Stochastic Arithmetic in Multiprecision
The SAM library

LIP6, Sorbonne Université, CNRS
Paris, France

Contents

Chapter 1

Introduction

1.1 Aim of the SAM library

The arithmetic commonly used on computers for scientific programming is floating point arithmetic. This arithmetic only approximates exact arithmetic. Consequently each arithmetic statement generates a round-off error. So when a correct program with regard to syntax and logical organization is running on a computer, every produced result is unavoidably given with a so called "computing error". This error is due to all the round-off errors produced along the elementary statements required to obtain the result. Sometimes the error may be such that the final result is really wrong (and not only inaccurate).
The aim of the SAM library presented here is to answer the following question:

What is the computing error due to floating point arithmetic on the results produced by any program running on a computer?

So, we want to estimate the round-off error on each result with a technique which is independent on the program and hence on the algorithm used.

SAM is a library, based on the MPFR library. More precisely, SAM is a set of data types, functions and subroutines that may be used in any program written in C/C++. It implements the CESTAC method in a synchronous way (the Discrete Stochastic Arithmetic DSA). With a few modifications in the source code, this library has for main purpose to estimate the effects of round-off error propagation on every numerical computed result. It also allows to study the effects of the initial data uncertainties upon computed results, as described in 2.7.

This implementation consists in replacing the computer deterministic arithmetic by a stochastic arithmetic (the Discrete Stochastic Arithmetic DSA) and in performing N times ($N = 3$) each elementary operation before executing the next statement.

Thus, it is as N identical programs were simultaneously running on N synchronized computers each of them using random arithmetic. So for each result, we obtain N samples from which we compute the mean value and the standard deviation which characterize the corresponding stochastic number. The value of this number is defined as the mean value of the different samples. The accuracy of this number, *i.e.* its number of exact significant digits, is estimated using the mean value and the standard deviation. If all the samples are equal to zero or if the number of exact significant (decimal) digits is less than one, then the number is defined as a computational zero [13]. This means that a computational zero is either the mathematical zero or a number without any significance.

So round-off error propagation can be analyzed step by step. Numerical instabilities and non significant results are detected. The branchings based on order relations may also be controlled. Therefore, this synchronous implementation of the CESTAC method allows to validate any scientific code during its run.

With the SAM library, one can run any scientific code using random arithmetic, without having to rewrite or notably change the initial code. This tool has been written in C++. This language enables to create new numerical **types** with their operators; furthermore the designating symbol of an operator can be chosen among the primitive symbols in the language (+, $*$,...). In other words, this language enables the so called "operator overloading". Thanks to these new properties, SAM has been developed for C/C++ programs.

Thus a new numerical type has been created, the **stochastic number**; it is nothing else than an N-set ($N = 3$) containing perturbed floating-point values (of type $mpfr\_t$). All the arithmetic operators (+, −, $*$, /) have been overloaded in such a manner that when an operator is used, the operands are N-sets and the returned result is a randomly perturbed N-set. The relational operators ($>$, $\geq$, $<$, $\leq$, $==$, $\neq$) are overloaded. All standard functions defined in "math.h" (sin, cos, exp, ...) have also been overloaded. Likewise, in/out statements have been modified, mainly the printing statement which gives as a result the mean value of the N-set written with only its exact significant (decimal) digits.

Furthermore, in order to enable the evaluation of the weight of uncertainties

on initial data on the results, a function called data_st may be used to perturb data as exposed in 2.7.8.

During the run of a program, as soon as a numerical anomaly (for example the product of non-significant numbers, or a relational test involving a non-significant result) is produced, some special counters are updated. At the end of the run, all information about numerical anomalies is printed on the standard output.

If no anomaly has been detected, it means that the program runs without any numerical problem. Results are then given with their accuracy - number of exact significant (decimal) digits.

If some numerical anomalies have been detected, they must be analysed. Helped by the debugger associated with the compiler, the user may retrieve the statements that produced the anomalies and determine if changes in the code are required.

The stochastic types and the overloaded or newly defined functions of the library are presented in the next sections.

1.2 The DSA (Discrete Stochastic Arithmetic)

1.2.1 The CESTAC method

The CESTAC (Contrôle et Estimation Stochastique des Arrondis de Calcul) method, which has been developed by La Porte and Vignes [14, 9, 8], enables one to estimate the number of exact significant digits of any computed result.

The basic idea of the method is defined in [6, 7] and consists of the following:

- to perform the same code N times with a different round-off error propagation for each run,
- to estimate the common part of these results and to consider that this part is representative of exact result.

In practice, these different round-off error propagations are obtained by using the random rounding mode defined below.

Each result ρ of a floating-point operation (assignment, arithmetical operation) which is not an exact floating-point value, is bounded by two floating-point values, one by default ρ^- and the other by excess ρ^+.

The random rounding mode consists, at the level of each floating-point operation or assignment, in choosing as a result randomly with an equal probability either ρ^- or ρ^+.
With this random rounding mode, the same program run several times provides different results, due to different round-off errors.

Let us consider a sequence of computations providing an exact result r. When this sequence is performed with the CESTAC [3, 11, 9] method, N results R_k, $k = 1,...N$ are obtained. From the formalization of the round-off errors of the floating-point arithmetic operations $(+, -, *, /)$ a probabilistic model for estimating the round-off error on the mean value $\overline{R}$ of the R_k, considered as the computed result, has been established. This model is a first order model. It means that the terms in 2^{-2p} (p being the number of bits of the mantissa) which appear in the expression of the round-off error of the floating-point multiplications and divisions have been neglected. Only the terms in 2^{-p} are considered.
This model is based on two hypotheses.

- Hyp1.: The elementary round-off errors α_i of the floating-point arithmetic operations are random independent, centered and uniformly distributed variables.

- Hyp2.: The approximation of the first order in 2^{-p} is legitimate.

It has been proved that if the two hypotheses hold then the R_k, $k = 1,...N$ are samples of the Gaussian distribution, centered on the exact result r. Thus it is possible to use the Student's test which allows to obtain a confident interval of $\overline{R}$ with a $(1 - \beta)$ probability and then to estimate the number of exact significant digits of $\overline{R}$ by the formula

$$C_{\overline{R}} = \log_{10} (\frac{\sqrt{N}|\overline{R}|}{\tau_\beta \sigma}) \tag{1.1}$$

with

$$\overline{R} = \frac{1}{N} \sum_{i=1}^{N} R_i$$

and

$$\sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} (R_i - \overline{R})^2.$$

τ_β is the value of the Student's distribution for $N-1$ degrees of freedom and a probability level $1-\beta$. In practice $N=3$, $\beta=0.05$ and then $\tau_\beta = 4.303$.

The result provided by eq(1.1) is reliable when the two previous hypotheses hold in practice [11, 15].

- Concerning Hyp1, with the use of random rounding, the α_i are truly independent random variables. However they are not exactly centered, consequently the $\overline{R}$ is biaised. But because of the robustness of Student's test, Hyp1 still holds. This hypothesis is not an inconvenience for the reliability of eq(1.1).

- Concerning Hyp2, it holds if the terms in 2^{-2p} are negligible in comparison to the terms in 2^{-p}. It has been proved that this fact is satisfied if

  - the operands of any multiplication are both significant
  - the divisor of any division is significant.

It is then absolutely necessary to control these two points during a run of code. Indeed if they are not satisfied, this means that the Hyp2 has been violated and then the results obtained with eq(1.1) must be considered as not reliable.
This control is done with the concept of computational zero, also named informatical zero, computed zero, or stochastic zero [13].

1.2.2 The computational zero

Each result provided by the CESTAC method is a "computational zero" denoted by @.0 if one of the two following conditions holds:

- $\forall i, i = 1, ...N, R_i = 0$

- $C_{\overline{R}} \leq 0$ ($C_{\overline{R}}$ obtained with eq(1.1))

When $C_{\overline{R}} \leq 0$, then $\overline{R}$ is an insignificant value.
From the concept of computational zero, also named informatical zero, discrete stochastic relations have been defined (equality and order relations).

1.2.3 Discrete stochastic relations

Let X and Y be N-samples provided by CESTAC method.

- Discrete stochastic equality denoted by $s=$ is defined as:
  $Xs = Y$ if $X - Y = @.0$
- Discrete stochastic inequalites denoted by $s>$ and $s\geq$ are defined as:
  $Xs > Y$ if $\overline{X} > \overline{Y}$ and $X - Y \neq @.0$
  $Xs \geq Y$ if $\overline{X} \geq \overline{Y}$ or $X - Y = @.0$

The Discrete Stochastic Arithmetic (DSA) [1, 15, 5] is defined from the CESTAC method, the concept of informatical zero and the discrete stochastic relations. With this DSA, it is possible to control the run of a scientific code, to detect the numerical instabilities and the violation of the hypotheses underlying the method.

Chapter 2

Reference guide

2.1 SAM types

SAM provides new numerical types (*stochastic types*) associated to a mantissa-length chosen by the user. When a SAM variable is declared, the number of bits of its mantissa must be given. For instance,

```
mp_st<122> res;
```

enables one to declare a stochastic variable named res with a 122 bit-long mantissa. Its precision (122 bits) will not change. Note that double variables are 53-bit mantissa length numbers and that float variables are 24-bit mantissa length numbers according to the IEEE standard floating-point arithmetic [16].
A stochastic variable consists in three `mpfr_t` variables and one integer variable to store its accuracy.

2.2 Assignment

The operator "=" is overloaded and accepts stochastic types. It sets a stochastic variable with different types of values: mp_st, float, double, int, unsigned int, long, unsigned long, or MPFR object. If an mp_st, a float, a double, or an MPFR object is set to a longer mp_st variable, then its value is perturbed.

A string can be assigned to an mp_st variable using the sam_set_str function. For instance, the following instructions are valid.

```
mp_st<80>a; //precision of 80 bits
sam_set_str(a,"1.234567");
cout<<a<<endl;
```

The associated output is:

```
0.12345670000000000000000000E+1
```

2.3 Conversion functions

The float, double, int, unsigned int, long, and unsigned long cast operators act on variables of stochastic type and work like for numerical predefined types. Thus the result is of classical type and the knowledge of the accuracy is lost. If X is a stochastic variable consisting in N samples X_i, for instance (int) X is computed as (int)($\frac{\sum_{i=1}^{N} X_i}{N}$).

2.4 Arithmetic operators

Arithmetic operators are overloaded and accept stochastic types and a mixture of classical types and stochastic types.

If the stochastic mp_st operands have different sizes, the arithmetic operation is performed with the largest size, *i.e.* in the greatest precision.

If a double variable d is mixed with `mp_st<N>` variables with `N`$>$ 53, the operations are performed with precision `N`. But only its first 15 decimal digits can be correct. Operations with mp_st variables are preferable, because if they have different sizes adequate perturbations can be performed. More information on the danger of mixing classical types and stochastic types is given in 3.10.
If an arithmetic operation is performed with a float variable f and a stochastic variable, f is casted to a double value, in accordance with the MPFR arithmetic function used.

2.5 Relational operators

Comparison operators are overloaded and accept stochastic types and a mixture of classical types and stochastic types. They take into account the accuracy of the operands. Thus when the expression a == 0.0 is true, it means that a is a *computational zero*, i.e.

- a is a mathematical zero or
- a has no exact significant digit.

Similarly, when the expression a >= b is true, it means that

- a-b is a computational zero or
- $\frac{\sum_{i=1}^{N} a_i}{N} > \frac{\sum_{i=1}^{N} b_i}{N}$,

and, when the expression a > b is true, it means that

- a-b is NOT a computational zero, i.e. has at least one exact significant digit, and
- $\frac{\sum_{i=1}^{N} a_i}{N} > \frac{\sum_{i=1}^{N} b_i}{N}$.

If the stochastic mp_st operands have different sizes, the comparison is performed in the greatest precision.

2.6 Mathematical functions

Mathematical functions have been extended to stochastic types. These are the following functions: fabs, floor, ceil, trunc, nearbyint, rint, lrint, llrint, sqrt, exp, exp2, expm1, log, log10, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, hypot. They accept arguments of mp_st stochastic type.
If mathematical functions have two mp_st stochastic arguments, they must have the same size. Otherwise cast operations are required.
For instance, the following instructions are valid.

```
mp_st<40> x = 1;
mp_st<50> y = 2;
mp_st<40> max;
max=fmax(x,(mp_st<40>)y);
cout << "max=" << max<< endl;
```

The associated output is:

```
max= 0.200000000000E+1
```

2.7 SAM specific functions

The previous part described how some classical C statements are slightly affected when using the SAM tool. Now we present functions that are specific to the library. Note that the functions SAM_init and SAM_end have to appear, respectively to initialize and to close the library. Other functions such as SAM_enable, SAM_disable, self_validation_only, data_st, nb_significant_digit, str and strp can also be used.

2.7.1 Initializing and closing the library

The SAM_init function has to be called once, early in the main program, before any kind of declaration.
This function has four integer arguments:
SAM_init(numb_instability, SAM_instability, cancel_level, init_random).
The first argument must always be present.

The user chooses the maximum number of numerical instabilities that will be detected.

- if numb_instability $= -1$, all the instabilities will be detected
- if numb_instability $= 0$, no instability will be detected
- if numb_instability $=$ M (strictly positive M), the first M instabilities will be detected.

The other arguments are optional.

The second argument allows the user to determine what kind of instabilities will be enabled or disabled.

There are 7 integer parameters in the library:
SAM_BRANCHING,
SAM_CANCEL,
SAM_DIV,
SAM_INTRINSIC,
SAM_MATH,
SAM_MUL,
SAM_POWER.
By default, the detection of all types of instability is enabled. The

user has only to specify what kind of instability is to be **disabled** by passing, as the second argument, the addition of the chosen parameters.

The third argument is an integer which is used to initialize some internal variables for random arithmetic. The default value for this argument is 51.

The fourth argument corresponds to the following. An unstable cancellation is pointed out when the difference between the number of exact significant digits (i.e. digits which are not affected by round-off errors) of the result of an addition or a subtraction and the minimimum of the number of exact significant digits of the two operands is greater than the cancel_level argument. The default value of this argument is 4. In other words, when one loses more than cancel_level significant digits in one addition or subtraction, SAM considers that a catastrophic cancellation has been detected (if the detection of this kind of instability is enabled).

The SAM_end function "closes" the library and prints to the standard output the result of the detection of numerical instabilities.

2.7.2 Printing a stochastic variable

When a stochastic variable is printed, only its exact significant digits appear. Thus its accuracy is easy to read. When the stochastic variable has no exact significant digit, @.0 is printed.

With the printf function, the %s format and the strp SAM function are used.

For C++ programmers, the classical cout and << notations have been overloaded for the stochastic types.

For instance, the following instructions are valid.

```
mp_st<40> x = 123;
printf("x=%s\n",strp(x));
cout << "x=" <<x<< endl;
```

The associated output is:

```
x= 0.122999999999E+3
x= 0.122999999999E+3
```

2.7.3 Printing the triplet associated with a stochastic variable

The display method prints the triplet associated with a stochastic variable. For instance, let d be a multiple-precision stochastic variable. The following instructions

```
printf("%s\n",strp(d));
d.display();
```

may provide

```
0.30E-64
3.0217133019536030e-65 -- 3.0133146666062181e-65 -- 3.0248565827034563e-65
```

The three multiple-precision values associated to d have 2 common significant digits.

2.7.4 Obtaining the triplet associated with a stochastic variable

The getX, getY and getZ methods enable one to get the triplet associated with a stochastic variable.
For instance, the following instructions are valid.

```
mp_st<40> a = 123;
cout << "a=" <<a<< endl;
mpfr_t *x,*y,*z;
x=a.getX();
y=a.getY();
z=a.getZ();
printf("x=");
mpfr_out_str (stdout, 10, 0, *x, MPFR_RNDN);
printf("\ny=");
mpfr_out_str (stdout, 10, 0, *y, MPFR_RNDN);
printf("\nz=");
mpfr_out_str (stdout, 10, 0, *z, MPFR_RNDN);
printf("\n");
```

The associated output is:

```
a= 0.122999999999E+3
x=1.2300000000000e2
y=1.2300000000000e2
z=1.2300000000000e2
```

2.7.5 Obtaining the number of exact significant digits of a stochastic variable

The nb_significant_digit and getAccuracy methods both return an integer giving the number of exact significant decimal digits of a stochastic variable when the method is called.
At some point x.nb_significant_digit() may return 7; later during the run it may return 5. If x becomes non-significant then x.nb_significant_digit() returns 0.
For instance, the following instructions are valid.

```
mp_st<40> a = 123;
cout << "a=" <<a<< endl;
int acc= a.nb_significant_digit();
cout << "acc=" <<acc<< endl;
acc=a.getAccuracy();
cout << "acc=" <<acc<< endl;
```

The associated output is:

```
a= 0.122999999999E+3
acc=12
acc=12
```

2.7.6 Obtaining the precision of a stochastic variable

The getPrecision method returns an integer giving the mantissa length (the number of bits) of of a stochastic variable.
For instance, the following instructions are valid.

```
mp_st<40> a = 123;
cout << "a=" <<a<< endl;
int p=a.getPrecision();
cout << "p=" <<p<< endl;
```

The associated output is:

```
a= 0.122999999999E+3
p=40
```

2.7.7 Testing if a variable is a numerical noise

The numericalnoise method acts on a stochastic variable and returns 1 if it is a numerical noise (*i.e.* it has no exact significant digit), -1 if it is zero (the 3 values that represent the stochastic variable are zero), 0 otherwise.
For instance, the following instructions are valid.

```
mp_st<40> a = 123;
cout << "a=" <<a<< endl;
int na=a.numericalnoise();
cout << "na=" <<na<< endl;

mp_st<40> b = 0;
cout << "b=" <<b<< endl;
int nb=b.numericalnoise();
cout << "nb=" <<nb<< endl;

mp_st<40> c(0.,-1.,1.);
cout << "c=" <<c<< endl;
int nc=c.numericalnoise();
cout << "nc=" <<nc<< endl;
```

The associated output is:

```
a= 0.122999999999E+3
na=0
b= 0.000000000000
nb=-1
c= @.0
nc=1
```

2.7.8 Reducing accuracy of initial data

Initial data are often known with less significant digits than provided by their internal representation. The data_st method allows the user to introduce some effective uncertainties on these data, reducing their initial accuracy. So the accuracy of results depends in some way on the accuracy of initial data.
The data_st method acts on a stochastic variable X and has two optional arguments: X.data_st(ERX,IER);

The first argument is an optional double argument that contains the relative or absolute uncertainty of the stochastic variable X. The second argument determines the kind of the uncertainty: relative or absolute. If X is a stochastic variable and ERX is a double value strictly less than 1, the X.data_st(ERX,IER); instruction modifies the values of the N samples in X according to the following formula:

$$X_i = X_i * (1 + ERX * ALEA) \text{ for } i = 1 \text{ to } N \text{ if IER} = 0$$

$$X_i = X_i + ERX * ALEA \text{ for } i = 1 \text{ to } N \text{ if IER} = 1$$

$ALEA$ is a random variable uniformly distributed between -1 and 1.
If ERX is 0, no perturbation takes place as if the statement was suppressed.
If ERX is absent, perturbation will concern only the last bit of the mantissa.
If IER is absent, it is like $IER = 0$. The data_st method without ERX must be used when data are considered as exact but cannot be exactly coded in the memory.

Chapter 3

User's guide

The use of the SAM library involves seven steps:

- declaration of the SAM library for the compiler,
- initialization of the SAM library,
- substitution of the type float or double by stochastic type mp_st in variable declarations,
- possible changes in the input data if perturbation is desired, to take into account uncertainty in initial values,
- change of output statements to print stochastic results with their accuracy,
- possible use of SAM functions to evaluate the number of exact significant digits,
- termination of the SAM library.

3.1 Declaration of the SAM library

The following pseudo-statement

```
#include <SAM.h>
```

must take place in any file which contains declarations of stochastic variables or SAM functions to be found by the compiler.

3.2 Initialization of the SAM library

The SAM_init function has to be called once, early in the main program, before any kind of declaration, to initialize the random arithmetic. For more information about the arguments of the SAM_init function, see 2.7.1.

3.3 Declaration of variables

3.3.1 Changes in the type of variables

To control the numerical quality of a variable, just replace its standard type by the stochastic type.
Example:

standard declarations	SAM declarations
float a, b;	**mp_st<24> a, b;**
double c;	**mp_st<53> c;**
float d[6], e, f;	**mp_st<24> d[6], e, f;**

3.4 Changes in assignments or arithmetic operations

3.4.1 Conversions between usual types and the stochastic types

In assignment statements, conversions are implicit from C float, double, int, unsigned int, long, or unsigned long types *to* and *from* the mp_st stochastic types (because the = operator is overloaded), **but for conversions from the mp_st stochastic types to standard types, the knowledge of accuracy is lost.**

3.4.2 Classical arithmetic operators

All arithmetic operators involving stochastic variables are overloaded. The result of expressions containing stochastic operands will be of stochastic type. Expressions may contain a mixture of stochastic types and classical types. However, as previously described in 2.4, operations with mp_st variables is recommended to get a correct accuracy estimation.

3.5 Changes in reading statements

The family of scanf functions is adapted to classical floating-point variables, which must be transformed into stochastic variables.

Example:

Initial C statements	Modified C statements for SAM
float x; ..... scanf("x = %14.7e \n", &x);	**char xaux[100];** **mp_st<24>** x; ..... scanf("%s", **xaux**); **sam_set_str(x,xaux)**;

Note that initial data read from a file or from keyboard may have sometimes to be duplicated in some way, because they are read as classical variables which are then assigned to stochastic variables.

3.6 Changes in printing statements

The strp function enables one to print a stochastic variable if printf is used. For example, if a float variable x becomes a mp_st variable, the printing instruction can be modified as follows:

Initial C statements	Modified C statements for SAM
float x; ... printf("x = %14.7e", x);	**mp_st<24>** x; ... printf("x = %s", **strp(x)**);

3.7 Constants passed as function arguments

Function definitions and function calls must sometimes be adapted because stochastic parameters of functions must not be passed by value.

Example:

Initial C statements	Modified C statements for SAM

float a;
 	
a=3.14*f(2.0);
...
 	
float f(float x)
{
...
}
	mp_st<24> aux, a;
	aux=2.0;
	a=3.14*f(aux);
	...
	
	mp_st<24> f(**mp_st<24>**x)
	{
	...
	}

3.8 Termination of the SAM library

The call to the SAM_end function must be the last program statement.

3.9 Numerical debugging with SAM

One can enable the detection of the following instabilities:
UNSTABLE DIVISION(S),
UNSTABLE POWER FUNCTION(S),
UNSTABLE MULTIPLICATION(S),
UNSTABLE BRANCHING(S),
UNSTABLE MATHEMATICAL FUNCTION(S),
UNSTABLE INTRINSIC FUNCTION(S),
LOSS OF ACCURACY DUE TO CANCELLATION(S).

The library counts the number of detections for each instability. The global information for these detections is printed out with the SAM_end function, see 2.7.1.
The accuracy estimated by SAM is valid if there is no deep numerical anomaly during the computation, i.e. no UNSTABLE DIVISION, UNSTABLE POWER FUNCTION and UNSTABLE MULTIPLICATION, see [2, 4, 1].

The meaning of the message is:

- **unstable division:** the divisor is non-significant
- **unstable power function:** one operand of the power function is non-significant

- **unstable multiplication:** both operands are non-significant
- **unstable branching:** the difference between the two operands is non-significant (a computational zero).

  The chosen branching statement is associated with the equality.
- **unstable mathematical function:**

  in the log, sqrt, exp or log10 function, the argument is non-significant.
- **unstable intrinsic function**:
  - when using integer cast functions, the integral part of the argument can not be exactly determined due to the round-off error propagation;
  - in the fabs function: the argument is non-significant;
  - the floor, ceil or trunc function returns different values for each component of the stochastic argument.
- **loss of accuracy due to cancellation**: as explained in 2.7.1, a cancellation is pointed out when the difference between the number of exact significant digits (i.e. digits which are not affected by round-off errors) of the result of an addition or a subtraction and the minimimum of the number of exact significant digits of the two operands is greater than the cancel_level argument. The default value of this argument is 4. In other words, when one loses more than cancel_level significant digits in one addition or subtraction, SAM considers that a catastrophic cancellation has been detected (if the detection of this kind of instability is enabled).

To perform actual numerical debugging, it is necessary, for each instability, to identify the statement in the code that generates this instability. This can be performed directly using a symbolic debugger like **gdb** with Linux or as a background task using special input and output files.
In both cases, one has to put a breakpoint at the entry of the **instability** internal function of the SAM library. This function is called each time a numerical instability is detected. To get the right label for this system and compiler dependent function, one can use the following statement:

nm *name_of_the_binary_code* | grep instability

For instance, using **gdb** with Linux, the general statement which enables the detection of all the instabilities in a single run is

nohup gdb *name_of_the_binary_code* < gdb.in >! gdb.out &

The *gdb.in* file may contain:

```
break instability
run
while 1
where
cont
end
```

where prints out the complete trace of the instability which has stopped the run and **cont** makes the execution going on.
P.S.: **nohup** allows to keep the process alive even when logging off.
The *gdb.out* file will contain all the traces of instabilities.

3.10 Warning: the danger of mixing classical types and stochastic types

In this section we point out the fact that mixing classical types and stochastic types may lead to an incorrect accuracy estimation. In the following examples, the result accuracy is correctly estimated, except in 3.10.4 where an `mp_st<70>` variable is added with a `double`.

3.10.1 Adding mp_st variables initialized with sam_set_str

Adding `mp_st<70>` variables initialized with `sam_set_str`:

Let us consider the following instructions.

```
mp_st<70> x,y,z;         //precision is 70 bits, i.e. 21 decimal digits
sam_set_str(x,"1.23");
cout << "x=" <<x<< endl;
x.display();
sam_set_str(y,"4.56");
cout << "y=" <<y<< endl;
y.display();
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.12300000000000000000E+1
1.23000000000000000004 -- 1.23000000000000000004 -- 1.23000000000000000004
y= 0.45600000000000000000E+1
4.56000000000000000022 -- 4.56000000000000000022 -- 4.56000000000000000022
z= 0.57899999999999999999E+1
5.78999999999999999992 -- 5.78999999999999999992 -- 5.79000000000000000060
```

x and y are declared as mp_st<70> variables, their precision is 70 bits, *i.e.* 21 decimal digits. They are initialized using sam_set_str. One can check that their accuracy is 21 decimal digits. The three values that represent x (resp. y) are displayed. As a remark, display prints the three values that represent an mp_st variable with 2 more digits than its precision, so 23 digits here. The addition x+y is performed with the random rounding mode. The accuracy of the sum z (20 digits) is correctly estimated.

Adding an mp_st<70> variable and an mp_st<53> variable initialized with sam_set_str:

Let us consider the following instructions.

```
mp_st<70> x,z;          //precision is 70 bits, i.e. 21 decimal digits
mp_st<53> y;            //precision is 53 bits, i.e. 15 decimal digits
sam_set_str(x,"1.23");
cout << "x=" <<x<< endl;
x.display();
sam_set_str(y,"4.56");
y.data_st();
cout << "y=" <<y<< endl;
y.display();
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.12300000000000000000E+1
1.230000000000000000004 -- 1.230000000000000000004 -- 1.230000000000000000004
y= 0.45599999999999E+1
4.5600000000000005 -- 4.5599999999999987 -- 4.5599999999999996
z= 0.578999999999999E+1
5.790000000000004973769 -- 5.789999999999987210201 -- 5.789999999999996092053
```

x and z are declared as mp_st<70> variables, their precision is 70 bits, *i.e.* 21 decimal digits. y is declared as an mp_st<53> variable, its precision is 53 bits, *i.e.* 15 decimal digits. x and y are initialized using sam_set_str. y that is shorter than x is perturbed using the data_st method. The last bit of y is perturbed. The three values that represent y are slightly different. As previously mentioned, display prints the three values that represent an mp_st variable with 2 more digits than its precision, so 23 digits here. The addition x+y is performed with the random rounding mode. The accuracy of the sum z (15 digits) is correctly estimated.

3.10.2 Adding mp_st variables initialized with double values

Let us consider the following instructions.

```
mp_st<70> x,y,z;         //precision is 70 bits, i.e. 21 decimal digits
x=1.23;
cout << "x=" <<x<< endl; //accuracy of x is 15 decimal digits (see output)
x.display();
y=4.56;
cout << "y=" <<y<< endl; //accuracy of y is 15 decimal digits (see output)
y.display();
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.123000000000000E+1
1.2300000000000002553520 -- 1.229999999999998456778 -- 1.229999999999999822364
y= 0.455999999999999E+1
4.5600000000000003685970 -- 4.559999999999993560674 -- 4.559999999999996092015
z= 0.578999999999999E+1
5.7900000000000006239440 -- 5.789999999999992017418 -- 5.789999999999995914379
```

x and y are declared as mp_st<70> variables, their precision is 70 bits, *i.e.* 21 decimal digits. They are initialized using double values. One can check that their accuracy is 15 decimal digits. If an mp_st<N> variable with N$>$ 53 is set from a double value, it is perturbed and its accuracy is 15 decimal digits. The three values that represent x (resp. y) are different because of this perturbation. The accuracy of the sum z (15 digits) is correctly estimated.

3.10.3 Adding mp_st variables initialized with integers

Let us consider the following instructions.

```
mp_st<70> x,y,z;
x=123;
cout << "x=" <<x<< endl;
x.display();
y=456;
cout << "y=" <<y<< endl;
y.display();
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.12300000000000000000E+3
1.23000000000000000000e2 -- 1.23000000000000000000e2 -- 1.23000000000000000000e2
y= 0.45600000000000000000E+3
4.56000000000000000000e2 -- 4.56000000000000000000e2 -- 4.56000000000000000000e2
z= 0.57899999999999999999E+3
5.79000000000000000000e2 -- 5.79000000000000000000e2 -- 5.79000000000000000000e2
```

`x` and `y` are declared as `mp_st<70>` variables, their precision is 70 bits, *i.e.* 21 decimal digits. They are initialized using integers. One can check that their accuracy is 21 decimal digits. If an `mp_st` variable is set from an integer, it is not perturbed. The three values that represent `x` (resp. `y`) are equal. The accuracy of the sum `z` (21 digits) is correctly estimated.

3.10.4 Adding an mp_st variable and a double

Adding an `mp_st<70>` variable and a `double`:

Let us consider the following instructions.

```
mp_st<70> x,z;
sam_set_str(x,"1.23");
cout << "x=" <<x<< endl;
x.display();
double y=4.56;
cout << "y=" <<y<< endl;
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.12300000000000000000E+1
1.23000000000000000000e4 -- 1.23000000000000000000e4 -- 1.23000000000000000000e4
y=4.56
z= 0.57899999999999996092E+1
5.78999999999999961985 -- 5.78999999999999961985 -- 5.78999999999999962053
```

`x` and `z` are declared as `mp_st<70>` variables, their precision is 70 bits, *i.e.* 21 decimal digits. `x` is initialized using `sam_set_str`. One can check that its accuracy is 21 decimal digits. The sum of `x` with a `double y` is performed with the random rounding mode. If a `double` variable `y` is added with an `mp_st<N>` variable with `N`>53, the addition is performed with precision `N`. However **only the first 15 digits of `y` are correct**. The **accuracy** of the sum `z` (20 digits) is **not correctly estimated**. In the output, one can observe that the last decimal digits of `z` are not correct. `y` **should be declared as an `mp_st<53>` variable** as show in 3.10.1.

Adding an `mp_st<50>` variable and a `double`:

Let us consider the following instructions.

```
mp_st<50> x,z;
sam_set_str(x,"1.23");
cout << "x=" <<x<< endl;
x.display();
double y=4.56;
cout << "y=" <<y<< endl;
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.123000000000000E+1
1.2300000000000004 -- 1.2300000000000004 -- 1.2300000000000004
y=4.56
z= 0.57899999999999E+1
5.7899999999999991 -- 5.7899999999999991 -- 5.7900000000000063
```

`x` and `z` are declared as `mp_st<50>` variables, their precision is 50 bits, *i.e.* 15 decimal digits. `x` is initialized using `sam_set_str`. One can check that its accuracy is 15 decimal digits. The sum of `x` with a `double y` is performed. The accuracy of `y` is 15 decimal digits. The accuracy of the sum `z` is 14 digits. If a `double` variable `y` is added with an `mp_st<N>` variable with `N`≤ 53, the accuracy of the sum is correctly estimated.

3.10.5 Adding an mp_st variable and an integer

Let us consider the following instructions.

```
mp_st<70> x,z;
sam_set_str(x,"1.23");
cout << "x=" <<x<< endl;
x.display();
int y=456;
cout << "y=" <<y<< endl;
z=x+y;
cout << "z=" <<z<< endl;
z.display();
```

The associated output is.

```
x= 0.12300000000000000000E+1
1.230000000000000000004 -- 1.230000000000000000004 -- 1.230000000000000000004
y=456
z= 0.45722999999999999999E+3
4.57229999999999999958e2 -- 4.57229999999999999958e2 -- 4.57230000000000000002e2
```

`x` and `z` are declared as `mp_st<70>` variables, their precision is 70 bits, *i.e.* 21 decimal digits. `x` is initialized using `sam_set_str`. One can check that its accuracy is 21 decimal digits. The sum of `x` with an integer `y` is performed with the random rounding mode. The accuracy of the sum `z` is 20 digits. The accuracy is correctly estimated.

Chapter 4

Structure of the SAM library

The source codes of the SAM library are located in the `src` directory.

- `SAM_add.h` contains the operators related to addition: +, ++, +=
- `SAM_convert.h` contains:
  - the `data_st` method that takes into account data uncertainty at the initialization of stochastic variables.
  - conversion functions from `mp_st` objects to classical values
- `SAM_digitnumber.h` contains `nb_significant_digit` that computes the accuracy of an `mp_st` object
- `SAM_div.h` contains the operator /=
- `SAM_eq.h` contains the definition of the comparison operator ==
- `SAM_ge.h` contains the definition of the comparison operator >=
- `SAM_gt.h` contains the definition of the comparison operator >
- `SAM.h` contains the definition of the `mp_st` class
- `SAM_intr.h`: contains the definition of the following functions when called with stochastic arguments: `fabs, floor, ceil, trunc, nearbyint, rint, lrint, llrint`
- `SAM_le.h` contains the definition of the comparison operator <=
- `SAM_lt.h` contains the definition of the comparison operator <

- `SAM_math.h` contains the definition of the math functions when called with stochastic arguments.
- `SAM_MPFR_templates.h` provides simple template wrapper for common MPFR functions to make code appear more generic
- `SAM_mul.h` contains the operator *=
- `SAM_ne.h` contains the definition of the comparison operator !=
- `SAM_numericalnoise.h` contains the `numericalnoise` function that returns 1 if the argument is numerical noise, -1 if it is zero, and 0 otherwise.
- `SAM_op2.h` contains the definition of arithmetic and relational operators with at least one stochastic argument
- `SAM_perturbation.h` contains the `perturbationLastBit` function that perturbs a stochastic value when it is assigned to a longer stochastic variable.
- `SAM_private.h` contains declarations for instability detection and random number generation
- `SAM_random.cpp` contains functions for random number generation
- `SAM_str2.h` contains the strp function related to stochastic variables printing
- `SAM_str.h` is required to print stochastic variables constains the display method that prints the triplet associated with a stochastic variable and the str function
- `SAM_sub.h` contains the operators related to subtraction: -, –, -=
- `SAM_to.h`
  - contains the constructors
  - defines all the functions involving at least one argument of stochastic type which overload the assignment statement =
  - contains the `sam_set_str` method that assigns a string to an `mp_st` variable.
- `SAM_type.cpp` contains the following functions:

  - `SAM_init` that initializes the SAM library
  - `SAM_end` that "closes" the SAM library
  - `SAM_enable` that enables the detection of a kind of numerical instability
  - `SAM_disable` that disables the detection of a kind of numerical instability
  - `self_validation_only` that enables the detection of the multiplication instability, the division instability and the power instability. It disables the others.

- `SAM_unstab.cpp` contains the instability function that manages the different kinds of instabilities detected by SAM.

Chapter 5

Test runs

You first need to install GMP (GNU Multiprecision Library) and MPFR for running the tests.
We present, with the examples included in the distribution, an illustration of the use of the SAM library and the benefits of the DSA. For each example, we describe the results obtained using the standard floating-point arithmetic and then the results provided by the SAM library.
As a remark, the results may depend on the processor or the compiler, especially when the digits printed out using the standard floating-point arithmetic are affected by round-off errors. With SAM, only the exact significant digits appear in the results.

5.1 Example 1: a rational fraction function of two variables

In the following example [12], the rational fraction

$$F(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

is computed with $x = 77617$, $y = 33096$. The first 30 digits of the exact result are -0.827396059946821368141165095479.
Using IEEE double precision arithmetic with rounding to the nearest, one obtains: res = 5.764607523034235E+17.
With SAM, we obtain a satisfactory result when the working precision is greater than 121 bits. So using SAM in 122 bits, one obtains:

```
|  Polynomial function of two variables  |
|  with SAM                              |
------------------------------------------
res=-0.827396059946821368141165095479816 29
-------------------------------------------------------------------
SAM software
No instability detected
```

5.2 Example 2: solving a second order equation

The roots of the following second order equation are computed:

$$0.3x^2 - 2.1x + 3.675 = 0.$$

The exact results are: Discriminant d=0, x1=x2=3.5.
Using IEEE single precision arithmetic with rounding to the nearest, one obtains:

```
----------------------------------
|  Second order equation         |
|  without SAM                   |
----------------------------------
d = -2.861023e-06
There are two complex solutions.
z1 = +3.500000e+00  +  i * +8.457279e-04
z2 = +3.500000e+00  +  i * -8.457279e-04
```

and using SAM with 100 bits, one obtains:

```
----------------------------------
|  Second order equation         |
|  with SAM                      |
----------------------------------
d =  @.0
Discriminant is zero.
The double solution is  0.350000000000000000000000000E+1
-------------------------------------------------------------------
SAM software
There are 2 numerical instabilities
1 UNSTABLE BRANCHING(S)
1 LOSS OF ACCURACY DUE TO CANCELLATION(S)
```

The standard floating-point arithmetic cannot detect that d=0. The wrong branching is performed and the result is false.
The SAM software takes into account the accuracy of operands in the order relations or in the equality relation and, therefore, the correct branching is performed and the exact result is obtained.

5.3 Example 3: computing a determinant

The determinant of Hilbert's matrix of size 11 is computed using Gaussian elimination without pivoting strategy. The determinant is the product of the different pivots. Hilbert's matrix is defined by: $a(i,j) = 1/(i+j-1)$, $1 \leq i \leq 11$, $1 \leq j \leq 11$. All the pivots and the determinant are printed out. The first exact digits of the determinant are $3.0190953344493 * 10^{-65}$.

Using IEEE double precision arithmetic with rounding to the nearest, one obtains a determinant that has only two exact significant digits.

```
-------------------------------------------------------
|  Computation of the determinant of Hilbert's matrix  |
|  using Gaussian elimination without SAM              |
-------------------------------------------------------
Pivot number 0 = 1.000000000000000e+00
Pivot number 1 = 8.333333333333331e-02
Pivot number 2 = 5.555555555555526e-03
Pivot number 3 = 3.571428571428830e-04
Pivot number 4 = 2.267573696145566e-05
Pivot number 5 = 1.431549050529594e-06
Pivot number 6 = 9.009749264103679e-08
Pivot number 7 = 5.659971084095516e-09
Pivot number 8 = 3.551369635569034e-10
Pivot number 9 = 2.226762517485834e-11
Pivot number 10 = 1.399228241996033e-12
Determinant     = 3.028594438809703e-65
```

and using SAM with 100 bits, one obtains:

```
-------------------------------------------------------
|  Computation of the determinant of Hilbert's matrix  |
|  using Gaussian elimination with SAM                 |
-------------------------------------------------------
Pivot number 0 =  0.10000000000000000000000000000E+1
```

```
Pivot number 1 =  0.83333333333333333333333333333E-1
Pivot number 2 =  0.55555555555555555555555555556E-2
Pivot number 3 =  0.35714285714285714285714285714E-3
Pivot number 4 =  0.22675736961451247165532879819E-4
Pivot number 5 =  0.14315490505966696442887E-5
Pivot number 6 =  0.90097492694895292297900E-7
Pivot number 7 =  0.56599706949357299008E-8
Pivot number 8 =  0.35513541615283011400E-9
Pivot number 9 =  0.22264681662832228E-10
Pivot number 10 =  0.139503017937545E-11
Determinant      =  0.301909533444935E-64
-----------------------------------------------------------------
SAM software
No instability detected
```

5.4 Example 4: computing a second order recurrent sequence

This sequence was proposed by J.-M. Muller [10]. The first 30 iterations of the following recurrent sequence are computed:

$$U_{n+1} = 111 - \frac{1130}{U_n} + \frac{3000}{U_n U_{n-1}}$$

with $U_0 = 5.5$ and $U_1 = \frac{61}{11}$. The exact limit is 6.
Using IEEE double precision arithmetic with rounding to the nearest, one obtains:

```
-------------------------------------
| A second order recurrent sequence |
| without SAM                       |
-------------------------------------
U(3) = +5.590163934426237e+00
U(4) = +5.633431085044127e+00
U(5) = +5.674648620512615e+00
U(6) = +5.713329052423919e+00
U(7) = +5.749120920462043e+00
U(8) = +5.781810933690098e+00
U(9) = +5.811314466602178e+00
U(10) = +5.837660476543959e+00
```

```
U(11) = +5.861018785996283e+00
U(12) = +5.882524608269310e+00
U(13) = +5.918655323805488e+00
U(14) = +6.243961815306110e+00
U(15) = +1.120308737284091e+01
U(16) = +5.302171264499677e+01
U(17) = +9.473842279276452e+01
U(18) = +9.966965087355071e+01
U(19) = +9.998025776093678e+01
U(20) = +9.999882245337588e+01
U(21) = +9.999992970745579e+01
U(22) = +9.999999580049865e+01
U(23) = +9.999999974893262e+01
U(24) = +9.999999998498109e+01
U(25) = +9.999999999910112e+01
U(26) = +9.999999999994618e+01
U(27) = +9.999999999999677e+01
U(28) = +9.999999999999980e+01
U(29) = +9.999999999999999e+01
U(30) = +1.000000000000000e+02
The exact limit is 6.
```

and using SAM in double precision (53 bits), one obtains:

```
------------------------------------
| A second order recurrent sequence |
| with SAM                          |
------------------------------------
U(3) =  0.5590163934426E+1
U(4) =  0.563343108504E+1
U(5) =  0.56746486205E+1
U(6) =  0.571332905E+1
U(7) =  0.57491209E+1
U(8) =  0.5781811E+1
U(9) =  0.581131E+1
U(10) =  0.58376E+1
U(11) =  0.586E+1
U(12) =  0.59E+1
U(13) =  0.6E+1
U(14) =  @.0
U(15) =  @.0
```

```
U(16) =  @.0
U(17) =  @.0
U(18) =  0.9E+2
U(19) =  0.99E+2
U(20) =  0.999E+2
U(21) =  0.99999E+2
U(22) =  0.999999E+2
U(23) =  0.9999999E+2
U(24) =  0.99999999E+2
U(25) =  0.999999999E+2
U(26) =  0.99999999999E+2
U(27) =  0.999999999999E+2
U(28) =  0.9999999999999E+2
U(29) =  0.99999999999999E+2
U(30) =  0.100000000000000E+3
The exact limit is 6.
----------------------------------------------------------------
SAM software
CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed.
There are 12 numerical instabilities
9 UNSTABLE DIVISION(S)
3 UNSTABLE MULTIPLICATION(S)
```

if 40 iterations are performed using SAM with 100 bits , one obtains:

```
-------------------------------------
| A second order recurrent sequence |
| with SAM                          |
-------------------------------------
U(3)= 0.559016393442622950819672131E+1
U(4)= 0.563343108504398826979472l4E+1
U(5)= 0.567464862051015096304008 3E+1
U(6)= 0.571332905238051554903219E+1
U(7)= 0.574912091970263804 3705E+1
U(8)= 0.578181092048561557947E+1
U(9)= 0.581131423829399572 32E+1
U(10)= 0.583765654895871196 2E+1
U(11)= 0.58609515225161319E+1
U(12)= 0.5881377215841419E+1
U(13)= 0.589915390579007E+1
```

```
U(14)= 0.59145249506789E+1
U(15)= 0.592774140778E+1
U(16)= 0.59390504855E+1
U(17)= 0.5948687492E+1
U(18)= 0.595687073E+1
U(19)= 0.5963799E+1
U(20)= 0.596965E+1
U(21)= 0.59746E+1
U(22)= 0.5979E+1
U(23)= 0.598E+1
U(24)= 0.6E+1
U(25)= @.0
U(26)= @.0
U(27)= @.0
U(28)= @.0
U(29)= @.0
U(30)= 0.10E+3
U(31)= 0.100E+3
U(32)= 0.1000E+3
U(33)= 0.10000E+3
U(34)= 0.1000000E+3
U(35)= 0.10000000E+3
U(36)= 0.100000000E+3
U(37)= 0.1000000000E+3
U(38)= 0.100000000000E+3
U(39)= 0.1000000000000E+3
U(40)= 0.10000000000000E+3
The exact limit is 6.
------------------------------------------------------------
SAM software

CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed.

There are 15 numerical instabilities
11 UNSTABLE DIVISION(S)
4 UNSTABLE MULTIPLICATION(S)
```

The warnings UNSTABLE DIVISION(S) are generated by divisions where the denominator is a computational zero. Such operations make the com-

puted trajectory turn off the exact trajectory and then, the estimation of accuracy is not possible any more. Even using the multiple precision (100 bits), the computer cannot give any significant result after the iteration 24.

5.5 Example 5: computing a polynomial root

This example deals with the improvement and optimization of an iterative algorithm by using SAM features. This program computes a root of the polynomial

$$f(x) = 1.47x^3 + 1.19x^2 - 1.83x + 0.45$$

by Newton's method. The sequence is initialized with $x = 0.5$. The exact value of the root is $3/7 = 0.428571428571428571...$

First we use IEEE double precision arithmetic with rounding to nearest and we stop the iterative algorithm $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ with the criterion $|x_n - x_{n-1}| \leq 10^{-12}$. The associated instructions are (for clarity, print instructions are not given):

```
int i,nmax=100;
double y, x, eps=1.e-12;
y = 0.5;
for(i = 1;i<=nmax;i++){
   x = y;
   y = x-(1.47*x*x*x+1.19*x*x-1.83*x+0.45) / (4.41*x*x+2.38*x-1.83);
   if (fabs(x-y)<=eps) break;
}
```

We obtain:

```
-------------------------------------------------------------
| Computation of a root of a polynomial by Newton's method  |
| without SAM                                               |
-------------------------------------------------------------
x( 33) = +4.285714285823216e-01
x( 34) = +4.285714285823216e-01
```

Only the first 10 digits are correct.

Using SAM, the stopping criterion can be changed. Iterations are stopped when the difference between two successive iterates is a computational zero

(there is no more useful information to compute at the next iteration). SAM is used with a working precision of 100 bits (30 decimal digits). The polynomial coefficients are initialized using the `sam_set_str` function. The associated instructions are (for clarity, some print instructions are not given):

```
SAM_init(-1);
mp_st<100> y, x, t[6], half, exact_root;
int i, nmax=100;
sam_set_str(half,"0.5");
sam_set_str(t[0],"1.47");
sam_set_str(t[1],"1.19");
sam_set_str(t[2],"-1.83");
sam_set_str(t[3],"0.45");
sam_set_str(t[4],"4.41");
sam_set_str(t[5],"2.38");
//if an integer is assigned to an mp_st, no perturbation is performed.
exact_root=3;
exact_root/=7;
y = half;
for(i = 1;i<=nmax;i++){
   x = y;
   y=x-((t[0]*x*x*x+t[1]*x*x+t[2]*x+t[3])/(t[4]*x*x+t[5]*x+t[2]));
   if (x==y) break;
}
printf("x(%3d) = %s\n",i-1,strp(x));
printf("x(%3d) = %s\n",i,strp(y));
printf("exact root = %s\n",strp(exact_root));
SAM_end();
```

One obtains:

```
------------------------------------------------------------
|  Computation of a root of a polynomial by Newton's method |
|  with SAM                                                 |
------------------------------------------------------------
x( 46)     =  0.428571428571429
x( 47)     =  0.42857142857143
exact root =  0.428571428571428571428571428571
------------------------------------------------------------------
SAM software
```

```
CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed.

There are 83 numerical instabilities
1 UNSTABLE DIVISION(S)
1 UNSTABLE BRANCHING(S)
81 LOSS OF ACCURACY DUE TO CANCELLATION(S)
```

All the digits in the result obtained are correct. As a remark, in Newton's method, a division by a computational zero may suggest a double root.

Then we use the following instructions (for clarity, some print instructions are not given):

```
SAM_init(-1);
mp_st<100> y, x, exact_root;
int i, nmax=100;
exact_root=3;
exact_root/=7;
y = 0.5;
for(i = 1;i<=nmax;i++){
   x = y;
   y = x-(1.47*x*x*x+1.19*x*x-1.83*x+0.45) / (4.41*x*x+2.38*x-1.83);
   if (x==y) break;
}
printf("x(%3d)     = %s\n",i-1,strp(x));
printf("x(%3d)     = %s\n",i,strp(y));
printf("exact root = %s\n",strp(exact_root));
SAM_end();
```

The polynomial coefficients are double values. Only their 15 first digits are correct. In the following results, only the 8 first digits are correct.

```
-------------------------------------------------------------
|  Computation of a root of a polynomial by Newton's method |
|  with SAM                                                 |
-------------------------------------------------------------
x( 29)     =  0.428571431755965987340
x( 30)     =  0.428571431755965987340
exact root =  0.428571428571428571428571428571
-------------------------------------------------------------
```

```
SAM software
There are 47 numerical instabilities
1 UNSTABLE BRANCHING(S)
46 LOSS OF ACCURACY DUE TO CANCELLATION(S)
```

5.6 Example 6: solving a linear system

In this example, SAM is able to provide correct results which were impossible to be obtained with the standard floating-point arithmetic. The following linear system $Ax = b$ is solved using Gaussian elimination with partial pivoting.

$$\begin{pmatrix} 21 & 130 & 0 & 2.1 \\ 13 & 80 & 4.74\ 10^{8} & 752 \\ 0 & -0.4 & 3.9816\ 10^{8} & 4.2 \\ 0 & 0 & 1.7 & 9\ 10^{-9} \end{pmatrix} . x = \begin{pmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6\ 10^{-8} \end{pmatrix}$$

The exact solution is $x^t_{sol} = \left(1, 1, 10^{-8}, 1\right)$. Using IEEE single precision arithmetic with rounding to the nearest, one obtains:

```
-------------------------------------------------------
| Solving a linear system using Gaussian elimination |
| by partial pivoting without SAM                    |
-------------------------------------------------------
x_sol(0) = +6.261988e+01 (exact solution: xsol(0)= +1.000000e+00)
x_sol(1) = -8.953979e+00 (exact solution: xsol(1)= +1.000000e+00)
x_sol(2) = +0.000000e+00 (exact solution: xsol(2)= +1.000000e-08)
x_sol(3) = +1.000000e+00 (exact solution: xsol(3)= +1.000000e+00)
```

and using SAM with 53 bits, one obtains:

```
-------------------------------------------------------
| Solving a linear system using Gaussian elimination |
| with partial pivoting                              |
-------------------------------------------------------
x_sol(0) = 0.10000000000E+1 (exact solution: xsol(0)= 0.100000000000000E+1)
x_sol(1) = 0.999999999999 (exact solution: xsol(1)= 0.100000000000000E+1)
x_sol(2) = 0.100000000000000E-7 (exact solution: xsol(2)= 0.100000000000000E-7)
x_sol(3) = 0.999999999999999 (exact solution: xsol(3)= 0.100000000000000E+1)
-------------------------------------------------------------
SAM software
```

```
There is  1 numerical instability
1 LOSS OF ACCURACY DUE TO CANCELLATION(S)
```

Using standard floating-point arithmetic, during the reduction of the third column, the matrix element $A(3,3)$ is equal to 4864. But the exact value of $A(3,3)$ is zero. The standard floating-point arithmetic cannot detect that a(3,3) is insignificant. This value is chosen as pivot. That leads to erroneous results. SAM detects the non-significant value of $A(3,3)$. This value is eliminated as pivot. That leads to satisfactory results.

5.7 Example 7: when SAM fails

SAM is based on a probabilistic model. It should never be forgotten that all the estimations computed by SAM are probabilistic, even if the probability is close to 1. Moreover, the theoretical model shows that SAM is able to estimate the round-off errors to the first order. If they represent the global round-off errors, SAM works well but, if they are dominated by terms of greater order, SAM may fail.
In the present example, we have the same behaviour but only with additions and subtractions, so without any warning of numerical instability. Let us perform the following computation:

```
x=6.83561e+5;
y=6.83560e+5;
z=1.00000000007;
r = z - x;
r1 = z - y;
r = r + y;
r1 = r1 + x;
r1 = r1 - 2;
r = r + r1;
//      r = ((z-x)+y) + ((z-y)+x-2)
```

The exact result is $1.4\,10^{-10}$. The result obtained using IEEE double precision arithmetic with rounding to the nearest is 2.32830643653870E-10.

With SAM in double precision (53 bits), because we essentially perform the same computation, $((z-x)+y)$ and $((z-y)+x-2)$, we find that if the same rounding mode is chosen for both parts, the final result appears as exact but it is wrong. It happens in one case out of four and the result provided by SAM is then 0.116415321826935E-009 with 15 exact significant

digits. If computations are performed 100000 times using SAM, one may obtain:

```
Example created on purpose to make CADNA fail
The same result r is computed for a number of iterations
chosen by the user.
The exact result is 1.4E-10.
But in 1 case out of 4, SAM estimates an incorrect accuracy.
Enter the number of iterations: 1000
Last value of r:  @.0
Number of iterations when CADNA estimates an incorrect accuracy: 254
----------------------------------------------------------------
SAM software
There are 3746 numerical instabilities
746 UNSTABLE BRANCHING(S)
3000 LOSS OF ACCURACY DUE TO CANCELLATION(S)
```

The last value of `r` is printed out, and also the number of times when the result was wrong.

Bibliography

[1] J. Vignes, Discrete Stochastic Arithmetic for Validating Results of Numerical Software, *Special Issue of Numerical Algorithms*, 2004, 37, pp. 377-390

[2] J.-M. Chesneaux, Étude théorique et implémentation en ADA de la méthode CESTAC, *Thèse de l'université P. et M. Curie*, Paris, 1988.

[3] J.-M. Chesneaux, Modélisation théorique et conditions de validité de la méthode CESTAC, C.R.A.S., Paris, série 1, tome 307, 19881 pp. 417-422.

[4] J.-M. Chesneaux, L'arithmétique stochastique et le logiciel CADNA, Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, 1995.

[5] J.-M. Chesneaux, The equality relations in scientific computing, *Num. Algo 7*, 1994, pp. 129-143.

[6] J. Vignes, Error analysis in computing. International Federation for Information Processing Congress, Stockholm, Aug. 1974, pp. 610-614.

[7] J. Vignes, New Methods for evaluating the validity of the results of mathematical computations. Math. and Comp. in Sim., 20, 1978, pp. 227-249.

[8] J. Vignes, Estimation de la précision des résultats de logiciels numériques. *La Vie des Sciences, Comptes Rendus, série générale*, 7, 1990, pp. 93-143.

[9] J. Vignes, A stochastic arithmetic for reliable scientific computation, *Math. and Comp. in Sim.* 35, 1993, pp. 233-261.

[10] J.-M. Muller, Arithmétique des ordinateurs, Masson, 1989.

[11] J.-M. Chesneaux, J. Vignes, Sur la robustesse de la méthode CESTAC, *C.R. Acad. Sc.* Paris, Sér. I Math. 307, 1988, pp. 855-860.

[12] S.M. Rump, Algorithms for Verified Inclusions - Theory and Practice. In R.E. Moore, editor, Reliability in Computing, volume 19 of Perspectives in Computing, pages 109-126. Academic Press, 1988.

[13] J. Vignes, Zéro mathématique et zéro informatique. *La Vie des Sciences, C.R. Acad. Sci.*, Paris, 4, 1, janvier 1987, pp. 1-13.

[14] J. Vignes, M. La Porte, Error analysis in computing, *Information Processing 74*, North-Holland, 1974.

[15] J. Vignes, A Stochastic Approach to the Analysis of Round-off Error Propagation. A Survey of the CESTAC Method. Proceedings of Real Numbers and Computer Conference. Marseille, 1996 pp. 233-251.

[16] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August, 1985, reprinted in SIGPLAN 22, 2, pp. 9-25.