The `fussy` language: Implementation of an automatic error propagation algorithm

S. Bhatnagar
National Radio Astronomy Observatory

Nov. 2003

Abstract

Formal propagation of random errors in a mathematical expression follow a precise prescription based on calculus. This requires the computation of the variation of the function with respect to each of the independent variables used to construct the function. These variations are added in quadrature to compute the final numerical error. For complicated expressions, computation of all the partial derivatives is often cumbersome and hence error prone.

The `fussy`[1] scripting language, described here, implements an algorithm for automatic propagation of random measurement errors in an arbitrary mathematical expression. It is internally implemented as a virtual machine for efficient runtime performance and can be used as an interpreter by the user. A simple `C` binding to the interpreter is also provided. Mathematical expressions can be implemented as a collection of sub-expressions, as sub-program units (functions or procedures) or as single atomic expressions. Errors are correctly propagated when a complex expression is broken up into smaller sub-expressions. Sub-expressions are assigned to temporary variables which can then be used to write the final expression. These temporary variables are not independent variables and the information about their dependence on other constituent independent variables is preserved and used on-the-fly in error propagation.

The scripting syntax of `fussy` is similar to that of `C`. It is therefore easy to use with minimal learning and can be used in every day scientific work. Most other related work found in the literature is in the form of libraries for automatic differentiation. Only two tools appear to have used it for automatic error propagation. Use of these libraries and tools require sophisticated programing and are targeted more for programmers than for regular every day scientific use. Also, such libraries and tools are difficult to use for correct error propagation in expressions composed of sub-expressions.

[1]The name reflects the original intention of designing a language for fuzzy arithemetic. It is also a pun on those who (wrongly) consider error propagation as too much fuss!

1 Introduction

If $\vec{x}$ is a vector of independent experimentally measured quantities with associated random measurement error $\delta\vec{x}$, the formal error on a function $f(\vec{x})$ is given by

$$\delta f = \sqrt{\sum_i \left(\frac{\partial f}{\partial x_i}\delta x_i\right)^2} \tag{1}$$

Further, if $f(\vec{x})$ is a functional, e.g. $f(\vec{x}) = g(h(k(\vec{x})))$, then the partial derivative of f is given by the derivative chain rule:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial x_i} \tag{2}$$

Therefore, for the computation of δf, one requires:

1. the partial derivative of the function with respect to each independent variable ($\partial f/\partial x_i$)
2. δx_i - the measurement error
3. chain rules of differential calculus for the mathematical operators (which will use the x_i's and $\partial f/\partial x_i$'s).

In the following sections, the implementation of an algorithm for automatic computation of partial derivatives and propagation of random errors in an arbitrary mathematical expression is described. The algorithm is implemented as a scripting language called `fussy` and can be used as an interpreter by the user. The syntax is similar to that of the `C` language making it easy to use in an interactive session or as a scripting environment. Each user defined variable in `fussy` is treated as an independent variable and expressions can be constructed using an arbitrary number of variables. It is error prone to express complicated expressions as single atomic statements and usually the final expression is built out of sub-expressions and temporary variables. For the purpose of random error propagation however, temporary variables are dependent on the independent variables (normal user defined variables) on the right hand side of an assignment operator. The algorithm described below does correct error propagation in the final expression composed of such temporary variables. A special language feature is used to distinguish between such temporary and normal variables as well as to associate measurement errors with numbers (see Appendix B.3).

Although it is possible to code Eq. 1 in other tools (Gillespie 1992; Harrison 1995; Bischof et al. 1997), it requires sophisticated programing and learning often arcane, new programing tools. This is usually time consuming and enough of a bother to discourage its use for the purpose of error propagation in every-day scientific use. The work of Stoutemyer (1977) using the `REDUCE` algebraic manipulation language (Hearn 1971, 1995) was one of the first which used automatic symbolic differentiation for error analysis in *single atomic mathematical expressions*. Harrison (1995) has used a similar approach and developed a tool for `Mathematica`[2]. There

[2]©2003 Wolfram Research, Inc.

are program development libraries (Barton & Nackman 1994; Griewank et al. 1996; Tsukanov & Hall 2003) for automatic differentiation which could be used for similar purpose. However they too suffer from the same problem of requiring more effort from the user than is possible in everyday work. Besides, most of these existing tools will be hard to use for multi-variate expressions and functionals. They are even harder (if not impossible) to use directly for complicated expressions expressed as a combination of sub-expressions. Apart from the difficulty of use, the two tools which do use automatic differentiation for error propagation require access and familiarity with other packages (the `REDUCE` package or the commercially sold package `Mathematica`).

The `fussy` interpreter is implemented internally as a virtual machine (VM) with a stack of its own. The derivative chain rule (Eq. 2) is implemented using a separate VM which has a separate stack *per independent variable* to hold the intermediate partial derivatives. At the terminal nodes of a parsing tree (e.g. the '=' operator) the values from these stacks are used to evaluate Eq. 1. A user program written in `fussy` is compiled into the VM instruction-set, referred to as the op-codes, to manipulate the VM stack (`VMS`), call built-in functions, perform basic mathematical operations or call user defined sub-program (functions or procedures). These op-codes are function calls which perform the operation they represent (mathematical operators, built-in function call or branching to a sub-program unit) as well as the steps required for automatic error propagation. Since user defined programs/expressions are translated into these op-codes, errors are correctly propagated in the mathematical expression in any arbitrary user program.

A simple `C` binding to the interpreter is also provided. The user program can be supplied to the interpreter via an in-memory string using the function `calc(char *InString, edouble &ans, FILE *InStream, FILE *OutStream)`. The contents of the `InString` are parsed and converted to a VM instruction set. The result of the execution of this program is returned in `ans`. The last two arguments are not used in this case. Alternatively, if `InString` is set to `NULL` and the last two arguments set to valid file pointers, the interpreter will take the input from `InFile` and use `OutFile` as the output stream. A similar `C++` interface of type `calc(char *InString, ostream &ResultStream, FILE *InStream, FILE *OutStream)` writes the result of the program supplied in `InString` or via the file pointer `InStream` to the output stream `ResultStream`. `OutStream` in both interfaces is used as the output file for the error messages.

For a better understanding, in Section 2 I describe the algorithm for automatic random error propagation for the simpler case of a single variate expression. Section 3 describes the complete algorithm, along with the logic for the various operators in the form of pseudo code. The correctness of the algorithm is demonstrated in Section 4 using numerical examples. On the lines of proof of correctness of the algorithm, it is also argued that the algorithm is general and will work for any arbitrary expression. In Appendix A, a step-wise description of the algorithm for a general mathematical expression is given. Appendix B describes the syntax of the `fussy` language.

Figure 1: The parsing tree for the expression $f(x) = \sin(x) * \cos(x) + \sin(\cos(x))$

2 Error propagation: Single variable case

For the case where f is a function of a single measurable x, the right hand side of Eq. 1 can be evaluated as follows. Each leaf of the parsing tree will either be (1) a constant, (2) a variable, or (3) another sub-tree representing a sub-expression. The derivatives can be computed by the repeated application of the derivative chain rule. Starting from the bottom of the tree, a value of 1 is pushed on the Derivative Stack (`DS`) (equivalent of putting $\partial x/\partial x$ on the stack) for every leaf of the tree (which, at the bottom, correspond to the symbols from the symbol table or constants). The nodes of a tree corresponds to one of the arithmetic operators ('+', '-', '/', '*', '^', and '**') or built-in functions, which are implemented as function calls. These functions push the result of the operations on the `VMS` while the corresponding partial derivatives are pushed on the `DS`.

The final result and the error propagation will in general use the values from both the stacks (the `VMS` and the `DS`). E.g. for $f(x) = \sin(x) * \cos(x)$, when the execution reaches the node for the '$*$' operator, the `VMS` will have two values, namely $\sin(x)$ and $\cos(x)$. The `DS` also has two values, namely the two derivatives $\partial \sin(x)/\partial x = \cos(x)$ and $\partial \cos(x)/\partial x = -\sin(x)$. The value of f is pushed on the `VMS`, and its derivate $(\partial \sin(x)/\partial x * \cos(x) + \sin(x) * \partial \cos(x)/\partial x)$, computed using both the stacks, pushed on the `DS`. The '=' operator rule finally takes the value from the `DS`, and compute the right hand side of Eq. 1.

An arbitrary expression composed of user defined variables or built-in functions, will itself be represented as a sub-tree. Hence, applying the above algorithm recursively, case (3) above (a sub-expression) will also be correctly handled.

2.1 Example

Let $f(x) = \sin(x) * \cos(x) + \sin(\cos(x))$ (this includes three sub-expressions one of which is a functional), represented as a tree in Fig. 1. A value of 1 is pushed on the `DS` whenever a symbol from the symbol-table is pushed on the `VMS`. When branch **1** in the above tree is reduced, a call to the built-in function sin pops a value from the `VMS` (which is x) and a value from the `DS` (say dx, which is 1). It then pushes the value of $\sin(x)$ on the `VMS` and a value of $dx * \partial \sin(x)/\partial x = 1 * \cos(x)$ on the `DS`. Similar operations are done for evaluating $\cos(x)$.

When the execution reaches node **2**, the `VMS` has the values $\sin(x)$ (`L`) and $\cos(x)$ (`R`) and the `DS` has $\cos(x)$ (`dL`) and $-\sin(x)$ (`dR`). Since '$*$' is a binary operator, when node **2** is reduced, two values each from the `VMS` and the `DS` are poped. The multiplication operator then pushes `L*R`=$\sin(x)*\cos(x)$ on the `VMS` while `L*dR + R*dL`=$\cos^2(x)-\sin^2(x)$ is pushed on the `DS` (note that this uses values from the `DS` as well as from the VMS). Both the stacks now have one value each - `VMS` the value of the sub-expression $\sin(x)*\cos(x)$ and the `DS` the value of the derivative of this sub-expression.

Next, branch **3** is evaluated. Again, 1 and x are pushed on the `DS` and the `VMS` respectively. A call to cos compute the derivative of $\cos(x)$ (namely, $-\sin(x)$) and multiplies it by the top of the `DS` (which is 1). When call for sin is made, its argument ($\cos(x)$) and the derivative of the argument are on the `DS` and `VMS` respectively. A value from the `VMS` and `DS` each (say `L`=$\cos(x)$ and `dL` =$-\sin(x)$) respectively) are poped. $\sin(L) = \sin(\cos(x))$ and `dL`$*\sin(L) = -\sin(x)*\sin(\cos(x))$ are pushed on the `VMS` and `DS` respectively. This is the equivalent of Eq. 2 for branch **3**. At this stage, the two values on the `VMS` are the values of the two sub-expression and `DS` has the values of the partial derivatives of the two sub-expressions.

Reduction of the node **4** will then again invoke the rule for the derivative and the binary operator for addition: pop two values each from the `VMS` and `DS`, push the result of the operator on the `VMS`, and the derivative (`dL+dR` in this case) on the `DS`. Top of the `DS` now has $\partial f/\partial x$ and the '=' operator computes Eq. 1.

3 Error propagation: Algorithm for the multi-variate case

The algorithm for the multi-variate case is similar to the single-variate case described above, but slightly more complicated. Multi-variate expressions have the added complication that the terms in the summation of Eq. 1 have to be evaluated separately for *each independent* variable. This means that a `DS` and a table of measurement errors (`ME`) *per independent* variable have to be maintained.

Symbols (variables and constants) in `fussy` are tagged with a number or a list of numbers (the IDs) and a type. Symbols representing normal variables are of the type `VAR` and have a single unique ID, a random error and a partial derivative (of value 1) associated with them. Symbols representing sub-expressions are of the type `PARTIAL_VAR` and have a list of IDs and a corresponding list of random errors and partial derivatives associated with them. List of unique IDs and the random errors of the independent variables in the expression on the right-hand side (RHS) of the assignment operator constitute the list of IDs and random errors for the `PARTIAL_VAR` type symbols. When a symbol (of either type) is pushed on the `VMS`, the entire list of associated IDs is copied to the ID list of the object on the stack. The corresponding random errors and partial derivatives are also copied in the appropriate locations in the `ME` table and pushed on the appropriate `DS` respectively. For example, let the IDs of x_1 and x_2 be 1 and 2 respectively. The result of $x_1 * x_2$ on the top of the `VMS` will have an ID list of `{1,2}` retaining the information that the result is statistically dependent on the independent variates x_1 and x_2. If this result is further used as part of another expression, this information will be used to propagate

the chain rule for these variables correctly.

Since any expression is built using basic mathematical operators or built-in functions, for the purpose of proving the correctness of the error propagation algorithm for any arbitrary expression, it is sufficient to prove that the algorithm works for the fundamental mathematical operators and built-in functions. The algorithms for evaluating the partial derivatives involving mathematical operators and the final evaluation of the resulting error is described below as pseudo code (see Appendix A for an example). The algorithms are described using the following pseudo functions:

- `push(S)`: to push the symbol or value `S` on the `VMS`.
- `push(S,DS[i])`: to push the symbol or value `S` on the i^{th} `DS`.
- `pop()`: to pop a symbol from the `VMS`.
- `Top(DS[i])`: represents the value on the top of the i^{th} `DS`.

The `DS` is indexed by the symbol ID(s) (say, `N`). When a symbol from one of the symbol tables is pushed on the `VMS`, a value of 1 is pushed on `DS[N]` and the associated measurement error is copied into `ME[N]`. `L` and `R` in the pseudo code represents the symbols on the left-hand-side (LHS) and right-hand side (RHS) of the operator respectively. Two functions $f(x, a)$ and $g(x, b)$ with one common variable (x) and one non-common variable each (a and b) are used as the LHS and RHS operands in the explanation below.

All the operators described below are binary operators. They all pop two values from the top of the `VMS`, compute the result by applying the corresponding operator, store the value in a temporary stack object, set its ID to the union of the IDs of the operands and push it on the `VMS`. These operations are performed by the following pseudo function:

```
ComputeResult(L,R,Expr)
  {
      L = pop();     R = pop();
      S = Expr(L,R);
      S.IDList = union(L.IDList,R.IDList);
      push(S);
  }
```

`Expr` implements the arithmetic of the mathematical operation on `L` and `R`.

Partial derivatives with respect to all the IDs in the ID lists of the operands are at the top of the corresponding `DS`s. For all the IDs common between the two operands, two values are poped from the corresponding `DS`, say `dxR` and `dxL`. The common IDs represent the variables which are part of both the operands (here x). Operations to compute the partial derivatives with respect to these common variables (equivalent of the $\frac{\partial}{\partial x}$ operator) are represented by the pseudo function `dCommonVar` below. The function `CommonExpr` implements the arithmetic for

the derivative computation and is set to the appropriate function for the various operators. These values are computed for each ID in the set composed of the intersection of the ID lists of the two operands, and pushed on the corresponding DS (here, the ID of x). Finally, all IDs common between the two operands are removed from the ID lists of each operand.

```
dCommonVar(L,R,CommonExpr)
  {
      IDList = intersection(L.ID,R.ID);
      for ID in IDList
        {
           dxR = pop(DS[ID]);
           dxL = pop(DS[ID]);
           push(CommonExpr(L,R,dxL,dxR),DS[ID]);
           L.Remove(ID);  R.Remove(ID);
        }
  }
```

The list of IDs of the two operands now has IDs corresponding to the non-common variables only (a and b here). Operations for the partial derivatives of the operands with respect to these variables is represented by the pseudo function dNonCommonVar below. LExpr and RExpr computes the value of these derivatives for the LHS and RHS of the operator (equivalent of computing $\frac{\partial f(x,a)}{\partial a}$ and $\frac{\partial g(x,b)}{\partial b}$) using the values from the top of the appropriate DS.

```
dNonCommonVar(L,R,LExpr, RExpr)
  {
      for ID in L.IDList
         Top(DS[ID]) = LExpr(L,R,Top(DS[ID]));

      for ID in R.IDList
         Top(DS[ID]) = RExpr(L,R,Top(DS[ID]));
  }
```

3.1 The multiplication operator

The following code returns the result of the operator (L*R) on the top of the VMS with an ID equal to L.ID $\cup$ R.ID.

```
Expr(L,R) {return L*R};  /* Compute the value for
ComputeResult(L,R,Expr);    the multipication operator */
```

Following code returns the value(s) of the partial derivative(s) (here dxR*L + dxL*R) with respect to *all* the common variables in the two operands, on the top of the appropriate DSs.

```
Expr(L,R,dxL,dxR) {return dxR*L + dxL*R;}
dCommonVar(L,R,Expr);
```

The following code returns the value of the partial derivatives with respect to the variables not common between the operands. These values are returned on the top of the DS corresponding to the remaining IDs of the two operands.

```
LExpr(L,R,dx) {return dx*L};
RExpr(L,R,dx) {return dx*R};
dNoncommonVar(L,R,LExpr,RExpr)
```

3.2 The division operator

As before, the result of the division operator is computed as L/R and returned on the VMS using the following code.

```
Expr(L,R) {return L/R};  /* Compute the value for
ComputeResult(L,R,Expr);    the division operator */
```

For all IDs common between L and R, the partial derivative is computes as (R*dxL - L*dxR)/(R*R) and returned on the appropriate DSs using the following code.

```
Expr(L,R,dxL,dxR) {return (R*dxL - L*dxR)/(R*R)};
dCommonVar(L,R,Expr);
```

This is equivalent to computing

$$\frac{\partial}{\partial x}\left[\frac{f(x,a)}{g(x,b)}\right] = \frac{1}{g(x,a)^2}\left[g(x,a)\frac{\partial f(x,a)}{\partial x} - f(x,a)\frac{\partial g(x,b)}{\partial x}\right] \tag{3}$$

Next, partial derivatives with respect to each of the non-common variables are computed (here with respect to a as $\frac{1}{g(x,b)}\frac{\partial f(x,a)}{\partial a}$ and with respect to b as $-\frac{f(x,a)}{g(x,b)^2}\frac{\partial g(x,b)}{\partial b}$) and returned on the appropriate DSs by the following code.

```
LExpr(L,R,dx) {return dx/R};
RExpr(L,R,dx) {return -L*dx/(R*R)};
dNoncommonVar(L,R,LExpr,RExpr);
```

3.3 The addition operator

The first set of operations is same as that for other operators except that the value pushed on the VMS is L+R.

```
Expr(L,R) {return L+R};  /* Compute the value for
ComputeResult(L,R,Expr);    the addition operator */
```

Since the partial derivatives of the expression with respect to the non-common variables is already on the appropriate DS, no separate operation is required for these variables. The partial derivatives with respect to the common set of variables is computed as $\frac{\partial f(x,a)}{\partial x}$ and $\frac{\partial g(x,b)}{\partial x}$. The pseudo code for these operations is:

```
Expr(L,R,dxL,dxR) {return (dxL + dxR)};
dCommonVar(L,R,Expr);
```

3.4 The subtraction operator

The pseudo code for the subtraction operation is functionally same as that for the addition operator.

```
Expr(L,R) {return L-R};  /* Compute the value for the
ComputeResult(L,R,Expr);    subtraction operator     */

/* Compute the derivative w.r.t. common variables */
Expr(L,R,dxL,dxR) {return (dxL - dxR)};
dCommonVar(Expr,L,R);
```

This is equivalent of computing $\frac{\partial f(x,a)}{\partial x} - \frac{\partial g(x,b)}{\partial x}$. In addition to the above operations, the partial derivatives of the RHS operand with respect to all the non-common variables needs to be negated (here $\partial g(x,b)/\partial b$).

```
IDList = union(L.IDList, R.IDList);
for ID in IDList /* unique IDs in L and R */
  if (ID in R.IDList)
    {
      dxR = pop(DS[ID]);
      push(-dxR, DS[ID]);
    }
```

3.5 The power operator

Again, the result of the to-the-power operator (L^R), with an ID equal to L.ID $\cup$ R.ID is pushed on the VMS using the code:

```
Expr(L,R) {return L^R};  /* Compute the value for
ComputeResult(L,R,Expr);    the power operator */
```

The partial derivative of the expression with respect to the common variable (here x) is computed as:

$$\frac{\partial}{\partial x}\left(f(x,a)^{g(x,b)}\right) = f(x,a)^{g(x,b)}\left[\frac{g(x,b)}{f(x,a)}\frac{\partial f(x,a)}{\partial x} + \log(f(x,b))\frac{\partial g(x,b)}{\partial x}\right] \tag{4}$$

The pseudo code for this operation is:

```
Expr(L,R,dxL,dxR) {return (L^R)*((R/L)*dxL + log(L)*dxR)};
dCommonVar(L,R,Expr);
```

The partial derivatives with respect to the non-common set of IDs correspond to $g(x,b)f(x,a)^{[g(x,b)-1]}\frac{\partial f(x,a)}{\partial a}$ and $f(x,a)^{g(x,b)}\log(f(x,b))\frac{\partial g(x,b)}{\partial b}$. Partial derivatives of f and g with respect to a and b are computed using the functions `LExpr` and `RExpr`. The computed partial derivatives are pushed back on the appropriate `DS`s. The pseudo code for this operation is:

```
LExpr(L,R,dx) {return R*(L^(R-1))*dx};
RExpr(L,R,dx) {return (L^R)*log(L)*dx};
dNoncommonVar(L,R,LExpr,RExpr);
```

At the terminal operators (e.g. the assignment operator '='), a single value (the result of the right hand side of the terminal operator) is poped from the `VMS`. The propagated error is then computed using the values from the top of all `DS`s corresponding to the IDs in the ID list of the poped value. The values from these `DS`s are the partial derivatives of the expression with respect to the various independent variables used in the expression on the right hand side ($\partial f/\partial x_i$ in Eq. 1). The corresponding measurement errors (δx_i in Eq. 1) are in the appropriated locations in the `ME` table. Using these values, Eq. 1 is evaluated. This is the final propagated error in the expression.

4 Examples

Following are some examples to demonstrate as well as test the correctness of the error propagation algorithm of Section 3. In the following examples, various functions are written in different algebraic forms and the results for the different forms is shown to be exactly same (e.g. $\cos(x)$ vs. $\sqrt{1-\sin^2(x)}$, $\tan(x)$ vs. $\sin(x)/\cos(x)$). These examples also verify that the combination of a function and its inverse simply returns the argument (e.g $asin(\sin(x)) = x$), as well as functions like $\sinh(x)/((\exp(x)-\exp(-x))/2)$ (which is really a complicated way of writing 1!) returns a value of 1 with no error. However, if the values of two independent variates x_1 and x_2 and their corresponding errors are same, the value of expressions like $\sin^2(x_1)+\cos^2(x_2)$ will be 1 but the error will not be zero.

```
Value of x          =  1.00000 +/- 0.10000
```

```
Value of y          =  2.00000 +/- 0.20000
Value of x1         =  1.00000 +/- 0.10000
Value of x2         =  1.00000 +/- 0.10000

sin(x)              =  0.84147 +/-  0.05403
sqrt(1-sin(x)^2)    =  0.54030 +/-  0.08415
cos(x)              =  0.54030 +/-  0.08415

tan(x)              =  1.55741 +/-  0.34255
sin(x)/cos(x)       =  1.55741 +/-  0.34255

asin(sin(x))        =  1.00000 +/-  0.10000
asinh(sinh(x))      =  1.00000 +/-  0.10000
atanh(tanh(x))      =  1.00000 +/-  0.10000
exp(ln(x))          =  1.00000 +/-  0.10000

sinh(x)             =  1.17520 +/-  0.15431
(exp(x)-exp(-x))/2  =  1.17520 +/-  0.15431

sinh(x)/((exp(x)-exp(-x))/2) =  1.00000
x/exp(ln(x))                 =  1.00000

sin(x1)*sin(x1)       =  0.70807 +/- 0.09093
sin(x1)*sin(x2)       =  0.70807 +/- 0.06430
sin(x1)^2+cos(x1)^2   =  1.00000 +/- 0.00000
sin(x1)^2+cos(x2)^2   =  1.00000 +/- 0.12859
```

4.1 Recursion

Following is an example of error propagation in a recursive function. The factorial of x is written as a recursive function $f(x)$. Its derivative is given by $f(x)\left[\frac{1}{x}+\frac{1}{x-1}+\frac{1}{x-2}+\cdots+\frac{1}{2}+1\right]$. The term in the parenthesis is also written as a recursive function $df(x)$. It is shown that the propagated error in $f(x)$ is equal to $f(x)df(x)\delta x$.

```
>f(x) {if (x==1) return x; else return x*f(--x);}
>df(x){if (x==1) return x; else return 1/x+df(--x);}
>f(x=10pm1)
        3628800.00000 +/- 10628640.00000
>(f(x)*df(x)*x.rms).val
        10628640.00000
```

Figure 2: The parsing tree for $f(x_1, x_2) = \frac{x_1 \times x_2 + x_1}{x_2}$

Similarly, the recurrence relations for the Laguerre polynomial of order n and its derivative evaluated at x are given by

$$L_n(x) = \begin{cases} 1 & n = 0 \\ 1 - x & n = 1 \\ \frac{(2n-1-x)L_{n-1}(x)-(n-1)L_{n-2}(x)}{n} & n \geq 2 \end{cases} \quad (5)$$

$$L'_n(x) = (n/x)\left[L_n(x) - L_{n-1}(x)\right] \quad (6)$$

These are written as recursive functions `l(n,x)` and `dl(n,x)` and it is shown that the propagated error in $L_n(x)$ is equal to $L'_n(x)\delta x$.

```
>l(n,x){
   if (n<=0) return 1;
   if (n==1) return 1-x;
   return ((2*n-1-x)*l(n-1,x)-(n-1)*l(n-2,x))/n;
 }
>dl(n,x){return (n/x)*(l(n,x)-l(n-1,x));}
>l(4,x=3pm1)
            1.37500 +/-    0.50000
>(dl(4,x)*x.rms).val
        0.50000
```

APPENDIX

A An example of a multi-variate expression

Parsing tree for the multi-variate expression $f(x_1, x_2) = \frac{x_1 \times x_2 + x_1}{x_2}$ is shown in Fig. 2. The sequence of operations at the stages marked by **1,2** and **3** are as follows. In the following, <variablename>.rms() and <variablename>.ID refers to the random error and the ID associated with the variable respectively.

1. Stage **1**:

   The IDs of x_1 and x_2 are set to 0 and 1 respectively and they are pushed on the VMS. x_1.rms() and x_2.rms() are copied in ME[0] and ME[1] respectively, while a value of 1 is pushed on DS[0] and DS[1].

   Operator '*' pops two values (say L and R) from the VMS. A value L*R with an ID equal to L.ID $\cup$ R.ID is pushed on the VMS and a list of common IDs is made as IDList = L.IDList $\cap$ R.IDList (here, an empty list). Next, for all IDs in L, a value is poped from DS[L.ID] (say, dx), and dx*R is pushed back on DS[L.ID]. Similar operation is done for all IDs in R.

   Top of DS[0] and DS[1] now has x_2 and x_1 respectively, while the top of the VMS has the value $x_1 \times x_2$ with an IDList={1,0}.

2. Stage **2**:

   x_1 is pushed on the VMS, its ID is set to zero, x_1.rms() is copied to ME[0] (a redundant operation) and a value of 1 is pushed on DS[0].

   Operator '+' pops two values (L and R) from the VMS (these have values $x_1 \times x_2$ and x_1). L+R with an IDList= L.ID $\cup$ R.ID is pushed back on the VMS. This IDList will be {0,1}. For IDs common between L and R (here {0}), two values are poped from the DS, and their addition pushed back on the DS. The common ID is then removed from the ID list of both operands. For the remaining IDs in L (R has no IDs left), values are poped from the corresponding DS, added together and pushed back on the appropriate DS.

   Hence, at the end of the operator '+', DS[0] will have x_2+1 and DS[1] will have x_1. Top of the VMS has the value $x_1 \times x_2 + x_1$.

3. Stage **3**:

   x_2 is pushed on the VMS, x_2.rms() is copied in ME[1] (another redundant operation) and a value of 1 is pushed on DS[1].

   Operator '/' pops two values (L and R) from the VMS. A value of L/R with an IDList=L.ID $\cup$ R.ID ($\{0, 1\} \cup \{0\}$) is pushed on the VMS. A list of common IDs between L and R is made using IDList = L.IDList $\cap$ R.IDList. For all IDs in IDList, two values are poped from the corresponding DS (say, dxR (=x_1) and dxL (=1)) and (R*dxL - L*dxR)/(R*R) pushed back on the same DS. The corresponding IDs are then removed from L and R.

   Next, for all the remaining IDs in L, a value is poped (say dx) from DS[L.ID] and dx/R is pushed back on the same DS. Similarly, for all IDs in R, a value from DS[R.ID] is poped into dx and -L*dx/(R*R) is pushed back on the same DS.

   At the end of this stage, DS[0] has the value $1+1/x_2$ and DS[1] has the value $-x_1/x_2^2$. VMS now has $\frac{x_1 \times x_2 + x_1}{x_2}$.

Finally, the values from the top of *all* DSs are multiplied with corresponding values in ME table, squared, added and the square root of the final result is taken. That value will be

$\sqrt{\left[\left(1+\frac{1}{x_2}\right)\times\delta x_1\right]^2+\left[-\frac{x_1}{x_2^2}\times\delta x_2\right]^2}$. This is the final error propagated through the expression.

B Syntax

This appendix describes the `fussy` syntax. Statements are interactively executed as soon as they are completed. The virtual code for the sub-programs (function or procedure) is held in the memory and executed when the sub-programs are called.

B.1 Numbers

Numbers in `fussy` are represented as floating point numbers and can be specified with or without the decimal point, or in the exponent format. Optionally, an error can also be associated with the numbers via the `pm` directive. E.g., 75.3 ± 10.1 can be expressed as `75.3pm10.1`. Numbers can also be tagged with units (see Section B.1.1) or a `C`-styled printing format (see Section B.9).

B.1.1 Units

Numerical values can be specified along with their units. As of now, the only units supported are degree, arcmin, arcsec, hours, minute, and seconds. These can be specified by appending `'d', ''', '"', 'h', 'm', 's'` respectively to the numeric values. Internally, all numeric values are always stored in the MKS system of units. The default units for a variable used to specify angles or time is radians. If the values are specified along with any of the above mentioned units, the values are still stored internally as radians. However while printing (see Section B.8), the values are formatted automatically and printed with the appropriate units.

B.2 Operators and built-in functions

The normal binary operators of type `expr` <op> `expr`, where `expr` is any expression/variable/constant and <op> is one of `'+', '-', '/', '*'`, '^' and `'**'` binary operators perform the usual mathematical operations in addition to error propagation. The comparison operators '<', '>', '=', '`!=`', '<=', '>=' and the logical operators '`||`' and '`&&`' have the usual meaning. Apart from the usual operation, the `var=expr` assignment operator also does the error propagation in the expression on the RHS and assigns it as the error for the variable on the LHS. In addition to this, the assignment operator for partial variables (`pvar:=expr`) is also defined. This does not propagate the errors on the RHS but instead transfers all the required information for error propagation to the variable on the LHS (see Section B.4). The result of these assignment statements is the value of the variable on the LHS. Hence expressions like `sin(x=0.1pm0.02)` are equivalent to `'x=0.1pm0.02;sin(x);'`. The prefix and postfix operators <op>`var` and

`var<op>` where `<op>` is either `'++'` or `'--'` and `var` is any user defined variable are also defined. These increment or decrement the value of the variables by one. The prefix and postfix operators operate on the variables before and after the variable is further used respectively.

In addition, two operators of type `expr.<op>` where `<op>` is either `val` or `rms` are also defined. These operators extract the value and the associated (propagated) error in `expr` which can be any mathematical expression or a variable.

B.3 Expressions/Statements

Numbers and variables can be combined with the mathematical operators and logical operators to form an expression. Expressions can be used as arguments to built-in or user defined functions (see Section B.6). An expression followed by a NEWLINE prints its result on the output stream (see Section B.8) in the default format (see Section B.9).

For the purpose of error propagation, the print statement and the assignment operator (the "=" operator but not the ":=" operator; see Section B.4) are treated as the terminal nodes of the parsing tree which invokes the final error propagation.

Assigning a value to a variable also creates the variable. The type of the value assigned to the variable determines its type (and overrides the value or the type of a previously declared variable). E.g.

```
>H_0=75pm10
>H_0
          75.00000 +/-   10.00000
>H_0="The Hubble constant\n"
>H_0
 The Hubble constant
```

A semi-colon (`';'`) is a delimiter to separate multiple expressions in a single line. Statements on separate lines need not be delimited by semi-colons (though it is not an error to do so). Compound statements are a group of simple statements, grouped using the curly-brace pair (`'{'` and `'}'`) (e.g. `{a=1.5; b=2;}`). As may be obvious, compound statements can also be nested. The `'/*'` and `'*/'` pair can be used as comment delimiters. Comment delimiters however cannot be nested.

B.4 Sub-expressions

A special assignment operator ':=' is used to assign sub-expressions to user defined variables. Sub-expression variables are different from normal variables in that their propagated error is computed on-the-fly when required, i.e. when they are printed or are assigned to a normal variable using the '=' operator or at an operator node of a parsing tree when used in another expression. E.g.

```
>x=1pm0.1
>s:=sin(x);c:=cos(x);
>sin(x)/cos(x) /* Compute tan(x) as sin(x)/cos(x) */
    1.55741 +/-    0.34255
>s/c           /* Compute tan(x) using two PARTIAL_VAR */
    1.55741 +/-    0.34255
>tan(x)        /* Direct computation of tan(x) */
    1.55741 +/-    0.34255
>s2=s;
>s2/c          /* Compute tan(x) with a normal variable
                  and one PARTIAL_VAR.  Error propagates
                  differently */
    1.55741 +/-    0.26236
```

B.5 Variables and function/procedure names

Variable/function/procedure names can be of any length and must match the regular expression `[a-zA-Z_]+[a-zA-Z0-9_]*`. That is, the names must start with an alphabet or `'_'` and can be followed by one or more alpha-numeric characters or `'_'`.

B.6 Function/procedure

Sub-programs can be written as functions or procedures. The only difference between functions and procedures is that functions *must* return a value while procedures must *not* return a value. The type of a sub-program which returns a value using the `return` <`expression`> statement becomes `func`. If `return` is not used, or is used without an `expression`, the type becomes `proc`. The type of the sub-program therefore need not be declared. It is an error to use a procedure in an expression or pass a procedure as an argument to another sub-program where a function should have been passed.

A function or procedure declaration begins with a variable name followed by an argument list. The argument list is enclosed by a round bracket pair (`'('` and `')'`). A `'()'` specifies an empty argument list. The function body is in enclosed between the `'{'` and `'}'` brackets. E.g.

```
>/* An example of a funtion declaration */
>f() { return sin(PI/2); }
>/* An example of a procedure declaration */
>p() {print "Value of f() = ",f(),"\n";}
>f()
          1.00000
>p()
Value of f() =    1.00000
```

A sub-program can be passed as an argument to another sub-program. An argument corresponding to a sub-program can be specified using the `func` (for a function) or `proc` (for a procedure) directive. E.g.

```
>f(x) { return sin(x); }
>p(func fa,x) {print "The value of f(",x%5.2f,") =",fa(x),"\n";}
>p(f,10)
The value of f(10.00) =  -0.54402
```

All symbols (variables, functions, procedures) used in the sub-program code must be either global variables declared *before* the sub-program declaration or must be one of the argument list. Temporary variables, the scope of which is within the sub-program only, can be declared using the `auto` directive. E.g.

```
>f(x) { return sin(x); }
>p(func fa,x)
  {
    auto t;
    t=fa(x);
    print "The value of f(",x%5.2f,") =",t,"\n";
  }
>p(f,10)
The value of f(10.00) =  -0.54402
```

B.7 Control statements

The `if-else`, `while-` and `for-`loops constitute the program control statements. These loops can be broken at any stage with the use of the `break` statement. As of now, the conditions which control the logic is evaluated ignoring the error with the control variables. Ultimately the goal is to provide a language feature to specify a significance level and the conditional statements return true if the error on the evaluated value is within the significance level, else return false.

B.7.1 `if-else`

The syntax for the `if-else` statement is:

```
if (condition)
   if-body-statment;

     or

if (condition)
```

```
if-body-statment else
else-body-statment;
```

The `if-body-statement` and the `else-body-statement` can be any valid compound or simple statement. In case of a simple statement, the terminating semi-colon is necessary.

B.7.2 `while-loop`

The syntax for the `while-loop` is:

```
while (condition)
   body-statment
```

The `body-statement` can be either a simple or a compound statement and in case it is a simple statement, the terminating semi-colon defines the end of the loop.

B.7.3 `for-loop`

The syntax for the `for-loop` is:

```
for (init;condition;incr)
  body-statment
```

where `init` is a comma (`','`) separate list of simple statements for initializing the loop variables. E.g. `init` can be `i=0,j=0,k=0`. `condition` is a simple, single statement while `incr` is a list of comma separated statement(s). The `body-statement` can be any valid simple or compound statement. `init` statements are executed first followed by the `condition` statement. If the result of the `condition` statement is non-zero (logical true), the `body-statements`, the `incr` statement(s) and the `condition` statements are executed in a sequence till the result of the `condition` statement is zero (logical false). E.g. following is a valid `for-loop` with 3 loop-variables, only one of which is checked in the condition:

```
for (i=0,j=0,k=0;i<10;i=i+1,j=j+1;k=k+1)
   print "i= ",i," j= ",j," k= ",k,"\n";
```

B.8 Print statement

The `print` statement takes a comma separated list of objects to be printed. These objects can be quoted-strings, variables, constants, condition statements or user defined function names. The list can consist of any number of objects and is terminated by a semi-colon. The format in which the numeric values are printed is defined by the format modifier associated with the values (see Section B.9). All escaped-characters used in C-styled printing have the same effect as in the output of the C-styled `printf` statement.

B.9 Formatting

Values can be formatted for printing in a variety of ways. The format in which a variable is printed is associated with the variable and consists of a `printf` styled formatting string (with extensions for specifying the units of the numerical values as well). E.g., if `x=75pm10`, by default `x` will be printed using the `'%10.5f'` format. The default print format can be modified using the `'.'` operator on a variable. E.g., one can fix the default print format of `x` to `'%5.2f'` by `x.=%5.2f`.

The print format of a value can also be temporarily modified by specifying the format along with the variable/value. E.g. the value of `x` can be printed in the exponent format as `print x%E` or in the in hexadecimal format as `print x%x`.

An extra formatting, not available in `printf` formatting, is that of printing the individual bit values using the `%b` format. With this, the value is printed in binary (1 or 0) format. `%B` does the same thing except that it prints a space after every 8 bits. The value is casted into a `unsigned long` integer before printing.

```
>x=10;x%B
     00000000 00000000 00000000 00001010
```

If the units of a value are specified, the print format is also appropriately modified. If a variable has units of time or angle, its print format is automatically set to `%hms` or `%dms` and are printed in the `XXhXXmXX.XXs` and `XXdXX'XX.XX"` styles respectively.

ACKNOWLEDGEMENTS

I thank D. Oberoi and R.K. Singh for many useful discussions. This work was started and largely done while I was working at the National Center for Radio Astrophysics (NCRA), Pune of the Tata Institute of Fundamental Research (TIFR), Mumbai, India and continued at my current position at the National Radio Astronomy Observatory (NRAO), Socorro, USA. All of this work was done on computers running the GNU/Linux operating system and I wish to thank the numerous contributors to this software.

References

Barton, J. J. & Nackman, L. R. 1994, Scientific and Engineering C++: an introduction with advanced techniques and examples (Reading, Mass.: Addison-Wesley Publishing Company)

Bischof, C. H., Roh, L., & Mauer, A. 1997, Software–Practice and Experience, 27, 1427

Gillespie, D. 1992, GNU Emacs Calc Manual: For Calc Version 2.02 (59 Temple Place, Suite 330 Boston, MA 02111, USA.: Free Software Foundation)

Griewank, A., Juedes, D., & Utke, J. 1996, ACM Transactions on Mathematical Software, 22, 131

Harrison, D. 1995, Experimental Data Analysis (EDA) tool for Mathematica, http://www.upscale.utoronto.ca/GeneralInterest/~Harrison/ErrProp.html

Hearn, A. C. 1971, in Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, 128–133

Hearn, A. C. 1995, REDUCE User's Manual Version 3.6 (RAND Publication CP78 (Rev. 7/95))

Stoutemyer, D. R. 1977, ACM Transactions on Mathematical Software, 3, 26

Tsukanov, I. & Hall, M. 2003, International Journal for Numerical Methods in Engineering, 56, 1949