Towards a multiprecision MATLAB environment

Walter Schreppers
CANT, Universiteit Antwerpen

November 17, 2004
WOG symposium, Universiteit Gent

Overview

1. current type of systems: symbolic vs. hardware precisions (the gap?)
2. tools used to create the new system MPL (precompiler, custom MATLAB parser)
3. results and demo of MPL

Availability of high precision numeric routines

- computer algebra systems: symbolic and exact arithmetic, high precision floating-point (e.g. Maple, Mathematica, . . .)
  $\Rightarrow$ but sadly things like NSOLVE are not IEEE 854 compliant
- popular programming environments: standard hardware precisions (float, double) (e.g. MATLAB, Octave, GSL, . . .)

Is there a need for high precision?

increase of memory/speed of computers

$\Downarrow$

larger mathematical models to be solved

$\Downarrow$

floating-point with unit roundoff of 10^{-32}

$\Downarrow$

extend MATLAB with high precision capabilities!

Prototyping vs. compiled code

- programming environment: experimental (interpreted) code (debugging)
- export and compile optimized code of final version

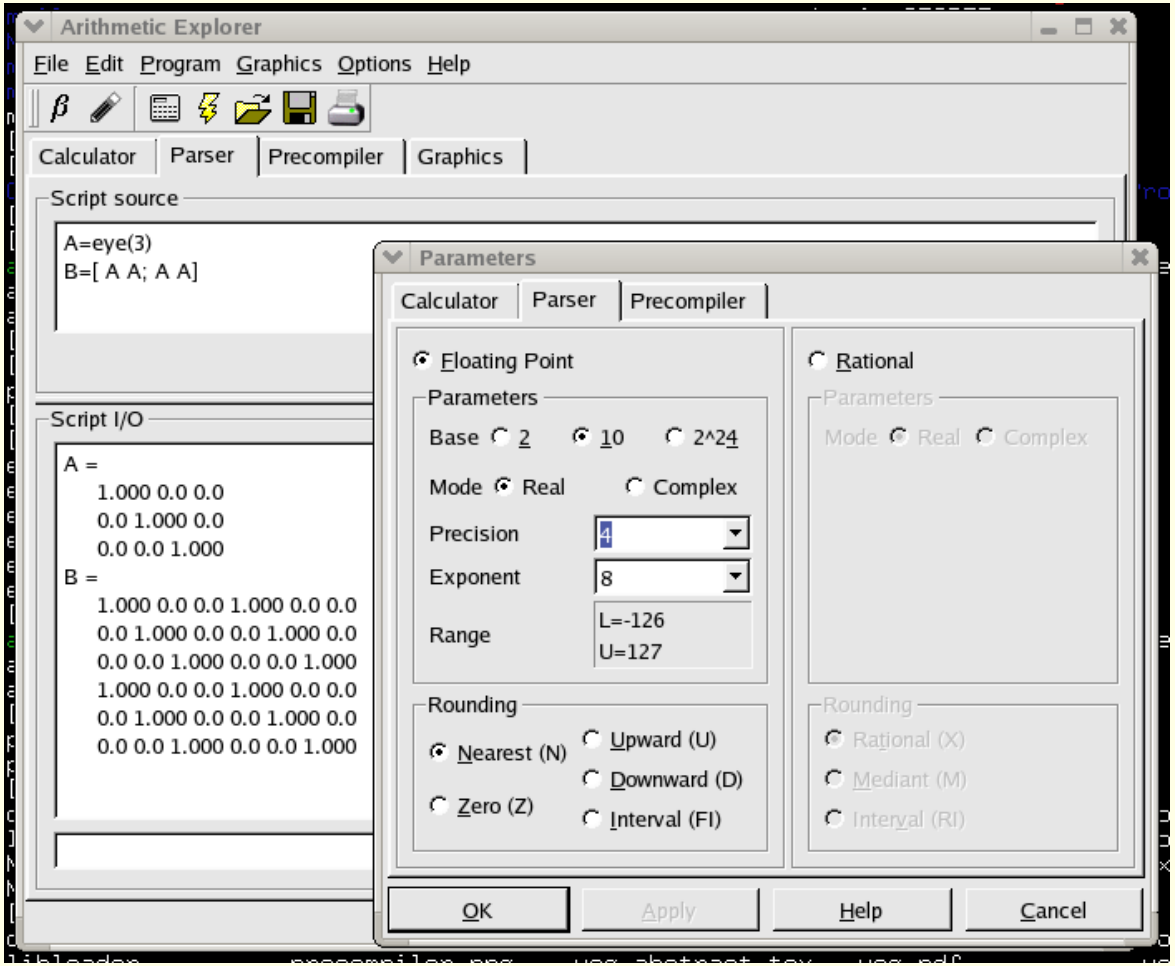
$$\Downarrow$$

also compiled code is important!

High precision?

- high? $\mathrm{t} \gg 64$ bits precision, $\beta = 2^{\mathrm{i}}$ or 10^{j}
- preferably IEEE 854 compliant
  $\Rightarrow$ additional benefit compared to CAS:
  they do not comply with floating-point standards!
- infinite precision based on GMP

Parameter settings in GUI



"Upgrading" existing code from standard to high precision?

- existing multiprecision floating-point libraries:
  MpIeee, CLN, FMLIB, MPFR, MPFUN, . . .
- only MPFUN has a transcription program to convert existing code into multiprecision code
- only MpIeee is fully IEEE 854 compliant

Precompiler for type conversion

- what?
  generic transcription program converts C/C++ code to C++ code using a C++ multiprecision library of choice
- how?
  convert any type in the input to another type in the output using a simple XML configuration file; special care is taken w.r.t. constants
- future?
  converting constants (manual intervention is always necessary!)

Precompiler: sample configuration file

```
<?xml version="1.0"?>
<document>
  <include> BigInt.hh     </include>
  <include> MpIeee.hh     </include>
  <include> ArithmosIO.hh </include>
  <source name="sfloat">
    <keyword> float </keyword>
    <keyword> double </keyword>
  </source>
  <source name="sint">
    <keyword> int </keyword>
  </source>
  <target name="tint"><keyword> BigInt </keyword></target>
  <target name="tfloat"><keyword> MpIeee </keyword></target>
  <rhs name="afloat">
    <token> integer </token>
    <token> decimal </token>
  </rhs>
  <rhs name="aint"><token> integer </token></rhs>
  <!---------- the conversion rules ------------>
  <convert name="float to mpieee">
```

```
    <source name="sfloat"/> <target name="tfloat"/>
  </convert>
  <convert name="int to bigint">
    <source name="sint"/> <target name="tint"/>
  </convert>
  <convert name="decimal to mpieee">
    <rhs name="afloat"/> <target name="tfloat"/>
    <operation> toStringConstructor </operation>
  </convert>
  <convert name="integer to bigint">
    <rhs name="aint"/> <target name="tint"/>
    <operation> toStringConstructor </operation>
  </convert>
</document>
```

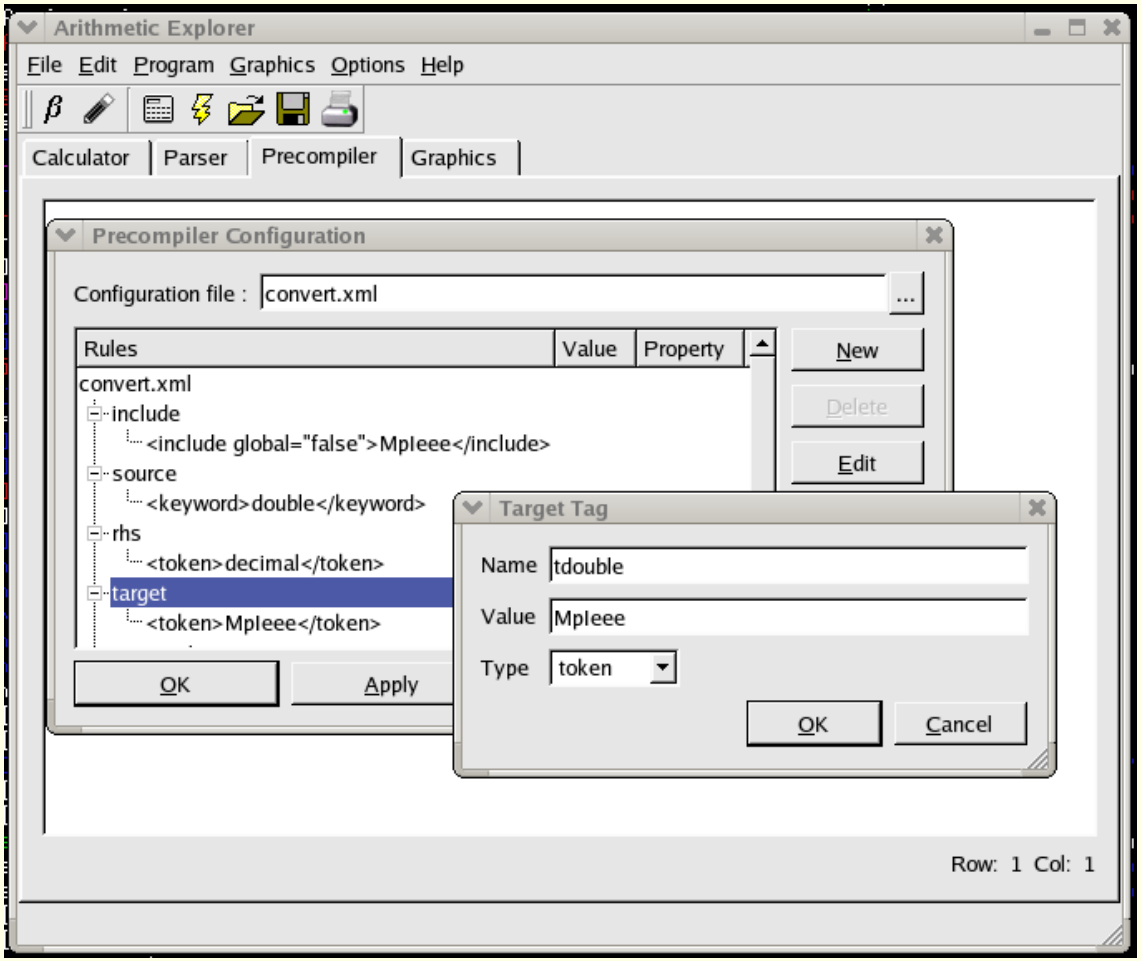
Precompiler: conversion example input

```
#include <stdio.h>

float func(float& a){
  float k;
  for(int i=1;i<5;i++)
    for(int j=1;j<3;j++){
      char k=32;
      if(i+j>5) k='y';
      if(k=='y') printf("k=%c, i+j=%i\n",k,i+j);
    }
  k=2*a;
  return 3*a;
}
...
```

Precompiler: conversion example output

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "BigInt.hh"
#include "MpIeee.hh"
#include "ArithmosIO.hh"
#include <stdio.h>

MpIeee func(MpIeee & a){
  MpIeee k;
  for(BigInt i= BigInt( "1" );i<BigInt( "5" );i++)
    for(BigInt j= BigInt( "1" );j<BigInt( "3" );j++){
      char k=32;
      if(i+j>BigInt( "5" )) k='y';
      if(k=='y') {cout<<"k="<<k<<", i+j="<<i+j<<"\n";}
    }
  k=MpIeee( "2" )*a;
  return MpIeee( "3" )*a;
}
...
```

Precompiler: conversion in GUI



High precision matrix library

- basic MATLAB type is a matrix
- existing matrix libraries? e.g. MTL, TNT, Boost, all "fully templated" . . .
- . . . but not in reality! hard coded **`float`**, **`double`** and **`int`** appear in MTL and TNT $\Rightarrow$ useless for true multiprecision classes.
- Boost: templated C++ classes for dense, sparse, triangular, banded, . . . ; (BLAS level 1,2,3)

MPL

- what?
  MATLAB compatible parser
- aim?
  - high precision capabilities
  - dynamically (un)load shared libraries in the parser (comparable with MATLAB's libloader), e.g. Numerical Recipes and GSL

Multiprecision MATLAB parser MPL

- syntax/semantics had to be reverse engineered (documentation helped e.g. for operator precedence)
- implementation in C++ without use of lex/yacc, OO design and can be compiled on many platforms/compilers (MacOS X, Windows, Linux, Solaris, ...)
- subset of MATLAB languange but superset in types: MpMatrix, BigMatrix, RatMatrix, CMpMatrix, ... (with mode function)
- various toolboxes included (Automatic Diff., LibLoader, ...)
- parser and compiler can run as standalone or integrated in the GUI
- is able to compile scripts into C++ $\Rightarrow$ standalone exe

Challenges when calling external routines

- arguments?
  - conversion of the arguments inside the parser to the arguments of the C++ functions:
    - efficiency
    - creating the desired arguments at runtime? what about structs/classes?
- matrix bounds?
  - dimensions of the arguments themselves (e.g. out of bounds indices)
- help?
  - which functions can be used from the library, what are their arguments and what routines do they implement?

LibLoader

- based on `libtool` and `ffcall`. These are grouped in a wrapper class LibLoader and uses some other classes like Argument etc.
- LibLoader provides capability to load shared libraries and to call their functions at runtime
- argument types of functions are limited (double, double*, double**, char and float variants and void*!) $\Rightarrow$ the void pointer used to circumvent compiler type checking and allowing classes to be passed!

Interfacing to Numerical Recipes

- documentation how to use routines with MTL and TNT . . .
  $\Rightarrow$ create own Boost wrappers
- use precompiler to convert standard variables and constants in NR routines
- get the largest subset of working routines (79% at this moment) into a shared library

Numerical Recipes Example (1)

- converted Numerical Recipes Library using precompiler and writing wrapper matrix classes for MpIeee manually
- we have our MATLAB parser and a libloader toolbox which can load these libraries
- how do I use these routines in practice?

Numerical Recipes Example: the script (2)

```
libname = '/home/wschrep/ArithmosRelease/Programs/Parser/scripts/recipes';
loadlibrary( libname );

if libisloaded( libname )
        disp ( 'ok, recipes loaded :)' )
        calllib 'recipes' , 'dummytest', 1 ,2 , 3, 4, 5 %what will happen?
        calllib 'recipes' , 'rtflsp', 8.8, 9.2, 1e-64
        unloadlibrary( libname );
else
        disp('recipes not loaded');
end
```

Numerical Recipes Example, the C routine rtflsp(3)

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtflsp( DP func(const DP), const DP x1, const DP x2, const DP xacc )
{
  cout << "in rtflsp at line " << __LINE__ <<endl;
  cout << " x1          = " << x1 << endl;
  cout << " x2          = " << x2 << endl;
  cout << " xacc        = " << xacc << endl;
  cout << " func adres  = " << &func << endl;

        const int MAXIT=30;
        int j;
        DP fl,fh,xl,xh,dx,del,f,rtf;
  cout << "in rtflsp at line " << __LINE__ <<endl;
        fl=func(x1);
        fh=func(x2);
```

```
cout << "fl = " << fl << endl;
cout << "fh = " << fh << endl;


      if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
      if (fl < 0.0) {
              xl=x1;
              xh=x2;
      } else {
              xl=x2;
              xh=x1;
              SWAP(fl,fh);
      }
      dx=xh-xl;
      for (j=0;j<MAXIT;j++) {
              rtf=xl+dx*fl/(fl-fh);
              f=func(rtf);
              if (f < 0.0) {
                      del=xl-rtf;
                      xl=rtf;
                      fl=f;
              } else {
                      del=xh-rtf;
                      xh=rtf;
```

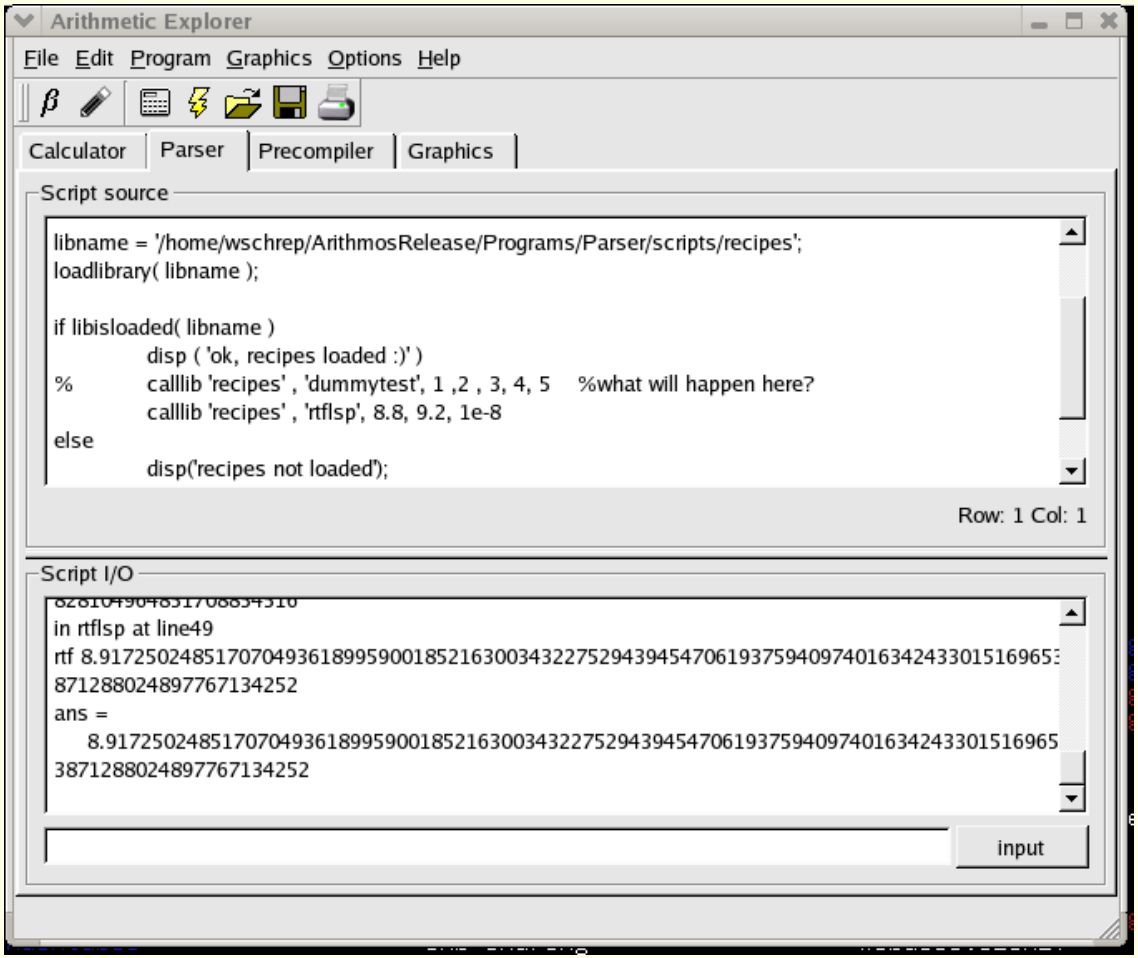
```
                fh=f;
            }
            dx=xh-xl;
            cout << "rtf " << rtf << endl;
            if (fabs(del) < xacc || f == 0.0) return rtf;
        }
        nrerror("Maximum number of iterations exceeded in rtflsp");
        return 0.0;
}
```

Numerical Recipes Example: the script? (4)

```
MpIeee funcWilk(MpIeee x)
{
  MpIeee coeff[21];
  coeff[0]  = MpIeee( "24329020081766400000" );
  coeff[1]  = -MpIeee( "87529480367616000000" );
  coeff[2]  = MpIeee( "138037597536407040000" );
  coeff[3]  = -MpIeee( "128709312451509888000" );
  coeff[4]  = MpIeee( "80378118226450517760" );
  coeff[5]  = -MpIeee( "35999795179476072000" );
  coeff[6]  = MpIeee( "12066478037803733600" );
  coeff[7]  = -MpIeee( "3113336431613906400" );
  coeff[8]  = MpIeee( "630308120992948960" );
  coeff[9]  = -MpIeee( "101422998655114500" );
  coeff[10] = MpIeee( "13075350105403950" );
  coeff[11] = -MpIeee( "1355851828995300" );
  coeff[12] = MpIeee( "113102769953810" );
  coeff[13] = -MpIeee( "7561111845000" );
  coeff[14] = MpIeee( "401717716300" );
  coeff[15] = -MpIeee( "16722808200" );
```

```
  coeff[16] = MpIeee( "53327946" );
  coeff[17] = -MpIeee( "1256850" );
  coeff[18] = MpIeee( "20615" );
  coeff[19] = -(MpIeee( "210" ) + MpIeee("1.1920928955078125^-7"));  // 2^(-23)
  coeff[20] = MpIeee( "1" );

  MpIeee result;
  int i;

  result=coeff[20];
  for(i= 19 ;i>=0 ;i--)
    result=result*x+coeff[i];

  return result;
}
```

Numerical Recipes Example (5)



Questions?

The End.