MPL: a multiprecision Matlab-like environment

Walter Schreppers, Franky Backeljauw, and Annie Cuyt

University of Antwerp (CMI)
Department of Mathematics and Computer Science
Middelheimlaan 1, B-2020 Antwerpen, Belgium
{walter.schreppers,franky.backeljauw,annie.cuyt}@ua.ac.be

Abstract. A number of generic tools, some developed by the authors, some developed in cooperation with other teams and others available freely, are combined into an environment, called MPL from Multi Precision Lab, which offers a cross-platform variable precision alternative to Matlab. Among the tools we mention for instance our C/C++ precompiler for type conversion, the GMP arithmetic library complemented with our own IEEE-854 compliant multi-radix multiprecision MpIeee library, the Boost matrix library, our own Matlab parser, the libraries FFCall and GNU Libtool. The functionality of the well-known Matlab toolboxes is available through the multiprecision equivalent of one's library of choice, generated using the same tools. We mention, among others, GSL, Numerical Recipes, an automatic differentiation toolkit [1], a hybrid polynomial solver [2] and so on.

1 Introduction

While symbolic computing environments have the tendency to also support variable precision numeric routines besides symbolic and exact arithmetic, popular numeric programming environments such as Matlab usually do not offer any higher precisions besides the standard hardware precisions.

Since predictions, based on the growth in the size of mathematical models solved as the memory and speed of computers increase, suggest that floating-point arithmetic with unit roundoff of the order of 10^{-32} is needed for some applications on future supercomputers, we want to investigate the possibility to offer high precision floating-point and exact rational arithmetic in a Matlab-like environment.

Furthermore, numeric code that has passed an experimental stage, is often run in optimized and compiled form and not from within a computing environment. This feature is supported as well.

In the subsequent sections we discuss the building blocks that constitute the MPL environment. The sections 2, 3, 5, 7, 8, and 9 all concern packages developed by ourselves in the past few years.

2 High precision arithmetic

In our context the notion high precision varies from more than 64 digits of binary precision to infinite precision rational arithmetic. A finite precision radix β arithmetic implementation ($\beta = 2^i$ or $\beta = 10^j$) is preferably fully IEEE-854 compliant. Such an implementation offers an additional benefit compared to the symbolic computing environments which do not comply with the floating-point standard(s).

Our infinite precision C++ classes `Rational` and `BigInt` for rational and big integer arithmetic are based on the well-known GMP library [3]. For multiprecision floating-point arithmetic, a number of libraries have been developed in the past decade. We refer among others to MpIeee [4], CLN [5], FMLIB [6], MPFR [7] and MPFUN [8].

`MpIeee` is a C++ class that offers fully IEEE-854 compliant, multi-radix and variable precision floating-point aritmetic. Its implementation allows to encapsulate the data structure together with the routines to create, manipulate and destroy this structure, thus offering an easy-to-use interface to the concept it implements. Furthermore, the possibility of operator overloading allows the use of the ordinary mathematical operators (such as $+$, $-$, $/$, $\times$, $\sqrt{\ }$, but also sin, cos, abs, ...).

Operators have a big impact on the overall runtime performance, as their usage often involves the creation of several temporary objects. These temporaries come from the fact that the operator that is being called, does not know where the result will be stored. Instead, it creates a temporary object to store the result. To actually return the result, yet another (unnamed) temporary object is created, which is nothing more than an implicit copy of the result. It is needed because simply returning a reference to the result would most probably cause memory leakage. It is clearly necessary to try to avoid as many of these temporaries as possible, if not all.

In `MpIeee`, a technique called delayed evaluation is used to avoid the need for temporaries altogether. This technique, as its name indicates, consists in delaying the operation until it knows the existence of a target object to store the result. This is done using some interim object, called a proxy, which simply stores references to the operands and an indication of the operation to which it applies. The actual operation is then performed through a modified assignment operator which takes this interim object as its operand. This way, the assignment operator knows its target object (the left hand side of the operation) as well as which operation needs to be performed on the given operands (the right hand side of the operation). As such, the full computation can be executed without the need for any temporary objects.

This technique can be completely implemented using inline functions, which are almost always resolved at compile time when optimization settings apply. Hence it is the fastest approach we can aim for. In the sequel `MpIeee` is therefore our preferred multiprecision package. Besides being multi-radix and fast, it is also the only one in its class offering full IEEE-854 compliance.

3 Precompiler for type conversion

With the exception of MPFUN, none of the high precision libraries comes with a transcription program to automatically convert existing source code, using standard precisions, into code that uses the multiprecision types of the library. This problem is addressed now.

In [9] we describe an easy to use, generic C/C++ transcription program or precompiler for the conversion of C/C++ source code into new code that uses a C++ multiprecision library of choice. The precompiler can convert any type in the input source code to another type in the output source code. The input source can be C or C++, while the output code generated by the precompiler and using the new types, is C++. The type conversion is based on a simple configuration file, provided by the developer of the multiprecision library or by the user of the precompiler.

During the transcription of the code, special care is taken with respect to constants, among others to avoid the default conversion by the C++ compiler of decimal literals to their standard double precision binary representation. Hence constants need to be signaled to the user of the precompiler to make sure that they are provided with sufficient accuracy. This can be guaranteed either by string initialization from the decimal literal or by providing sufficiently accurate representations in different radices for constants such as π, e, $\ln 2$, $\sqrt{2}$, ...

The precompiler can be told to skip the conversion of certain variables, such as the running variables in `for`-loops or even a complete function implementation. Use of the precompiler saves a lot of time and avoids errors that otherwise easily occur in a manual conversion. At the same time, great care has been taken to obtain precompiled code with performance similar to that of manually converted code.

4 High precision matrix library

The basic Matlab type is a matrix. Several matrix libraries are freely available, among which MTL (Matrix Template Library), TNT (Template Numerical Toolkit) and Boost. While MTL and TNT claim to be fully templated, in reality the code still contains hard coded `float`, `double` and `int` variables. This renders them useless when trying to generate a true multiprecision matrix library by use of the above precompiler. Fortunately, Boost provides templated C++ classes for several types of matrices: dense, sparse, triangular, banded, symmetric, hermitian, etc. The library covers the usual basic linear algebra operations on vectors and matrices and provides BLAS level 1, 2, 3 functionality.

Our matrix library is based on the templated Boost uBLAS routines and uses our high precision data types implemented in the classes `MpIeee`, `Rational` and `BigInt` as its data types. When different types are used in an expression, the arguments are automatically converted to a predefined (larger) type unless the cast operator is used to force conversion to a certain data type. The resulting library is very time and memory efficient by using advanced template techniques similar to the delayed evaluation techniques used by our own high precision classes.

5 Matlab parser

MPL (Multi Precision Lab) implements a subset of the Matlab language but has a superset of multiprecision types. Starting from Matlab scripts, our lexer and parser construct an abstract syntax tree. This tree can then be interpreted in our environment or compiled into C++ sources. In their turn, these C++ sources can easily be compiled into standalone executables. These executables are faster because no iteration in the abstract syntax tree is required.

Here is a short overview of the implemented subset:

- Block encapsulation: `begin`, `end`.
- Loops: `while`, `for`.
- Control structures: `if`, `else`, `elseif`, `switch`, `case`, `otherwise`, `break`.
- Input/Output: `disp`, `print`, `println`, `input`.
- Functions: `function`, `return`.
- Library loader: `loadlibrary`, `calllib`, `unloadlibrary`, `libisloaded`.
- Matrix creation: `zeros`, `ones`, `eye`.
- Complex variables: `i`, `j`.
- Built-in elementary functions: `sin`, `sinh`, `asin`, `asinh`, `cos`, `cosh`, `acos`, `acosh`, `cotan`, `cotanh`, `acotan`, `acotanh`, `tan`, `atan`, `tanh`, `atanh`, `exp`, `exp2`, `exp10`, `log`, `log2`, `log10`.
- Special constant values: `Inf`, `inf`, `NaN`.
- Built-in functions: `transpose`, `colon` (incl. range operations used in for-loops etc.), `inv`, `horzcat`, `vertcat`, `help`, `sqrt`, `pow`, `mod`, `rem`.
- Relational operators : <, >, <=, >=, =, , ==, |, &, ||, && with same precedence as Matlab.
- Arithmetic operations : $+,-,*,.*,./,.\backslash,\wedge,.\wedge$ with same precedence as Matlab.
- Cell array's : basics implemented but not yet complete.

Subsequently, we extend the Matlab language with functions to alter the current floating-point environment settings:

- `mode`: with an argument `mpieee`, `rational`, `complex`, `double`, `int` or `logical` to specify the type.
- `rounding`: with an argument nearest, up, down, zero to specify the rounding.
- `exponent`: two arguments specifying the minimal exponent L and the maximal exponent U with $L = 1 - U$.
- `radix`: one argument which sets the value of the radix.
- `outputmode`: one argument with a value between 1 and 13 for the different output modes and 0 to reset the status flags.

Garbage collection is done using reference counting pointers. This is not as efficient as for instance mark-and-sweep but for our purposes (especially for compiling into readable C++ source code) it is the best option. Also we are able to use the already available shared pointer from the Boost library. Due to the nature of our language we do not have to worry about cycles in object pointers and (inefficient) tracing routines to resolve them.

Strangely, generic memory management libraries do not seem to be freely available. This is a pity because every garbage collecting language (such as Perl, Python, Java, ...) has to implement memory management schemes.

6 Runtime library loader

The runtime library loader LibLoader enables the loading of shared libraries at runtime. This way the MATLAB parser can be extended with various algorithms selected from high precision precompiled versions of numerical libraries of choice such as Numerical Recipes [10] and GSL [11]. An overview of the MPL environment and more precisely how LibLoader fits in the picture, is shown in Figure 1.

The implementation of LibLoader consists of various wrapper classes around the free cross-platform libraries FFCall and GNU Libtool. We have to circumvent compile time checking of types by using the `void*` pointer, since it is the only way to pass class objects to functions that are loaded dynamically. Proper use of the precompiler guarantees that the object types of the loaded library and the types used in our interpreter and compiler are identical.

Another pitfall when loading arbitrary libraries and calling arbitrary routines is the precise knowledge of the arguments and their dimension(s). Without this information, no successful calls can be made without getting errors, or worse, memory leaks. This is not surprising, since the same caution is required when calling the routines from ordinary C/C++.

7 Interfacing to Numerical Recipes

Let us now describe how a multiprecision version of the Numerical Recipes library can be loaded into MPL. First, the precompiler is used to convert every function file so that it uses a multiprecision type of choice. The precompiled output files are then compiled and linked into a shared library. Unfortunately some files do not link or corrupt the generated multiprecision library because of the use of global or external variables. The shared library can then be loaded as follows:

```
libname = '/home/mpl/scripts/recipes';
loadlibrary( libname );
```

However, loading the library at runtime causes `MpIeee` objects to use a separate environment. This environment has to be synchronized with the environment used inside the parser. Therefore one has to add two functions, called

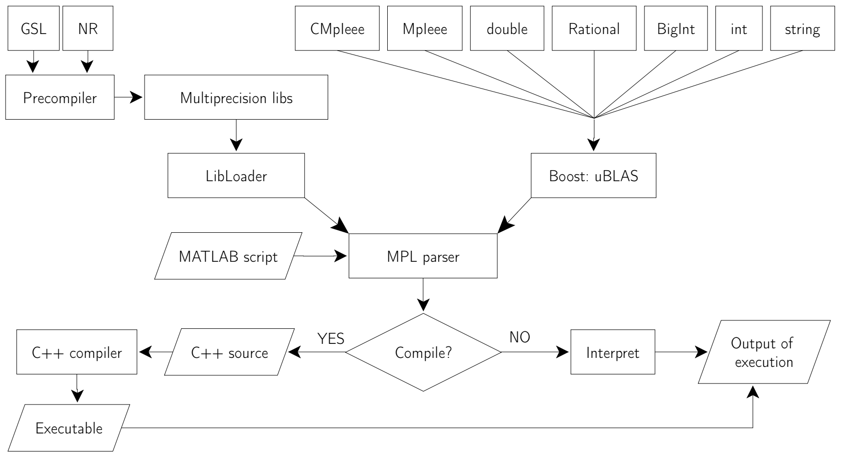

Fig. 1. Overview of MPL environment

`getEnvironment` and `setEnvironment`, to our shared library. These functions are called before and after every call of a Numerical Recipes routine in order to ensure that the correct internal representation is used. We can automate this by placing these extra calls in an external function file:

```
% External function file callrecipes.m
function result = callrecipes( funcname, varargin )
  libname = 'recipes';

  %copy current environment to library environment
  calllib libname, 'setEnvironment', MpIeeeEnvironment();

  result = calllib libname, funcname, varargin{:};

  %copy library environment back to current environment
  calllib libname, 'getEnvironment', MpIeeeEnvironment();
end
```

The actual call of a Numerical Recipes routine is then done by:

```
% actual calling of recipes routines
libname = '/home/mpl/scripts/recipes';
loadlibrary( libname );

if libisloaded( libname )
  callrecipes 'rtflsp', 8.8, 9.2, 1e-64
  callrecipes 'rtflsp', 8.8, 9.2, 1e-128  %higher precision
  unloadlibrary( libname );
```

```
else
  disp('recipes not loaded');
end
```

8 Interfacing to GSL

To call routines of the GSL library, we need to follow a slightly different approach. We fine tune the conversion of GSL using the precompiler so that all `double` and `float` instances in computations are replaced by a multiprecision type. The variables used exclusively for array or matrix indexing are left untouched for better performance. After this conversion we still cannot use the GSL routines directly in the parser like we did with Numerical Recipes. The reason for this is that GSL uses C `struct`s and specific memory allocating calls like `malloc` for the implementation of vector, matrix and interval types.

To map the types used in GSL to the types of the matrix library used in the parser, we need interfacing wrapper functions. Here is a skeleton of such a wrapper function:

```
#include <iostream>
#include "matrix.h"
#include <gsl/gsl_math.h> //gsl specific includes
#include <gsl/gsl_errno.h>
#include <gsl/gsl_roots.h>

void mexFunction(int nlhs, Matrix *plhs[], int prhs,
                 const Matrix *prhs[]){
  // convert the Matrix arguments in prhs
  // to needed structs defined in gsl.h

  // make the call to the gsl routine, for example:
  // bisection_iterate (void * vstate, gsl_function * f,
  //                    MpIeee * root,
  //                    MpIeee * x_lower,
  //                    MpIeee * x_upper)

  // copy returned value(s) into plhs
}
```

These wrapper functions are very similar to the MATLAB `mex` functions. One way to automate the construction of these wrapper functions is to use the SWIG tool which is also used successfully by Python to import various libraries.

9 Example

We implement a remarkable example [12, 13] where all hardware precisions, even quadruple precision, go wrong:

```
a = 77617;
b = 33096;
y = 333.75*b*b*b*b*b*b + ...
    a*a * (11*a*a*b*b - b*b*b*b*b*b - 121*b*b*b*b - 2) + ...
    5.5*b*b*b*b*b*b*b*b + a/(2*b)
```

This gives the wrong answer $y = 1.1726$ in MATLAB 6.5 revision 13 on an Intel Pentium 3 based system, as well as in MPL through `mode(double)`. Setting the radix and precision to 10 and 37 in MPL, by adding the commands

```
mode( mpieee );
precision( 37 );
radix( 10 );
```

the computed result for y is

```
y = -8.273960599468213681411650954798162920^-1
```

as it should be. MPL also allows a correct value for y to be computed in rational arithmetic.

This example shows that it is very straightforward to go from hardware to multiprecision using our MPL environment. All the functionality is also available from within a cross-platform GUI: run scripts, set the radix, increase or decrease the precision and exponent range, change the rounding modes and default types.

References

1. Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: C++ Toolbox for Verified Computing. Springer Verlag, Berlin, Heidelberg (1995)
2. Bini, D., Fiorentino, G.: Design, analysis, and implementation of a multiprecision polynomial rootfinder. Numerical Algorithms **23** (2000) 127–173
3. Granlund, T.: GNU MP: The GNU Multiple Precision Arithmetic Library. (2004)
4. Cuyt, A.: http://www.cant.ua.ac.be/arithmos/ (2004)
5. Haible, B.: CLN, a class library for numbers. (1997)
6. Smith, D.M.: Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic. ACM Trans. Math. Software **17** (1991) 273–283
7. Zimmermann, P., et al.: MPFR: a library for multiprecision floating-point arithmetic with exact rounding. (2000)
8. Bailey, D.: A FORTRAN 90-based multiprecision system. ACM Trans. Math. Software **21** (1995) 379–387
9. Schreppers, W., Cuyt, A.: A generic C/C++ precompiler. ACM Trans. Math. Software (2004) submitted.
10. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical recipes in C++. Cambridge University Press, Cambridge (2002)
11. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M., Rossi, F.: GNU Scientific Library Reference Manual. Second edn. Network Theory Ltd. (2003)
12. Cuyt, A., Verdonk, B., Becuwe, S., Kuterna, P.: A remarkable example of catastrophic cancellation unraveled. Computing **66** (2001) 309–320
13. Rump, S.: Algorithms for verified inclusions - theory and practice. In Moore, R., ed.: Reliability in Computing. (1988) 109–126