A Thread-Safe, Arbitrary Precision Computation Package

David H. Bailey
http://www.davidhbailey.com
Lawrence Berkeley National Lab (retired)
University of California, Davis

March 17, 2015
Some applications where high precision is essential

1. Planetary orbit calculations (32 digits).
2. Supernova simulations (32–64 digits).
3. Climate modeling (32 digits).
4. Optimization problems in biology and other fields (32 digits).
5. Coulomb n-body atomic system simulations (32–120 digits).
8. Scattering amplitudes of fundamental particles (32 digits).
10. The Taylor algorithm for ODEs (100–600 digits).
11. Ising integrals from mathematical physics (100–1000 digits).
12. Problems in experimental mathematics (100–50,000 digits).
Alexei Frolov of Queen’s University in Canada has used high-precision arithmetic to solve a generalized eigenvalue problem that arises in Coulomb n-body interactions. Matrices are typically 5,000 x 5,000 and are very nearly singular. Computations typically involve massive cancellation, and high-precision arithmetic must be employed to obtain numerically reproducible results. Frolov has also computed elements of the Hamiltonian matrix and the overlap matrix in four- and five-body systems. These computations typically require 120-digit arithmetic.

Frolov: “We can consider and solve the bound state few-body problems ... beyond our imagination even four years ago.”

Taylor’s method for ODEs with high-precision arithmetic

Numerical integration of the L25-R25 unstable periodic orbit for the Lorenz model during 16 time periods using TIDES with 300 digits, versus 1 time period using DP.
Experimental mathematics: Discovering new mathematical results

Methodology:

1. Compute various mathematical entities (limits, infinite series sums, definite integrals, etc.) to high precision, typically 100–10,000 digits.

2. Use algorithms such as PSLQ to recognize these numerical values in terms of well-known mathematical constants.

3. When results are found experimentally, seek formal mathematical proofs of the discovered relations.

Many results have recently been found using this methodology, both in pure mathematics and in mathematical physics.

“If mathematics describes an objective world just like physics, there is no reason why inductive methods should not be applied in mathematics just the same as in physics.” – Kurt Godel
The PSLQ integer relation algorithm

Let \((x_n)\) be a given vector of real numbers. An integer relation algorithm either finds integers \((a_n)\) such that

\[
a_1x_1 + a_2x_2 + \cdots + a_nx_n = 0
\]

(to within the “epsilon” of the arithmetic being used), or else finds bounds within which no relation can exist.

- The “PSLQ” algorithm of mathematician-sculptor Helaman Ferguson is the most widely used integer relation algorithm.
- Integer relation detection requires very high precision (at least \(n \times d\) digits, where \(d\) is the size in digits of the largest \(a_k\)), both in the input data and the algorithm.
- The multipair PSLQ algorithm is parallelizable, but only at a fairly low level. Only low-level, thread-based parallelism is efficient for this algorithm.

High-precision numerical integration

Given $f(x)$ defined on $(-1, 1)$, define $g(t) = \tanh(\pi/2 \sinh t)$. Then setting $x = g(t)$ yields

$$
\int_{-1}^{1} f(x) \, dx = \int_{-\infty}^{\infty} f(g(t))g'(t) \, dt \approx h \sum_{j=-N}^{N} w_j f(x_j),
$$

where $x_j = g(h_j)$ and $w_j = g'(h_j)$. Since $g'(t)$ goes to zero very rapidly for large $t$, the product $f(g(t))g'(t)$ typically is a nice bell-shaped function, so that the simple summation above converges very rapidly. Reducing $h$ by half typically doubles the number of correct digits.

We have found that tanh-sinh is the best general-purpose integration scheme for functions with vertical derivatives or singularities at endpoints, or for any function at very high precision ($> 1000$ digits). Otherwise we use Gaussian quadrature.


Ising integrals from mathematical physics

We applied our methods to study three classes of integrals: $C_n$ are connected to quantum field theory, $D_n$ arise in the Ising theory of mathematical physics, while the $E_n$ integrands are derived from $D_n$:

$$C_n := \frac{4}{n!} \int_0^\infty \cdots \int_0^\infty \frac{1}{\left(\sum_{j=1}^n (u_j + 1/u_j)\right)^2} \frac{du_1}{u_1} \cdots \frac{du_n}{u_n}$$

$$D_n := \frac{4}{n!} \int_0^\infty \cdots \int_0^\infty \frac{\prod_{i<j} \left(\frac{u_i-u_j}{u_i+u_j}\right)^2}{\left(\sum_{j=1}^n (u_j + 1/u_j)\right)^2} \frac{du_1}{u_1} \cdots \frac{du_n}{u_n}$$

$$E_n = 2 \int_0^1 \cdots \int_0^1 \left(\prod_{1 \leq j < k \leq n} \frac{u_k - u_j}{u_k + u_j}\right)^2 dt_2 dt_3 \cdots dt_n$$

where in the last line $u_k = t_1 t_2 \cdots t_k$.

Limiting value of $C_n$: What is this number?

Key observation: The $C_n$ integrals can be converted to one-dimensional integrals involving the modified Bessel function $K_0(t)$:

$$C_n = \frac{2^n}{n!} \int_0^\infty tK_0^n(t) \, dt$$

High-precision numerical values, computed using this formula and tanh-sinh quadrature, approach a limit. For example:

$$C_{1024} = 0.6304735033743867961220401927108789043545870787\ldots$$

What is this number? We copied the first 50 digits into the online Inverse Symbolic Calculator (ISC) at http://carma-lx1.newcastle.edu.au:8087. The result was:

$$\lim_{n \to \infty} C_n = 2e^{-2\gamma}.$$

where $\gamma$ denotes Euler’s constant. This is now proven.
Other Ising integral evaluations found using PSLQ

\[
D_3 = 8 + 4\pi^2/3 - 27 L_{-3}(2)
\]
\[
D_4 = 4\pi^2/9 - 1/6 - 7\zeta(3)/2
\]
\[
E_2 = 6 - 8 \log 2
\]
\[
E_3 = 10 - 2\pi^2 - 8 \log 2 + 32 \log^2 2
\]
\[
E_4 = 22 - 82\zeta(3) - 24 \log 2 + 176 \log^2 2 - 256(\log^3 2)/3
\]
\[
+ 16\pi^2 \log 2 - 22\pi^2/3
\]
\[
E_5 = 42 - 1984 \text{Li}_4(1/2) + 189\pi^4/10 - 74\zeta(3) - 1272\zeta(3) \log 2 + 40\pi^2 \log^2 2
\]
\[
- 62\pi^2/3 + 40(\pi^2 \log 2)/3 + 88 \log^4 2 + 464 \log^2 2 - 40 \log 2
\]

where \(\zeta\) is the Riemann zeta function and \(\text{Li}_n(x)\) is the polylog function. \(E_5\) remained a “numerical conjecture” for several years, but was proven in March 2014 by Erik Panzer.

\(E_5\) was reduced to a 3-D integral of a 60-line integrand, which was evaluated using tanh-sinh quadrature to 250-digit arithmetic using over 1000 CPU-hours on a highly parallel system. The PSLQ calculation required only seconds.
Algebraic numbers in Poisson potential functions

Lattice sums arising from the Poisson equation have been studied widely in mathematical physics and image processing. We numerically discovered, and then proved, that for rational \((x, y)\), the 2-D Poisson potential function satisfies

\[
\phi_2(x, y) = \frac{1}{\pi^2} \sum_{m, n \text{ odd}} \frac{\cos(m\pi x) \cos(n\pi y)}{m^2 + n^2} = \frac{1}{\pi} \log \alpha
\]

where \(\alpha\) is algebraic, i.e., the root of an integer polynomial

\[
0 = a_0 + a_1 \alpha + a_2 \alpha^2 + \cdots + a_n \alpha^n
\]

The minimal polynomials for these \(\alpha\) were found by PSLQ calculations, with the \((n + 1)\)-long vector \((1, \alpha, \alpha^2, \cdots, \alpha^n)\) as input, where \(\alpha = \exp(\pi \phi_2(x, y))\). PSLQ returns the vector of integer coefficients \((a_0, a_1, a_2, \cdots, a_n)\) as output.


Samples of minimal polynomials found by PSLQ

<table>
<thead>
<tr>
<th>k</th>
<th>Minimal polynomial for (\exp(8\pi\phi_2(1/k, 1/k)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(1 + 52\alpha - 26\alpha^2 - 12\alpha^3 + \alpha^4)</td>
</tr>
<tr>
<td>6</td>
<td>(1 - 28\alpha + 6\alpha^2 - 28\alpha^3 + \alpha^4)</td>
</tr>
<tr>
<td>7</td>
<td>(-1 - 196\alpha + 1302\alpha^2 - 14756\alpha^3 + 15673\alpha^4 + 42168\alpha^5 - 111916\alpha^6 + 82264\alpha^7)</td>
</tr>
<tr>
<td></td>
<td>(-35231\alpha^8 + 19852\alpha^9 - 2954\alpha^{10} - 308\alpha^{11} + 7\alpha^{12})</td>
</tr>
<tr>
<td>8</td>
<td>(1 - 88\alpha + 92\alpha^2 - 872\alpha^3 + 1990\alpha^4 - 872\alpha^5 + 92\alpha^6 - 88\alpha^7 + \alpha^8)</td>
</tr>
<tr>
<td>9</td>
<td>(-1 - 534\alpha + 10923\alpha^2 - 342864\alpha^3 + 2304684\alpha^4 - 7820712\alpha^5 + 13729068\alpha^6)</td>
</tr>
<tr>
<td></td>
<td>(-22321584\alpha^7 + 39775986\alpha^8 - 44431044\alpha^9 + 19899882\alpha^{10} + 3546576\alpha^{11})</td>
</tr>
<tr>
<td></td>
<td>(-8458020\alpha^{12} + 4009176\alpha^{13} - 273348\alpha^{14} + 121392\alpha^{15})</td>
</tr>
<tr>
<td></td>
<td>(-11385\alpha^{16} - 342\alpha^{17} + 3\alpha^{18})</td>
</tr>
<tr>
<td>10</td>
<td>(1 - 216\alpha + 860\alpha^2 - 744\alpha^3 + 454\alpha^4 - 744\alpha^5 + 860\alpha^6 - 216\alpha^7 + \alpha^8)</td>
</tr>
</tbody>
</table>

The minimal polynomial for \(\exp(8\pi\phi_2(1/32, 1/32))\) has degree 128, with individual coefficients ranging from 1 to over \(10^{56}\) (see next viewgraph). This computation required 10,000-digit precision. Most of the time was spent on PSLQ, not quadrature.

Other polynomials required up to 50,000-digit precision, with runs extending to several days. The ARPREC software failed on at least one of these problems.
Degree-128 minimal polynomial found for $k = 32$
What is needed for high-precision floating-point software?

1. A rock-solid-reliable arithmetic engine, with precision scalable to 1,000,000 digits or more — 1,000 or even 10,000 digits is no longer enough.

2. Highly efficient algorithms, so single-processor performance is as fast as possible, given constraints of portability and ease of installation.

3. A thread-safe design to facilitate parallel processing, both thread-based (low-level) and message-passing (high-level) parallelism.

4. A comprehensive library of tuned transcendentalss and special functions: sin, cos, exp, gamma, incomplete gamma, polylogarithms, Bessel functions, etc.


6. Interoperability with Maple and Mathematica.
Free software for high-precision computation

1. ARPREC: Arbitrary precision, with numerous algebraic and transcendental functions. High-level interfaces for C++ and Fortran-90.  

2. GFORTRAN: Now provides full REAL*16 (IEEE 128-bit, or 34-digit) support.

3. GMP: Produced by a volunteer effort and distributed under the GNU license.  
   http://gmplib.org.

4. MPFR: C library for multiple-precision floating-point computations with exact rounding, based on GMP.  

5. MPFR++: High-level C++ interface to MPFR.  

6. MPFUN90: Similar to ARPREC, but is written entirely in Fortran-90 and provides only a Fortran-90 interface.  

7. QD: Performs “double-double” (31 digits) and “quad-double” (62 digits) arithmetic. High-level interfaces for C++ and Fortran-90.  
Lessons learned from my own packages: ARPREC and QD

- Complicated configure files and install scripts are a problem — many users are reluctant to use these packages for this reason. Many support issues are for these items, not the underlying multiprecision software.

- My older package MPFUN90 (all Fortran-90) continues to be very popular because of its simplicity — downloads and citations still exceed those of ARPREC.

- Support is a nagging issue, given that Yozo Hida and Brandon Thompson are no longer available to help, and Xiaoye Li is very busy with other tasks.

- Special system settings and system-dependent features are problematic.

- In some cases, language features are not supported by certain systems.

- Double precision constants and expressions in code, e.g., 0.1d0, are not automatically converted to high precision, often producing unexpectedly poor accuracy in multiprecision results. Can something be done here?
Thread safety: A major challenge for arbitrary precision software

Since run times are greatly magnified by high-precision arithmetic, parallel processing is increasingly essential. However, available arbitrary-precision packages are not thread-safe at the application user level, and this kills thread-based parallelization, the most efficient form of parallelization:

- All high-level packages I am aware of use global variables, and these variables are deadly for thread safety.
- Most of these global variables can be dispensed with, but not the working precision level, which is changed frequently within the package itself and often by users also.
- The only reasonable design for user programming is via operator overloading, available in languages such as Fortran-90 and C++, which do not permit the working precision level to be passed.

What can be done?
MPFUN2015: A NEW thread-safe arbitrary precision package

- 100% thread safe. NO global read/write global variables.
- NO initialization required (unless over 20,000 digits required).
- NO system-dependent features or settings — the code only assumes 64-bit IEEE format (it does not depend on rounding mode).
- NO reliance on sophisticated language features that might not be supported.
- Installation is a snap — one compile line.
- Precision is scalable to over billion digits; largest size is over $10^{64,000,000}$.
- A full package of basic arithmetic functions. Higher-level routines for binary-decimal conversion, I/O and transcendentals.
- A full-featured high-level language interface for Fortran-90 is provided, so that most users need only make minor changes to existing double precision code.
- Extensive error checking and a solution to the double precision accuracy problem.
- Currently only Fortran-90 version; C++ version is being developed in Australia.
The MPFUN2015 multiprecision data structure

Each multiprecision (MP) datum consists of a vector of double-precision (DP) floats:

1. Word 0: Total space allocated for array, in words.
2. Word 1: Working precision level (in words) associated with this data.
3. Word 2: Number of mantissa words $N$; sign of word 2 is sign of value.
5. Word 4 to $N + 3$: Mantissa (whole numbers between 0 and $2^{48} - 1$).
6. Word $N + 4$ and $N + 5$: scratch; set to zero in output arrays.

Example: $(12., 6., -2., 0., 3., 70368744177664., 0., 0., 0., 0., 0., 0.)$ represents $-3.25$ (since $70368744177664 = 0.25 \times 2^{48}$). The working precision associated with this data is 6 (as specified in word 1). Note that $N$ mantissa words require at least $N + 6$ total array space.
Solution to the thread safety problem

Note that word 1 of each multiprecision datum is the working precision level associated with that datum (variable or array element). This solves the thread safety problem, although it requires a somewhat different programming style at the user level:

- The user sets a default (maximum) working precision level in a compile-time data statement.
- The result of any operation involving MP variables or array elements “inherits” the working precision level of the input operands.
- In the case of an operation with two or more operands with different working precision levels, the largest is chosen for the result.
- The built-in function \texttt{mpreal} allows one to convert a MP variable or array element to one with a different working precision level.
- The built-in function \texttt{mpwprec} returns the working precision level currently assigned to any MP variable or array element.
- Complications: Assignments to DP variables or constants, reading values from file, etc. (See later viewgraphs.)
The following low-level routines are provided:

- Addition, subtraction, multiplication and division of MP arguments.
- DP-to-MP and MP-to-DP conversion; MP times DP and MP divided by DP.
- Square root routine employs a Newton iteration with a dynamically varying precision level (roughly doubling with each iteration).
- $n$-th powers and $n$-roots.
- Nearest integer.
- Rounding and normalization.
Binary-to-decimal and I/O routine module

- Convert MP to decimal character string in En1.n2 format.
- Convert MP to decimal character string in Fn1.n2 format.
- Convert decimal character string to MP (may be in E or F format).
- Input decimal value from input file (may be in E or F format).
- Output MP variable or array element to file in E format.
Transcendental constant and function module

- log(2) and π to arbitrary precision. For requested precision levels less than 20,000 digits, the binary value is merely copied from a compile-time data statement.
- exp(x) and log(x).
- cos(x) and sin(x).
- cosh(x) and sinh(x).
- Conversion of (x, y) to angle in (−π, π).
- Routines use somewhat tricky variations of relatively conventional algorithms.

None of these routines requires any precalculated tables. This preserves thread safety.
Thread-safe algorithms for \( \exp(x) \) and \( \sin(x) \)

Arbitrary precision packages require an initialization step (not thread safe!) to precalculate data for \( \exp \) and \( \cos/\sin \). MPFUN2015 does not:

- The \( \exp \) routine first reduces the input argument to within the interval \((−\log(2)/2, \log(2)/2)\). Then it divides this result by \(2^{10} = 1024\), producing a very small value, which is then input to the Taylor’s series for \( \exp(x) \). When complete, the result is squared ten times, and then corrected for the initial reduction.

- The \( \sin \) routine first reduces the result to within \((−\pi, \pi)\), and then to the nearest quadrant. The result is then divided by 256, producing a very small value, which is then input to the Taylor’s series for \( \sin(x) \). When complete, \( \cos(x) \) is computed as \(\sqrt{1 − \sin^2(x)}\), and then the double-angle formulas for \( \sin(x) \) and \( \cos(x) \) are applied eight times. The result is adjusted for quadrant.

Note that neither scheme requires any precalculated data. Run time performance appears to be within a factor of two of ARPREC schemes, which precalculate extensive tables in an (thread-unsafe) initialization step.
High-level language interface module

This module contains the Fortran language interfaces that connect to all lower-level functions from user source code:

- Binary operations (+, −, ×, ÷).
- Intrinsics (sin, cos, exp, etc.).
- I/O facilities.
- Some special functions and subroutines provided as a part this package.
Easy compilation and linking

The package is simple to compile. Using the gfortran compiler, for instance, type:

```fortran
$ gfortran -O2 -c mpmodule.f90 mpfuna.f90 mpfunb.f90 mpfunc.f90 mpfund.f90 mpfunf.f90
```

You may have to repeat this command to fully resolve the modules. Thereafter, to compile and link an application program prog, type:

```fortran
$ gfortran -O2 prog.f90 mpmodule.o mpfuna.o mpfunb.o mpfunc.o mpfund.o mpfunf.o
```
Basic Fortran application programming instructions

Almost any working Fortran-90 code using double precision (DP) arithmetic can be easily converted to use the package:

- To invoke the package, place this line in every subprogram that contains a multiprecision variable or array:
  use mpmodule
- To designate a variable or array as MP, use the Fortran-90 type statement:
  type (mp_real) a, b(m), c(m,n)
- All basic operations are supported just as in conventional code, e.g.:
  d = a + b(i) * sqrt(3.d0 - c(i,j))
- The following standard Fortran-90 intrinsics are supported with MP arguments:
  abs, acos, aint, anint, asin, atan, atan2, cos, cosh, dble, exp, log, max, min, sign, sin, sqrt, tan, tanh
Handling double precision constants and expressions

- Problem: The code \( a = 3.14159d0 \), where the LHS is MP, does NOT produce the full-precision equivalent of 3.14159, since by standard rules of precedence, the RHS is evaluated in DP, then converted (by zero extension) to LHS.

- Solution: Write this as \( a = '3.14159' \). In MPFUN2015, this forces the constant to be evaluated to full precision.

- Related problem: The code \( b = a + 3.d0 * \sqrt{2.d0} \) does NOT produce a fully accurate result, since \( 3.d0 * \sqrt{2.d0} \) is performed in DP.

- Solution: Write this as \( b = a + 3.d0 * \sqrt{\text{mpreal}(2.d0)} \). This forces all operations to be done using MP routines.

Run-time solution to DP accuracy problems: The MPFUN2015 software checks all DP entities (constants, variables and expression values) at execution time to see if they have more than 40 significant bits. If so, it is flagged as an error. This feature catches 99.99% of the precision loss problems due to DP usage.

If you are sure the DP value is OK as is, use special routines (see next viewgraph).
Functions and subroutine provided by the package

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpcsshf</td>
<td>Subroutine</td>
<td>Returns both cosh and sinh of MP argument.</td>
</tr>
<tr>
<td>mpcssnf</td>
<td>Subroutine</td>
<td>Returns both cos and sin of MP argument.</td>
</tr>
<tr>
<td>mplog2</td>
<td>MP function</td>
<td>Returns log(2).</td>
</tr>
<tr>
<td>mpeform</td>
<td>Subroutine</td>
<td>Returns a character string with MP argument in E format.</td>
</tr>
<tr>
<td>mpfform</td>
<td>Subroutine</td>
<td>Returns a character string with MP argument in F format.</td>
</tr>
<tr>
<td>mpnrtf</td>
<td>MP function</td>
<td>Returns the $n$-th root of MP argument.</td>
</tr>
<tr>
<td>mppi</td>
<td>MP function</td>
<td>Returns $\pi$.</td>
</tr>
<tr>
<td>mpprodd</td>
<td>MP function</td>
<td>Returns the product of MP and DP args (without checking).</td>
</tr>
<tr>
<td>mpquotd</td>
<td>MP function</td>
<td>Returns the quotient of MP and DP args (without checking).</td>
</tr>
<tr>
<td>mpread</td>
<td>Subroutine</td>
<td>Reads MP argument from file.</td>
</tr>
<tr>
<td>mpreal</td>
<td>MP function</td>
<td>Converts the argument (DP, integer or string) to MP.</td>
</tr>
<tr>
<td>mpreald</td>
<td>MP function</td>
<td>Converts the argument (DP) to MP (without checking).</td>
</tr>
<tr>
<td>mpwprec</td>
<td>Int function</td>
<td>Returns working precision (in words) assigned to MP argument.</td>
</tr>
<tr>
<td>mpwrite</td>
<td>Subroutine</td>
<td>Writes MP argument to output unit in E format.</td>
</tr>
</tbody>
</table>
Dynamically changing the working precision

There are two versions of the language interface module:

1. Version 1 (in file mpfunf1.f90): Precision specifications are optional. Intended for applications with only occasional changes in the working precision level, or none.
2. Version 2: (in file mpfunf2.f90): Precision specifications are required. Intended for applications that frequently change the working precision level.

In particular, with Version 1:

- Assignments of the form $a = x$, where $a$ is MP and $x$ is DP, integer or literal, are permitted ($a$ is converted to MP with the default precision level).
- The functions \texttt{mpreal} (which converts a DP, integer or literal argument to MP), \texttt{mpread} (which inputs MP data from file), \texttt{mplog2} (which returns $\log(2)$), and \texttt{mppi} (which returns $\pi$) have an (optional) integer working precision argument.

With Version 2:

- Assignments of the form $a = x$, where $a$ is MP and $x$ is DP, integer or literal, are not permitted. You must use \texttt{mpreal} with precision level parameter instead.
- The optional precision level arguments in the above routines are \textit{required}. 

Working applications

One-, two- and three-level multipair PSLQ. These are workhorses of my own research. They are very unforgiving of even minor arithmetic errors:

- One-level multipair PSLQ (907 lines): Constant precision level.
- Two-level multipair PSLQ (1682 lines): Changes between MP and DP.
- Three-level multipair PSLQ (2103 lines): Changes between full precision (1000 digits), medium precision (120 digits) and DP; uses Version 2 of MPFUNF.

The first two were easy to convert — the third was more work.

The tanh-sinh quadrature test suite (911 lines):

- Tests the tanh-sinh quadrature routine on a suite of 15 integrals, producing (in most cases) 400-digit results.
- Many of the integrands are complicated expressions involving various transcendental functions.
- Dynamically changes the working precision level between 400 and 1000 digits: uses version 2 of MPFUNF.
- All tests appear to work satisfactorily.
Remaining work

- Additional testing and documentation.
- Support for the complex MP datatype.
- Implementation of routines for extra-high precision computation (e.g., FFT-based multiplication, quadratically convergent transcendentals, etc.).
- High-level language interfaces for other languages: C++, etc.
- Explore parallel implementations of high-level applications using the package.
Try it (and please send me bug reports)

The MPFUN2015 software is available at:
http://crd-legacy.lbl.gov/~dhbailey/mpdist.

The download zip file currently includes:

- All MPFUN2015 Fortran modules.
- Three PSLQ codes and one tanh-sinh code, together with sample output files for comparison with your own runs.
- A README.txt file with detailed instructions for installation and coding.
- A full technical paper will be available shortly.

This talk is available at