Numerical validation using the CADNA library practical work

Stef Graillat, Fabienne Jézéquel, Jean-Luc Lamotte
LIP6 Laboratory, P. and M. Curie University,
Paris, France
{Stef.Graillat, Fabienne.Jezequel, Jean-Luc.Lamotte}@lip6.fr

July 12, 2013

Contents

Chapter 1

Getting started

Step 1

If you use a pre-installed virtual machine, directly go to Step 2, otherwise:

Download the CADNA library.
Compile and install the library on your home directory.
The installation procedure may be for instance:

```
gunzip cadna_c-1.1.8beta.tar.gz
tar -xvf  cadna_c-1.1.8beta.tar
cd   cadna_c-1.1.8beta
./configure --prefix=‘pwd‘ --enable-64bit
make
make install
```

Step 2

Copy the exercises.tar.gz archive to your home directory, then:

```
gunzip exercises.tar.gz
tar -xvf  exercises.tar
cd  exercises
```

Chapter 2

Exercises

2.1 Exercise 1

This example has been proposed by S. Rump.

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

is computed with $x = 77617$, $y = 33096$. The 15 first digits of the exact result are -0.827396059946821.

2.1.1 Question 1

Compile and run the `rump.c` C program:

```
make rump
./rump
```

Copy it into the `rumpd.c` program.
Change the type variable type from float to double in the `rumpd.c` program.
Compile and run the `rumpd.c` C program:

```
make rumpd
./rumpd
```

2.1.2 Question 2

Implement the CADNA library in the `rump.c` and `rumpd.c` programs by creating two new programs called `rump_cad.cc` and `rumpd_cad.cc`.
Run the `rump_cad.cc` and the `rumpd_cad.cc` program.

2.2 Exercise 2

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: $a(i,j) = 1/(i+j-1)$. All the pivots and the determinant are printed out.

2.2.1 Question 1

Run the `hilbert.c` C program.
What do you think about the determinant value?
The exact value of the determinant is $3.0190953344493\,10^{-65}$.

2.2.2 Question 2

Implement the CADNA library in the `hilbert.c` program by creating a new program called `hilbert_cad.cc`. Run the `hilbert_cad.cc` program.

2.3 Exercise 3

This example has been proposed by J.-M. Muller.
The `muller.c` program computes the first 25 iterations of the following sequence:

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

with $U_0 = 5.5$ and $U_1 = \frac{61}{11}$.

2.3.1 Question 1

Run the `muller.c` C program.
What is the sequence limit?

2.3.2 Question 2

Implement the CADNA library in the `muller.c` program by creating a new program called `muller_cad.cc`.
Run the program.
Is the result correct? Why?

Launch the program with the `gdb` debugger:

```
gdb muller_cad
```

Set a break on the instability function (using the `break instability` command with `gdb`).
Run the program (using the `run` command with `gdb`).
What happens?
Use twice the `up` command with `gdb`.
Give the value of `i, a` and `b` (using the `print` command with `gdb`).
Continue the run (using the `cont` command with `gdb`).
To quit the debugger, use the `quit` command.

2.4 Exercise 4

This example deals with the improvement and optimization of an iterative algorithm by using new tools which are contained in CADNA. 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 iterative algorithm $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ is stopped with the criterion

$$|x_n - x_{n-1}| < 10^{-12}.$$

2.4.1 Question 1

Run the `newton.c` program.
What is the sequence limit?
How many iterations are computed?

2.4.2 Question 2

Implement the CADNA library in the `newton.c` program by creating a new program called `newton1_cad.cc`.
Run the program. How many iterations are computed?
Look at the CADNA report.

With the debugger, find the unstable division.
Why does an unstable division appear?
Change the stopping criteria to `x==y`. How many iterations are computed?

2.5 Exercise 5

The following linear system is solved using Gaussian elimination with partial pivoting. The system is

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

The exact solution is $x_{sol}^{t} \;=\; \left(1, 1, 10^{-8}, 1\right)$.

2.5.1 Question 1

Run the `gauss.c` program.
Are the results correct?

2.5.2 Question 2

Implement the CADNA library in the `gauss.c` program by creating a new program called `gauss_cad.cc`.
Run the program.
With the debugger, find where the instabilities appear.
Print the values `a[2][2]` and `a[3][2]` in the `gauss_cad.cc` program.
Do the same in the `gauss.c` program.

2.6 Exercise 6

The `jacobi.c` program implements Jacobi iteration to find the solution of a linear system.

2.6.1 Question 1

Run the `jacobi.c` program.
Are the results correct?
How many iterations are performed?
Try other values for ε.

2.6.2 Question 2

Implement the CADNA library in the `jacobi.c` program by creating a new program called `jacobi_cad.cc`.
Run the program.
Change the stopping criterion.

2.7 Exercise 7

Let us consider the logistic iteration defined by $x_{n+1} = ax_n(1 - x_n)$ with $a > 0$ and $0 < x_0 < 1$.
As a remark, a mathematically equivalent sequence is: $x_{n+1} = -a(x_n - \frac{1}{2})^2 + \frac{a}{4}$.

2.7.1 Question 1

Run the `logistic.c` program which computes the logistic iteration.
In the program both sequence expressions are given.

2.7.2 Question 2

Implement the CADNA library in the `logistic.c` program by creating a new program called `logistic_cad.cc`.
Run the program.
Change the stopping criterion in order to avoid useless iterations.

Chapter 3

User's guide

The use of the CADNA library involves seven steps:

- declaration of the CADNA library for the compiler,
- initialization of the CADNA library,
- substitution of the type float or double by stochastic types 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 CADNA functions to evaluate the number of exact significant digits,
- termination of the CADNA library.

The reader may refer to the sample program given in 3.8 with two versions, *i.e.* the initial C code and the code modified to be compiled with the CADNA library.

3.1 Declaration of the CADNA library

The following pseudo-statement

```
#include <cadna.h>
```

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

3.2 Initialization and termination of the CADNA library

The call to the cadna_init function must be added just after the **main program declaration statements** to initialize random arithmetic.
The call to the cadna_end function must be the last executed program statement.

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 associated stochastic type.

Example:

standard declarations	CADNA declarations
float a, b;	**float_st a, b;**
double c;	**double_st c;**
float d[6], e, f;	**float_st d[6], e, f;**

3.4 Changes in assignments or arithmetic operations

3.4.1 Conversions between usual types and stochastic types

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

With the following declarations:
float_st a, b;
float r;
the assignments a = r;, b = 2; and r = a; are correct but there is, of course, no information from the accuracy point of view for r.

When a variable is set to a value which can not be exactly coded on computer, the data_st method should be used.

Example:

Initial C statements	Modified C statements for CADNA
	#include <cadna.h>
float x, y;	**float_st x, y;**
x=1.234;	x=1.234;
	x.data_st();
y=-3.0;	y=-3.0;

3.4.2 Classical arithmetic operators

As previously described, all arithmetic operators on floating-point variables are overloaded and arithmetic expressions without functions do not have to be modified. Expressions may contain a mixture of stochastic types, classical types and integer types.

With the following declarations:
float_st a, b;
double_st c;
the statement c = a * a + b * 3; needs no change.

The result of expressions containing stochastic terms will be of stochastic type. As for classical types, double_st prevails over float_st.
So with the previous declarations, c = a * c + b * 3 needs no change.

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 CADNA
	#include <cadna.h>
float x;	**float xaux;**
	float_st x;
.....	.....
scanf("x = %14.7e \n" , &x);	scanf("x = %14.7e \n" , &**xaux**);
	x = xaux;

3.6 Changes in printing statements

Before printing each stochastic variable, it must be transformed into a string by the str or strp function. The required length is 15 for a float_st variable and 25 for a double_st variable. Therefore formats should be modified.

For example, if a float variable x becomes a float_st variable, the printing instruction can be modified as follows:

Initial C statements	Modified C statements for CADNA
	#include <cadna.h>
float x;	**float_st** x;
...	...
printf("x = %14.7e n" , x);	printf("x = %s n" , **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 CADNA
	#include <cadna.h>
float a;	**float_st aux, a;**
	aux=2.0;
a=3.14*f(2.0);	**a=3.14*f(aux);**
...	...
float f(x)	**float_st** f(x)
{	{
float x;	**float_st** x;
...	...
}	}

3.8 A example of numerical code and its modified version

The following source codes use the Gauss-Jordan method to invert a matrix.

3.8.1 Standard C source code

```
#include <stdio.h>
#define N 4
```

```
// Initialization:
void InitMat(float M[N][N])
{int i,j;
 for(i=0;i<N;i++)
   for(j=0;j<N;j++) scanf("%e",&M[i][j]);
}

// Inversion using the Gauss-Jordan method:
void InvertMat(float M[N][N])
{int i,j,k;
 float temp;
 for(k=0;k<N;k++)
   {temp = M[k][k];
    M[k][k] = 1.0;
    for(j=0;j<N;j++) M[k][j]/=temp;
    for(i=0;i<N;i++)
      if(i!=k)
        {temp=M[i][k];
         M[i][k] = 0.0;
         for(j=0;j<N;j++) M[i][j] -= temp*M[k][j];
        }
   }
}

// Display of a matrix:
void DisplayMat(float M[N][N])
{int i,j;
 for(i=0;i<N;i++)
   {for(j=0;j<N;j++)
      printf("%14.7e  ",M[i][j]);
    printf("\n");
   }
}

void main()
{float M[N][N];
 printf("Initial matrix:\n");
 InitMat(M);
 DisplayMat(M);
 InvertMat(M);
 printf("Inverted matrix:\n");
 DisplayMat(M);
}
```

3.8.2 Source code using the CADNA library

```
#include <cadna.h>
#include <stdio.h>
#define N 4

// Initialization:
void InitMat(float_st M[N][N])
{float aux;
 int i,j;
 for(i=0;i<N;i++)
     for(j=0;j<N;j++) {scanf("%e",&aux); M[i][j] = aux;}
}

// Inversion using the Gauss-Jordan method:
void InvertMat(float_st M[N][N])
{int i,j,k;
 float_st temp;
 for(k=0;k<N;k++)
   {temp = M[k][k];
    M[k][k] = 1.0;
    for(j=0;j<N;j++) M[k][j]/=temp;
    for(i=0;i<N;i++)
      if(i!=k)
        {temp=M[i][k];
         M[i][k]=0.0;
         for(j=0;j<N;j++) M[i][j] = M[i][j] - temp*M[k][j];
        }
   }
}

// Display of a matrix:
void DisplayMat(float_st M[N][N])
{int i,j;
 for(i=0;i<N;i++)
   {for(j=0;j<N;j++)
      printf("%s  ",strp(M[i][j]));
    printf("\n");
   }
}

void main()
{cadna_init(-1);
 float_st M[N][N];
 printf("Initial matrix:\n");
 InitMat(M);
 DisplayMat(M);
 InvertMat(M);
 printf("Inverted matrix:\n");
 DisplayMat(M);
 cadna_end();
}
```

3.8.3 Example of execution without CADNA

```
 Initial matrix:
 1.0000000e+00   2.0000000e+03   5.0000000e-01   4.0000000e+00
 2.9999999e-05   1.0000000e+00   2.0000000e+00   8.0000000e+00
 4.0000000e+00   5.0000000e-01   2.9999999e-08   2.0000000e+00
 2.0000000e+00   3.0000000e+00   5.0000000e-01   5.0000000e+09
Inverted matrix:
-6.2576764e-05  -8.1558341e-05   2.5001565e-01  -9.9964599e-11
 5.0009380e-04  -1.2504448e-04  -1.2502252e-04  -1.4995290e-13
-2.5004597e-04   5.0006253e-01   5.8761423e-05  -7.9992352e-10
-2.4999515e-13  -4.9991469e-11  -9.9937121e-11   2.0000000e-10
```

3.8.4 Example of execution with CADNA

```
 Initial matrix:
 0.1000000E+01   0.1999999E+04   0.5000000E+00   0.4000000E+01
 0.2999999E-04   0.1000000E+01   0.2000000E+01   0.8000000E+01
 0.4000000E+01   0.5000000E+00   0.2999999E-07   0.2000000E+01
 0.2000000E+01   0.3000000E+01   0.5000000E+00   0.5000000E+10
Inverted matrix:
-0.62E-04   @.0             0.250015E+00  -0.10E-09
 0.5000938E-03  -0.124E-03  -0.1250225E-03  -0.14E-12
-0.250045E-03   0.500062E+00   0.5876140E-04  -0.799923E-09
-0.250E-12  -0.49E-10  -0.999371E-10   0.2000000E-09
```

3.9 Numerical debugging with CADNA

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 cadna_end function.
The accuracy estimated by CADNA is valid if there is no deep numerical anomaly during the computation, i.e. no UNSTABLE DIVISION, UNSTABLE POWER FUNCTION and UNSTABLE MULTIPLICATION.
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 ??, 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, CADNA 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 CADNA 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.