GNU fussy

A calculator language with automatic error propagation
Fussy Version 2.0

Author: Sanjay Bhatnagar

This is a user manual for the GNU Fussy calculator language

Table of Contents

1 Introduction

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 GNU 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 run time 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 as single atomic expressions (Section Section 4.6 [Expressions/Statements], page 8), sub-expressions (Section Section 4.7 [Sub-expressions], page 8), or as sub-program units (functions or procedures; Section Section 4.9 [Function/procedure], page 9). 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 the `C` programming language. 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 programming 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.

1.1 Mathematical background

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 $\vec{x}$ is given by

$$\delta f = \sqrt{\left|\vec{\nabla} f \cdot \vec{\delta x}\right|^2} = \sqrt{\sum_i \left(\frac{\partial f}{\partial x_i}\delta x_i\right)^2}$$

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:

$$\frac{\partial f}{\partial x_i} = \frac{\partial g}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial x_i}$$

Therefore to compute δf one requires:

1. the partial derivative of the function with respect to each independent variable ($\partial f/\partial x_i$)

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

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).

The GNU fussy interpreter is implemented internally as a stack-based virtual machine (VM). The derivative chain rule is implemented using a separate VM which maintains a 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 δf (the first equation) above. 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.

2 Usage

GNU fussy can be run from the command-line with the following options:

```
fussy [-h|--help] [-q] [-d] [-t N] [prog1,prog2,...]

-h or --help
   Print this help

-q
   Run in quiet mode.  Do not print the copyright information

-d
   Sets the debugging mode (meant for developers)

-t N
   N is the number of Ctrl-C trials after which the interpreter
   gives up preaching good behavior and quits.  Default is 10000.
```

Use the `"help"` command in interactive mode to get more help about the language syntax.

The file `$HOME/.fussy`, if present, is processed at the very beginning of the start of the program.

2.1 Environment variables

fussy is sensitive to the following environment variables:

1. `HOME`: The path where it looks for the `.fussy` configuration file
2. `FUSSY_IGNOREEOF`: An integer value indicating the number of times a `Ctrl-D` input to the interpreter will be ignored before quitting. Default values is 10^6.

3 Examples

Following are some examples to demonstrate, as well as test the correctness of the error propagation algorithm.

3.1 Algebraic forms

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 $\arcsin(\sin(x)) = x$), 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

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

sin(x1)*sin(x1)       =  0.70807 +/- 0.09093
sin(x1)*sin(x2)       =  0.70807 +/- 0.06430

/* Expressions that evaluate to just x */

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

/* Complicated ways of computing 1.0! */

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

/* Complicated ways of computing 1.0 with single and multiple variates! */
```

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

3.2 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
```

Similarly, the recurrence relations for the Laguerre polynomial of order n and its derivative evaluated at x can be written as recursive functions. These are written in GNU fussy as `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
```

4 Syntax

This section describes the GNU 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.

4.1 Numbers

Numbers in GNU 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 Section 4.2 [Units], page 6) or a `C`-styled printing format (see Section Section 4.11.2 [Formatting], page 11).

4.2 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 Section 4.11 [Print statement], page 11), the values are formatted automatically and printed with the appropriate units.

4.3 Operators

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 Section 4.7 [Sub-expressions], page 8). 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.

4.4 Built-in functions and constants

The following built-in functions are available:

`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `exp`, `ln`, `log`, `fabs`, `fmod`, `sqrt`, `int`.

The following functions, useful for astronomical computations are defined. The latitude and longitude used for these computations are set in the global system variables `LONGITUDE` and `LATITUDE`.

- `time()`: returns the current time in the `hms` format.
- `lst()`: returns the Local Sideral time in the `hms` format.
- `day()`: returns the current day.
- `month()`: returns the current month.
- `year()`: returns the current year.
- `mjd(),fmjd()`: returns the current MJD and fractional MJD.
- `setlong(MyLongitude)`: Sets the global variables `LONGITUDE` to the given value.
- `setlat(MyLatitude)`: Sets the global variables `LATITUDE` to the given value.

4.4.1 Built-in constants

The following useful constants are available:

- `PI`: The value of PI
- `C`: The speed of light in meter/second
- `R2D`: Factor to convert angles in degree to radian
- `A2R`: Factor to convert angles in arcsecond to radian
- `kb`: The Boltzmann constant
- `PC2M`: Factor to convert from Parsec to meter
- `PC2LY`: Factor to convert from Parsec to Lightyear
- `AU2M`: Factor to convert from Astronomical Unit to meter
- `sigma`: [NOT YET USED] Threshold to use for the result of logical operators

4.5 Special operators

The fussy language defines the following special operators:

1. '`:=`': For assignment of partial results/sub-expression values.

   The partial assignment operator `:=` assigns value of partial results to variables. Expression like `pvar:=val` does not propagate the errors on the `val` but instead transfers all the required information for error propagation to the variable `pvar` (see Section Section 4.7 [Sub-expressions], page 8). Expressions like `sin(x:=0.1pm0.02)` are equivalent to `x:=0.1pm0.02;sin(x);`.
2. '`pm`': For associating an error with numerical values. E.g. 10 +/- 1.0 is expressed as 10pm1.
3. '`<expr>.rms`', '`<expr>.val`': For extracting the associated error and the value of the expression <expr>. E.g. x.rms is the error associated with x while x.val is the value of x.

4. '<expr>%<format>': Sets the print format of the result of the expression <expr> to the printf style format <format>. E.g. x%10.5f will print the value of x as a float in a 10 character field with 5 places after decimal.
5. '<var>.': Operator to set the default print format of a variable. E.g. x.=%7.2f will replace the default printf format (%10.5f) by %7.2f.

4.6 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 Section 4.9 [Function/procedure], page 9). An expression followed by a NEWLINE prints its result on the output stream (see Section Section 4.11 [Print statement], page 11) in the default format (see Section Section 4.11.2 [Formatting], page 11).

For the purpose of error propagation, the print statement and the assignment operator (the '=' operator but not the ':=' operator; see Section Section 4.7 [Sub-expressions], page 8) 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. %E.g.

4.7 Sub-expressions

The 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
```

4.8 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 `'_'`.

4.9 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 function 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
```

4.10 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.

4.10.1 `if-else`

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

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

     or

if (condition)
   if-body-statement else
   else-body-statement;
```

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.

4.10.2 `while-loop`

The syntax for the `while-loop` is:

```
while (condition)
   body-statement
```

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.

4.10.3 `for-loop`

The syntax for the `for-loop` is:

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

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";
```

4.11 Print statement

GNU fussy supports formatted printing of numbers, variables and strings using `printf`-styled print-format specification.

4.11.1 The `print` and `printn` statements

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 Section 4.11.2 [Formatting], page 11). All escaped-characters used in C-styled printing have the same effect as in the output of the C-styled `printf` statement.

The `printn` statement is exactly the same as `print` statement but emits a new-line character at the end.

4.11.2 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 caste 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.

5 General interactive commands

Following are some commands useful in an interactive session:

- `quit/bye`: The only two ways to quit from the interpreter. Politer (more civilized) of the two commands is more recommended! Typing `Ctrl-D` will exit the interpreter after number of trials given by the environment variable `FUSSY_INGNOREEOF` (see Section Section 2.1 [Environment variables], page 3) and typing `Ctrl-C` will attempt to teach a thing or two about life!
- `setfmt`: Set the system parameter 'fmt' used as the default format for printing numbers.
- `warranty`: Prints the warranty information.

5.1 Commands for the Virtual Machine (VM)

- `showvm`: Prints the resident VM program as op-codes.
- `showsym`: Prints the symbol table of variables.
- `showcsym`: Prints the symbol table of constants.
- `showid`: Prints the list of allocated IDs.

Index