme. The logical variable `xconv` is used in the program to indicate if this condition has been satisfied.
A convergence criteria for the function $\mathbf{f}$ is also implemented as

$$||\mathbf{f}^{(i)}|| < fmaxerr$$

A relative error test is not necessary since $\mathbf{f}^{(i)}$ is converging to $\mathbf{0}$ from (2). The logical variable `fconv` is used in the program to indicate if this condition is satisfied. The convergence tests are performed in the do loop labelled 30 below.

The iteration is successfully completed when both `fconv` and `xconv` return `true`. For a discussion of convergence considerations, see [24]. The values given below are system-dependent (often depending on the precision of the architecture) and problem specific and may need to be adjusted accordingly. Specifying too strict a tolerance may result in unattainable bounds and for this purpose a maximum number of iterations is specified using the variable `maxit`.

6.5.4 Further enhancements

The code below represents a typical speedup of the order of 10^2 to 10^3 over Mathematica's `FindRoot` (depending on the size of the problem). However, this example will only find real roots of a system of equations and is thus less robust. The limitation here is the linear solution routine (not the translation of complex-valued expressions into FORTRAN). Analogous linear solution routines for complex systems exist (see routines `cgesv.f` and `zgesv.f` from [1]), but are considerably more expensive to implement in terms of the arithmetic overhead involved.

```
In[4]:= !!newton.mf

        program newton
        implicit double precision(a-h,o-z)
        double precision jac
        logical fconv,xconv
        integer dim
        dimension x(<*
SetOptions[FortranAssign,AssignToArray->{x}];

      (***** General problem formulation. *****)

       (* Input equations in variables x[i]. *)

       f = { Sin[x[1] x[2]^2] - 1, Exp[ Cos[x[1]/x[2]] ] - 1 };
```

```
(* Input initial values - must the same dimension as f. *)

                   init = {1.,.5};

         (***** End of required input. *****)

dim = Length[f];    vars = Array[x,dim];    dim *>)
        dimension jac(<* dim *>,<* dim *>),ipiv(<* dim *>)
        dimension f(<* dim *>),xnew(<* dim *>)
        data dim/<* dim *>/
c
c       Specify convergence criteria and maximum number of iterations.
c
        data maxit/30/,relerr/1.d-10/,abserr/1.d-10/,fmaxerr/5.d-9/
<* (***** Assign initial values to x. *****)

FortranAssign[x,init] *>
        do 10 i = 1,dim
            xnew(i) = x(i)
10      continue
        do 20 k = 1,maxit
<* (***** Calculate Jacobian Matrix. *****)

FortranAssign[jac,Outer[D,f,vars]]
*>
<* FortranAssign[f,f] *>
c
c       LU decomposition of Jacobian matrix.
c
            call dgetrf(dim,dim,jac,dim,ipiv,info)
c
c       Linear solution routine overwriting f with solution.
c
            call dgetrs('No transpose',dim,1,jac,dim,ipiv,f,dim,info)
c
c       Update solution and error indicators.
c
            fconv = .true.
            xconv = .true.
            xmaxdiff = 0.d0
            xmax = 0.d0
            do 30 i = 1,dim
```

```
                xnew(i) = x(i) - f(i)
                if (abs(f(i)).gt.fmaxerr) then
                    fconv = .false.
                endif
                xnewabs = abs(xnew(i))
                if (xnewabs.gt.xmax) then
                    xmax = xnewabs
                endif
                xdiff = abs(xnewabs - abs(x(i)))
                if (xdiff.gt.xmaxdiff) then
                    xmaxdiff = xdiff
                endif
30          continue
            do 40 i = 1,dim
                x(i) = xnew(i)
40          continue
            if (xmaxdiff.gt.(relerr*xmax+abserr)) then
                xconv = .false.
            endif
c
c       Test if convergence criteria are satisfied.
c
            if (fconv.and.xconv) then
                goto 50
            else
                if (k.eq.maxit) then
                    write(*,100) k
                endif
            endif
20      continue
c
c       Output solution vector.
c
50      do 60 i = 1,dim
            write(*,*) x(i)
60      continue
100     format('Newton scheme did not converge after ',
     &  i2,' iterations')
        stop
        end
```

The template file is then processed as follows.

```
In[5]:= Splice["newton.mf",FormatType->OutputForm];
```

```
In[6]:= !!newton.f

        program newton
        implicit double precision(a-h,o-z)
        double precision jac
        logical fconv,xconv
        integer dim
        dimension x(2)
        dimension jac(2,2),ipiv(2)
        dimension f(2),xnew(2)
        data dim/2/
c
c       Specify convergence criteria and maximum number of iterations.
c
        data maxit/30/,relerr/1.d-10/,abserr/1.d-10/,fmaxerr/5.d-9/
        x(1)=1.
        x(2)=0.5
        do 10 i = 1,dim
            xnew(i) = x(i)
10      continue
        do 20 k = 1,maxit
        jac(1,1)=cos(x(1)*x(2)**2)*x(2)**2
        jac(1,2)=2.*cos(x(1)*x(2)**2)*x(1)*x(2)
        jac(2,1)=-(exp(cos(x(1)/x(2)))*sin(x(1)/x(2))/x(2))
        jac(2,2)=exp(cos(x(1)/x(2)))*sin(x(1)/x(2))*x(1)/x(2)**2
        f(1)=-1.+sin(x(1)*x(2)**2)
        f(2)=-1.+exp(cos(x(1)/x(2)))
c
c       LU decomposition of Jacobian matrix.
c
            call dgetrf(dim,dim,jac,dim,ipiv,info)
c
c       Linear solution routine overwriting f with solution.
c
            call dgetrs('No transpose',dim,1,jac,dim,ipiv,f,dim,info)
c
c       Update solution and error indicators.
c
            fconv = .true.
            xconv = .true.
            xmaxdiff = 0.d0
            xmax = 0.d0
```

```
          do 30 i = 1,dim
              xnew(i) = x(i) - f(i)
              if (abs(f(i)).gt.fmaxerr) then
                  fconv = .false.
              endif
              xnewabs = abs(xnew(i))
              if (xnewabs.gt.xmax) then
                  xmax = xnewabs
              endif
              xdiff = abs(xnewabs - abs(x(i)))
              if (xdiff.gt.xmaxdiff) then
                  xmaxdiff = xdiff
              endif
30        continue
          do 40 i = 1,dim
              x(i) = xnew(i)
40        continue
          if (xmaxdiff.gt.(relerr*xmax+abserr)) then
              xconv = .false.
          endif
c
c     Test if convergence criteria are satisfied.
c
          if (fconv.and.xconv) then
              goto 50
          else
              if (k.eq.maxit) then
                  write(*,100) k
              endif
          endif
20    continue
c
c     Output solution vector.
c
50    do 60 i = 1,dim
          write(*,*) x(i)
60    continue
100   format('Newton scheme did not converge after ',
     & i2,' iterations')
      stop
      end
```

6.5.5 Optimization of the function and Jacobian matrix

Structural knowledge and functional properties should be used by the symbolic environment in order to improve the efficiency of the resulting compiled code [21]. Optimisation can be used for more efficient evaluation of the function and its Jacobian matrix in Newton's method, in much the same way as in the Runge-Kutta example of section 6.4.4. For example, the optimization process can resolve matrix symmetries.

Illustration of these issues is given by a simple example. Consider a Jacobian matrix derived from a function of two variables $f(x, y)$.

```
In[1]:= f[x_,y_] := {x + y + 2 x Sin[y]^2, x + y + x^2 Sin[2 y]};

In[2]:= jac[f_List,vars_List] := Outer[D,f,vars];

In[3]:= matrix = jac[f[x,y],{x,y}];

In[4]:= matrix //MatrixForm

Out[4]//MatrixForm=
            2
1 + 2 Sin[y]            1 + 4 x Cos[y] Sin[y]

                                   2
1 + 2 x Sin[2 y]        1 + 2 x  Cos[2 y]
```

In fact this matrix turns out to be symmetric:

```
In[5]:= simpmat = Simplify[matrix];
In[6]:= simpmat //MatrixForm

Out[6]//MatrixForm=
2 - Cos[2 y]        1 + 2 x Sin[2 y]

                              2
1 + 2 x Sin[2 y]    1 + 2 x  Cos[2 y]
```

A considerable amount of computation can be saved if the property $m_{12} = m_{21}$ is utilised. Moreover, the functions `Cos[2 y]` and `Sin[2 y]` need only be evaluated once if the values are stored as temporary variables. This information is again resolved by syntactic optimization. An optimized computational sequence in FORTRAN may be obtained as:

```
In[7]:= FortranAssign[m, simpmat, AssignOptimize->True,
          OptimizePower->True,OptimizeTimes->False]
Out[7]//OutputForm=
        o1=cos(2.d0*y)
```

```
o2=sin(2.d0*y)
o3=1.d0+2.d0*o2*x
m(1,1)=2.d0-o1
m(1,2)=o3
m(2,1)=o3
m(2,2)=1.d0+2.d0*o1*x**2
```

Notice that the matrix components are optimized, but the encapsulating lists themselves are not, since the default setting `OptimizeNull->{List}` has been used to ignore such objects. Also we use the option `OptimizeTimes->False` to ignore sub-expressions with head `Times`. Such operations do not yield additional optimizations in this example and ignoring them speeds up the optimization process.

6.6 Code optimization using Splice

Some information is given here to explain how `Splice` can be used in conjunction with `Optimize` to automate code generation. Specifically, it is necessary to keep track of the number of optimization variables introduced during the optimization process, in order to make appropriate type and dimension declarations.

In FORTRAN, it is possible to declare all optimization variables to be the same type by using `implicit` statement declarations as mentioned previously. In C this is not possible and there are several strategies which may be adopted, the simplest of which is outlined below.

- Use the option `OptimizeFormat->False`. This has the effect of generating the optimization variables in array form `o[i]` (the default is a concatenated format `oi` which is more memory efficient). This is facilitated by a finishing touch during formatting, where optimization variables are returned in consecutively numbered form `o[1], o[2],...`.

- It is difficult to judge in advance exactly how many optimizations will be performed. `Optimize` returns a list of the form `{optseq,optexpr}`. The dimension of the optimization array corresponds to `Length[optseq]`.

- Use a two-pass `Splice`. The first pass fills in the body of the procedure and contains statements such as `o(ToExpression["<* arraydim *>"])`. The second pass is used to fill in the array dimension `arraydim`.

7 Conclusions and further enhancements

Many considerable shortcomings of Mathematica's format rules have been addressed. Particular attention has been given to the accuracy and syntax of translated code. At the same time `Format.m` has been written to perform efficiently, allow the user a high degree of control over the formatted output and interact seamlessly with the rest of the Mathematica system. The package has been written primarily for use with `Splice` but may also be used interactively within a session. The problem-specific editing of files has been minimised by the provision of numerous formatting options. Tedious syntax errors may thus be eliminated.

Many physical problems have a degree of mathematical structure which may be utilised in the formulation of a general template file. In this way precise mathematical information relating to a problem can be derived using Mathematica and passed to a more efficient computational environment.

It is possible to further enhance a symbolic-numeric interface by compiling, linking and executing programs and analysing results in a single Mathematica session. [11] provides an example of an interface for accessing numerical libraries in the Unix environment. Robb [5] has developed the commercial package `InterCall`, which links Mathematica with the NAG and IMSL libraries as well as user supplied routines. For example, `InterCall` facilitates the definition of a Mathematica function which executes external code. MacGregor and co-workers [14] also developed a code translation package for Mathematica as an internal project for Schlumberger plc., but this was never formally released.

A process analogous to the `Splice` form of unstructured communication will be available using `TemplateFile` in AXIOM [3] version 2.0 - along with additional tools for generating complete subprograms. However, the syntax of these additional tools is (necessarily) quite complicated in comparison with the template approach. The GENTRAN package [8] for REDUCE also enables the use of template files among other more complex techniques. Perhaps surprisingly there is no such facility yet in Maple.

Other utilities for C and FORTRAN code generation exist and may be useful. Public domain versions include:

1. `Toolpack` (also available from NAG [22]) is a utility for formatting and rewriting FORTRAN code. The utility addresses some performance issues (such as vectorisation of loops) whilst maintaining logical structure.

2. `f2c` - this is a FORTRAN to C translation tool. Complete source code is available from AT&T.

3. `GNU FORTRAN` - this is a front end (in development) which uses `f2c` and the `GNU CC` compiler. For up to date information, see [7].

The program `nb2tex` is also useful for converting Mathematica notebooks into TEX documents [19].

The package `Format.m` and related files are available from the MathSource archive [19] under the item number 0205-254. The current version supersedes that distributed as part of the electronic supplement to the Mathematica Journal [28]. The package `Optimize.m` and related files can be found under the item number 0206-592.

Acknowledgements

Thanks to Dave Withoff for his many comments and suggestions for improving the package and for details on Mathematica's evaluation process. Thanks also to Grant Keady for his interest and enthusiasm concerning a preliminary version of the package. The author is grateful to all those who have contributed suggestions (a list of contributors is given in the package).

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen (1992) The LAPACK users' guide, SIAM publications, Philadelphia.

[2] V. I. Arnold (1989) Mathematical methods of Classical Mechanics, Springer Verlag.

[3] AXIOM is a development of SCRATCHPAD which was originated by IBM in Yorktown Heights. It is now being developed and marketed by the Numerical Algorithms Group Ltd.

[4] R. L. Brenner (1984) Simplifying Large Algebraic expressions by computer, in Golden, V. E. editor, Proceedings of the 1984 Macsyma User's Conference, 50-109, General Electric, Schenectady, New York.

[5] K. A. Broughan, G. Keady, T. D. Robb, M. G. Richardson and M. C. Dewar (1991) Some symbolic computing links to the NAG numeric library, *SIGSAM bulletin*, 25(3), 28–37.

[6] M. C. Dewar (1992) Interfacing Algebraic and Numeric Computation, Ph.D. Thesis, University of Bath, U.K. available as Bath Mathematics and Computer Science Technical Report 92-54.

[7] The file of Frequently Asked Questions from the net news group comp.lang.fortran contains information on the status of the `GNU FORTRAN` project amongst other issues. It can be found at: `ftp://rtfm.mit.edu//pub/usenet-by-group/comp.lang.fortran/Fortran_FAQ`.

[8] B. L. Gates (1987) The GENTRAN User's Manual: Reduce Version. The RAND Corporation.

[9] D. Goldberg (1991), What Every Computer Scientist Should Know About Floating Point Arithmetic, ACM. Computing Surveys **23** 1, 5–48.

[10] S. P. Harbison and G. L. Steele (1991) C: A Reference Manual, Prentice-Hall.

[11] V. Khera and H. Greenside (1990) An Interface for Accessing External Numerical Libraries, The Mathematica Journal, Vol.1 Issue 2, 84–88.

[12] D. E. Knuth (1989) The TeXbook, Addison Wesley.

[13] J. D. Lambert (1991) Numerical Methods for Ordinary Differential Equations : The Initial Value Problem, John Wiley.

[14] B. MacGregor (1990) MathCode : A Code Generation Package for Mathematica, Schlumberger Technologies Corp.

[15] R. Maeder (1991) Programming in Mathematica, Addison Wesley.

[16] R. Maeder (1992) Advanced Programming in Mathematica, Course Notes: the Boston Mathematica Conference.

[17] MAPLE is a product of Waterloo Maple Software, 160 Columbia Street West, Waterloo N2L 3L3, Canada.

[18] MathLink Reference Guide (1991) Wolfram Research technical report, WRI.

[19] nb2tex is available from MathSource (mathsource@wri.com), the electronic archive of Mathematica-related programs, maintained by Wolfram Research Inc.

[20] L. P. Meissner and E. I. Organick (1980) FORTRAN 77 Featuring Structured Programming, Addison Wesley.

[21] M. P. W. Mutrie, B. W. Char and R. H. Bartels (1987) Expression Optimization Using High-Level Knowledge, Proceedings of EUROCAL '87, Springer-Verlag, 64–70.

[22] NAG FORTRAN Library Manual Mark 15 (1992) Numerical Algorithms Group Ltd., Wilkinson House, Jordan Hill Road, Oxford , U.K.

[23] NETLIB is a collection of sites containing public domain numerical software, such as LAPACK and ODEPACK. Send an email massage containing the line 'send index' for an overview. The current sites are : netlib@research.att.com (AT&T Bell Labs, New Jersey, USA), netlib@ornl.gov (Oak Ridge Nat. Lab, Tenn., USA), netlib@unix.hensa.ac.uk (Univ. of Kent, UK), netlib@nac.no (Oslo, Norway), netlib@cs.uow.edu.au (U. of Wollongong, NSW, Australia)

[24] T. S. Parker and L. O. Chua (1989) Practical Numerical Algorithms for Chaotic Systems, Appendix A, Springer-Verlag.

[25] R. D. Skeel and J. B. Keiper (1993) Elementary Numerical Computing with Mathematica, McGraw-Hill.

[26] C. Smith (1993) Notebooks into Books via LaTeX, *The Mathematica Journal*, Volume 3 Issue 3, 69–73.

[27] M. Sofroniou, Symbolic And Numerical Methods for Hamiltonian Systems, Ph.D. thesis, Loughborough University of Technology, United Kingdom (to appear).

[28] M. Sofroniou (1993), Extending The Built-In Format Rules, *The Mathematica Journal*, Volume 3 Issue 3, 74–80.

[29] M. Sofroniou, A Package For Expression Optimization, *Journal Of Symbolic Computation* (to appear).

[30] M. Sofroniou (1993), An Efficient Symbolic-Numeric Environment Using Mathematica, in *Proceedings of the Workshop on Symbolic and Numerical Computation*, ed. H. Apiola, University of Helsinki Technical Report Series, 69–83.

[31] W. Stallings (1990) Computer Organisation and Architecture : Principles of Structure and Function, 2nd edition, MacMillan.

[32] J. Stoer and R. Burlisch (1980) Introduction To Numerical Analysis, Springer Verlag.

[33] D. Stoutemyer (1993), Crimes and Misdemeanors in the Computer Algebra Trade, Notices of the AMS. **38** 7, 778–785.

[34] M. C. Wirth (1980), On the Automation of Computational Physics, Ph.D. thesis. Univ. Calif., Davis School of Applied Science, Lawrence Livermore Lab. (also in 1981 SYMSAC).

[35] S. Wolfram (1991), Mathematica: A Systems for Doing Mathematics by Computer, Addison Wesley, second edition.