MuPAT

User's Guide

Tsubasa Saito, Satoko Kikkawa

March 9, 2012

Contents

1 Installation

We show how to use MuPAT(Multiple Precision Arithmetic Toolbox) on Scilab. MuPAT has two main implementations: *MuPAT* is the natural toolbox which implemented only Scilab function, and the others, *MuPAT_win* and *MuPAT_maclin* are the high-speed implementation which calls functions written in C language.

1.1 System requirements

The supported Scilab Versions are 5 or more later. See below for further details of each implementation.

1.1.1 MuPAT

MuPAT is independent of any hardware and operating systems (OS). In other words, *MuPAT* requires only Scilab, because *MuPAT* is implemented using pure Scilab functions.

1.1.2 MuPAT_win

MuPAT_win is a high-speed implementation which calls a function defined in a Dynamic Link Library (DLL) written in C++. It runs only on a Windows OS. We checked that *MuPAT_win* can run on Windows XP, Vista and 7. We create DLLs with Microsoft Visual C++ Express Edition 2010 on Windows Vista.

1.1.3 MuPAT_maclin

MuPAT_maclin is also high-speed implementations. *MuPAT_maclin* runs on a Mac OS and a Linux OS. We checked that they run on Mac OS X Lion and Ubuntu 11.10. They call functions written in C. They depend on the C compiler. Scilab recommends GNU compiler collection for Mac OS X and GNU/Linux. Please see Scilab Online Help (Supported and compatible compilers, http://help.scilab.org/docs/5.3.3/en_US/supported_compilers.html) for details.

1.2 Load into Scilab

The loading procedure and the usage are the same among the above implementations. Hereinafter, we call these implementations collectively 'MuPAT'. To load MuPAT, we execute the following procedure.

1. Launch Scilab.
2. Enter `editor` into Scilab console and launch SciNotes.
3. Select 'Open' and open up a file builder.sce.
4. Load builder.sce into Scilab, then loader.sce is generated.
5. Select 'Open' and open up a file loader.sce.

6. Load loader.sce into Scilab.

Please see the appendix to check the folder structure of each implementation. Scilab is ready to use MuPAT if Scilab prints the following result when we enter `dd(1)`.

```
-->dd(1)
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    1.
       ans(3)
    0.
```

2 Usage

In MuPAT, we can use quadruple and octuple precision arithmetics almost the same way as double precision arithmetic, because Scilab can define new data types and apply overloading for operators and some Scilab functions.

2.1 Variable definition

In Scilab, double precision numbers are defined by the data type named *constant*. Scalars, vectors and matrices are treated in the same way as *constant*. In MuPAT, we define new data types named *dd* and *qd* to contain double-double and quad-double numbers [3, 5]. We can treat scalars, vectors and matrices of type *dd* and *qd* the same as *constant*. The following functions create a variable of *dd* or *qd*.

- `dd(a0,a1)`
  Defines a new variable of type *dd*. It needs one or two arguments. If we enter one argument, it assigns 0 (or zero vector or matrix) for `a1` automatically.
- `qd(a0,a1,a2,a3)`
  Defines a new variable of type *qd*. It needs from one to four arguments. If we enter from one to three arguments, it assigns 0 (or zero vector or matrix) for the other lower arguments automatically.

The *dd* (*qd*) variable has two (four) components at the maximum. Components of *dd* (`a0` and `a1`) and *qd* (`a0`, `a1`, `a2` and `a3`) are substituted in descending (renormalized) order.

Ex.1
When we enter `a = dd(1,0)`, the *dd* valuable `a` becomes `1`. We can also enter `a = dd(1)` to generate the same variable `a`. We can define a variable of type *qd* similarly. We enter `b = qd(1,0,0,0)`,

then the *qd* valuable `b` becomes `1`. We can also enter `b = qd(1)` to generate the same variable `b`.

```
-->a = dd(1,0)
 a  =
       a(1)
!dd  hi  lo  !
       a(2)
    1.
       a(3)
    0.
-->b = qd(1,0,0,0)
 b  =
       b(1)
!qd  d  !
       b(2)
    1.
       b(3)
    0.
       b(4)
    0.
       b(5)
    0.
-->A = [1,2,3;4,5,6];
-->AA = dd(A)
 AA  =
       AA(1)
!dd  hi  lo  !
       AA(2)
    1.    2.    3.
    4.    5.    6.
       AA(3)
    0.    0.    0.
    0.    0.    0.
```

2.1.1 Element insertion and extraction

Scilab enable us to insert or extract a partial variable easily, because we can use the symbol : (see details http://help.scilab.org/docs/5.3.3/en_US/extraction.html). MuPAT also enable us to insert or extract a partial variable of types *dd* and *qd*. We can use the following code:

- `A(i,j)=b   //insertion`

Inserts *dd* (or *qd*) variable `b` into the (i, j) element of the *dd* (or *qd*) variable `A`.

- `A(i,j)   //extraction`

  Extracts the (i, j) element of the *dd* (or *qd*) variable `A`.

```
-->A = [1,2,3;4,5,6];
-->AA = dd(A);
-->AA(2,3)
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    6.
       ans(3)
    0.
-->AA(1,:)
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    1.    2.    3.
       ans(3)
    0.    0.    0.
-->AA(2,2) = 10  //element insertion
 AA  =
       AA(1)
!dd  hi  lo  !
       AA(2)
    1.    2.     3.
    4.    10.    6.
       AA(3)
    0.    0.    0.
    0.    0.    0.
```

2.2 Input and Output

MuPAT has the following I/O functions. They support only a scalar variable. (i.e. Only 1 by 1 is allowed for the size of argument.)

- `ddprint(a)`

  Shows the variable of *dd* that has 31 significant digits in decimal.

· `qdprint(a)`

Shows the variable of *qd* that has 63 significant digits in decimal.

· `ddinput(s)`

Converts a string `s` into a variable of type *dd*. This function can read both the fixed-point number representation and the floating-point number representation.

· `qdinput(s)`

Convert a string `s` into a variable of type *qd*. This function can read both the fixed-point number representation and the floating-point number representation.

2.3 Type conversion

MuPAT has the following functions to convert data types.

· `d2dd(a)`

Convert a variable of type *costant* `a` into *dd*.

· `d2qd(a)`

Convert a variable of type *costant* `a` into *qd*.

· `dd2qd(a)`

Convert a variable of type *dd* `a` into *qd*.

· `qd2dd(a)`

Convert a variable of type *qd* `a` into *dd*.

· `getHi(a)`

Convert a variable of type *dd* or *qd* `a` into *constant*.

We show details of conversion below.

2.3.1 Convert constant into dd or qd

To convert the *constant* variable `a` into *dd* (*qd*), we use `d2dd(a)` (`d2qd(a)`). Or, we can also enter `dd(a)` (`qd(a)`) to convert data types of higher precision. Of course it is the same way even if `a` is a vector or matrix.

Ex.

We show how to convert each data type. We set a *constant* value `a=1`, and convert `a` into *dd* variable `b` and *qd* variable `c`.

```
-->a=1
 a  =
    1.
-->b=d2dd(a) // or b=dd(a)
 b  =
       b(1)
```

```
!dd  hi  lo  !
       b(2)
    1.
       b(3)
    0.
-->c=d2qd(a) // or c=qd(a)
 c  =
       c(1)
!qd  d  !
       c(2)
    1.
       c(3)
    0.
       c(4)
    0.
       c(5)
    0.
```

The strings `b(1)` and `c(1)` denote the respective data types. The value `1` is substituted in `b(2)` and `c(2)`. These functions can also be applied to a vector or a matrix. Next, we set a *constant* vector `d=[2;3]`, and convert `d` into a *dd* variable `e`.

```
-->d=[2;3]
 d  =
    2.
    3.
-->e=d2dd(d) // or e=dd(d)
 e  =
       e(1)
!dd  hi  lo  !
       e(2)
    2.
    3.
       e(3)
    0.
    0.
```

2.3.2 Convert dd or qd into constant

To convert the *dd* variable `a` into *constant*, we can use the following code:

```
-->getHi(a)
```

```
-->a.hi
-->a(2)
```

To convert the *qd* variable a into *constant*, we can use the following code:

```
-->getHi(a)
-->a.d
-->a(2)
```

Ex.

```
-->d=[2;3];
-->e=d2dd(d)
 e  =
       e(1)
!dd  hi  lo  !
       e(2)
    2.
    3.
       e(3)
    0.
    0.
-->e.hi
 ans  =
    2.
    3.
-->f = qd(1.2345);
-->getHi(f)
 ans  =
    1.2345
```

2.3.3 Convert between dd and qd

To convert the variable of type between *dd* and *qd*, we can use dd2qd(a) or qd2dd(a).

2.4 Arithmetic operators

We can use the same operators (+,-,*,/) for *dd* and *qd* as *constant*, even if variables are vectors or matrices.

Ex.1

Compute $1 + 10^{-20}$ with using *constant* and *dd*

```
-->1+1.0D-20 //double
 ans  =
    1.
-->e=dd(1);
-->f=dd(1.0D-20);
-->e+f //DD
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    1.
       ans(3)
    1.000D-20
```

Using *constant*, the information loss occurs. In contrast, using *dd*, lower part of *dd* retains the value `1.0D-20`.

Ex.2
Compute $1 \div 3 = 0.333\cdots$ using *constant*, *dd* and *qd* variables.

```
-->format('e',22)
-->f=1/3
 f  =
    3.333333333333333D-01
-->g=dd(1)/dd(3)
 g  =
       g(1)
!dd  hi  lo  !
       g(2)
    3.333333333333333D-01
       g(3)
    1.850371707708594D-17
-->ddprint(g)
 ans  =
    3.33333333333333333333333333333333E-1
-->h=qd(1)/qd(3)
 h  =
       h(1)
!qd  d  !
       h(2)
    3.333333333333333D-01
```

```
        h(3)
    1.850371707708594D-17
        h(4)
    1.027162637006526D-33
        h(5)
    5.701898048196684D-50
-->qdprint(h)
 ans  =
    3.333333333333333333333333333333333333333333333333333333333333333E-1
```

2.4.1 Implicit type conversion

We can also compute the sum of variables of *dd* and *constant* using the operator `+`. In other words, mixed precision arithmetic is also available. When we do mixed precision arithmetic of *dd* and *constant*, they will be automatically converted to *dd* in the computation. The same rule applies to *qd*. It is like an implicit type conversion between 'int' and 'double' on usual programming language.

2.5 The other operators

2.5.1 Relational operators

We can use the same operator (`==,~=,<,>,<=,>=`) for *dd* and *qd* as *constant*. These operators return `%T` or `%F`.

Ex.

```
-->format('e',22)
-->a=1/7;
-->b=1/dd(7);
-->c=1/qd(7);
-->disp(a)
    1.428571428571428D-01
-->ddprint(b)
 ans  =
    1.42857142857142857142857142857 1E-1
-->qdprint(c)
 ans  =
    1.428571428571428571428571428571428571428571428571428571428571 43E-1
-->a==b
 ans  =
  F
```

```
-->a<b
 ans  =
  T
-->a~=c
 ans  =
  T
-->a==c.d
 ans  =
  T
```

2.5.2 The others

- `A'`

  Returns the transpose of `A`.

2.6 Scilab functions

We can use some functions for *dd* and *qd* variables.

2.6.1 The same function as constant

The following functions which have the same as the functions of constant can be used for *dd* and *qd* variables.

- `sqrt(a)`

  Computes a square root of `a`.
- `sin(a)`

  Returns a sine of given angle.
- `cos(a)`

  Returns a cosine of given angle.
- `tan(a)`

  Returns a tangent of given angle.
- `exp(a)`

  Returns a exponent of given argument.
- `abs(a)`

  Returns an absolute value of `a`.
- `norm(x,n)`

  Computes some kind of vector norm of `x` depending on the second argument `n` If `x` is a *dd* or *qd* variable, the second argument can be `1`, `2` or `%inf`.
- `[L,U] = lu(A)`

  Computes the lu factolization of `A`. This function produces two matrices `L` and `U`. Types of

L and U are the same as the argument A.

· [Q,R] = qr(A)

Computes the QR decomposition $A = QR$ where Q is an orthogonal matrix and R is an upper triangular matrix.

2.6.2 The other functions

The other useful functions we defined for *dd* and qd are following:

· ddnrt(a,b)

Computes a n-th root of a *dd* variable a. Assume that the second argument n is an integer.

· ddzeros(m,n)

Defines a zero matrix of type *dd* that size is m by n.

· qdzeros(m,n)

Defines a zero matrix of type *qd* that size is m by n.

· ddeye(m,n)

Defines an identity matrix of type *dd* that size is m by n.

· qdeye(m,n)

Defines an identity matrix of type *qd* that size is m by n.

· ddones(m,n)

Returns a *dd* matrix made of ones that size is m by n.

· qdones(m,n)

Returns a *qd* matrix made of ones that size is m by n.

· ddrand(m,n)

Returns a (m,n) pseudorandom *dd* matrix.

· qdrand(m,n)

Returns a (m,n) pseudorandom *qd* matrix.

· ddip(x,y)

Computes a dot product of *dd* vectors x and y.

· qdip(x,y)

Computes a dot product of *qd* vectors x and y.

· ddnormF(a)

Computes a Frobenius norm of a *dd* matrix a.

· qdnormF(a)

Computes a Frobenius norm of a *qd* matrix a.

· ddGauss(A,b)

Executes Gaussian elimination for solving a linear equation that contains *dd* variables.

· qdGauss(A,b)

Executes Gaussian elimination for solving a linear equation that contains *qd* variables.

· ddinv(A)

Returns the inverse matrix of *dd* matrix A based on the LU factorization. (This function is included only *MuPAT_win* and *MuPAT_maclin*)

· `qdinv(A)`
  Returns the inverse matrix of *qd* matrix `A` based on the LU factorization. (This function is included only *MuPAT_win* and *MuPAT_maclin*)

Ex.

To create an identity matrix of type *dd* that the size is `2` by `2`, we enter the following code.

```
-->ddeye(2,2)
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    1.    0.
    0.    1.
       ans(3)
    0.    0.
    0.    0.
```

2.6.3 Mathematical constants

MuPAT has these mathematical constants.

· `ddpi()`
  Returns circular constant of *dd* that holds 31 significant digits.

· `qdpi()`
  Returns circular constant of *qd* that holds 63 significant digits.

Ex.

```
-->ddpi()
 ans  =
       ans(1)
!dd  hi  lo  !
       ans(2)
    3.1415927
       ans(3)
    1.225D-16
-->qdprint(qdpi())
 ans  =
    3.141592653589793238462643383279502884197169399375105820974944459E0
```

Bibliography

[1] ATOMS, http://atoms.scilab.org/toolboxes/DD_QD

[2] MuPAT and QuPAT, http://www.mi.kagu.tus.ac.jp/qupat.html

[3] Y. Hida, X. S. Li and D. H. Bailey, Quad-double arithmetic: algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (2000).

[4] Scilab Online Help, http://help.scilab.org/

[5] T. Saito, E. Ishiwata and H. Hasegawa, Development of quadruple precision arithmetic toolbox qupat on scilab, ICCSA2010, Proceedings Part II, LNCS 6017, 60-70 (2010).

A Contents of Toolbox

A.1 MuPAT

```
MuPAT
 ├ macros
 |   ├ buildmacros.sce
 |   ├ loadmacros.sce
 |   ├ dd.sci
 |   ├ qd.sci
 |   └ ········· (set of functions)
 ├ help
 |   ├ en_US
 |   |   ├ build_help.sce
 |   |   ├ load_help.sce
 |   |   ├ dd.xml
 |   |   └ ········· (set of help documents)
 |   └ builder_help.sce
 ├ etc
 |   ├ MuPAT.start
 |   └ MuPAT.quit
 ├ builder.sce (First we have to load into Scialb to run MuPAT)
 ├ loader.sce (Second we have to load into Scialb to run MuPAT)
 ├ changelog.txt
 ├ licence.txt
 ├ readme.txt
 └ users_guide.pdf (This file)
```

A.2 MuPAT_win

```
MuPAT_win
 ├ macros
 │   ├ buildmacros.sce
 │   ├ loadmacros.sce
 │   ├ dd.sci
 │   ├ qd.sci
 │   ├ ········· (set of functions)
 │   ├ dd.dll
 │   ├ qd.dll
 │   └ ········· (set of dynamic link library)
 ├ help
 │   ├ en_US
 │   │   ├ build_help.sce
 │   │   ├ load_help.sce
 │   │   ├ dd.xml
 │   │   └ ········· (set of help documents)
 │   └ builder_help.sce
 ├ etc
 │   ├ MuPAT.start
 │   └ MuPAT.quit
 ├ builder.sce (First we have to load into Scialb to run MuPAT_win)
 ├ loader.sce (Second we have to load into Scialb to run MuPAT_win)
 ├ changelog.txt
 ├ licence.txt
 ├ readme.txt
 └ users_guide.pdf (This file)
```

A.3 MuPAT_maclin

```
MuPAT_maclin
  ├ macros
  |   ├ buildmacros.sce
  |   ├ loadmacros.sce
  |   ├ dd.sci
  |   ├ qd.sci
  |   ├ ········· (set of functions)
  |   ├ dd.c
  |   ├ qd.c
  |   └ ········· (set of C programs)
  ├ help
  |   ├ en_US
  |   |   ├ build_help.sce
  |   |   ├ load_help.sce
  |   |   ├ dd.xml
  |   |   └ ········· (set of help documents)
  |   └ builder_help.sce
  ├ etc
  |   ├ MuPAT.start
  |   └ MuPAT.quit
  ├ builder.sce (First we have to load into Scialb to run MuPAT_maclin)
  ├ loader.sce (Second we have to load into Scialb to run MuPAT_maclin)
  ├ changelog.txt
  ├ licence.txt
  ├ readme.txt
  └ users_guide.pdf (This file)
```