# Moving SCA/LAPACK to Higher Precision

## without *too much* work

Yozo Hida

yozo@cs.berkeley.edu

Computer Science Division
EECS Department
U.C. Berkeley

January 30, 2007

# Outline

# Outline

# Current Status

- LAPACK 3.1 has been released.
  - Improved MRRR algorithm for symmetric eigenproblem.

    2006 SIAM SIAG LA Prize winning algorithm of Dhillon, Parlett
  - Faster Hessenberg QR (up to 10×).

    2003 SIAM SIAG LA Prize winning algorithm of

    Braman, Byers and Mathias.
  - Faster Hessenberg, tridiagonal, bidiagonal reductions.
  - Mixed precision iterative refinement.
  - Thread safety (`save` and `data` statements removed).
  - Many bug fixes.
- Feedback welcome: `http://www.netlib.org/lapack-dev/`

# Current Activities

- Faster algorithms.
  - Recursive blocked layouts.
  - Better QZ.
  - $O(n^2)$ companion matrix solver (polynomial roots).
- More accurate algorithms.
  - Extra-precise iterative refinement.
  - Jacobi SVD.
  - Higher precisions.
- Expanded contents: More LAPACK in SCALAPACK.
- Automated performance tuning.
- Improve ease of use.
- Community involvement (this means you!)

See `www.cs.berkeley.edu/~demmel` for details.

# Some Participants – more are welcome!

- UC Berkeley

  Jim Demmel, Ming Gu, W. Kahan, Beresford Parlett, Xiaoye Li, Osni Marques, Christof Vömel, David Bindel, Yozo Hida, Jason Riedy, Jianlin Xia, Jiang Zhu
- U. Tennessee, Knoxville

  Jack Dongarra, Victor Eijkhout, Julien Langou, Julie Langou, Piotr Luszczek, Stan Tomov
- Other Academic Institutions

  UT Austin, UC Davis, Florida IT, U Kansas, U Maryland, North Carolina SU, San Jose SU, UC Santa Barbara TU Berlin, FU Hagen, U Madrid, U Manchester, U Umeå, U Wuppertal, U Zagreb
- Research Institutions

  CERFACS, LBNL
- Industrial Partners

  Cray, HP, Intel, MathWorks, NAG, SGI

# What could go in LAPACK?

**for all** linear algebra problems **do**
  **for all** matrix types **do**
    **for all** data types **do**
      **for all** architectures and networks **do**
        **for all** programming interfaces **do**
          Produce fastest algorithm providing acceptable accuracy.
          Produce most accurate algorithm running at acceptable speed.

# What could go in LAPACK?

**for all** linear algebra problems **do**
**for all** matrix types **do**
**for all** data types **do**
**for all** architectures and networks **do**
**for all** programming interfaces **do**
Produce fastest algorithm providing acceptable accuracy.
Produce most accurate algorithm running at acceptable speed.

Some prioritization and automation is obviously needed.

Rest of the talk will concentrate on data types, specifically precisions higher than double precision.

# Motivation for higher precision

- LAPACK and SCALAPACK are widely used.
- User survey revealed small but significant portion of users wanted precision higher than double precision.
  `http://icl.cs.utk.edu/lapack-forum/survey/`
- We want to do least amount of work to support multiple precision.
- We want to support codes like

  n_bits $\leftarrow$ 32
  **repeat**
    n_bits $\leftarrow$ n_bits $\times$ 2
    Solve with n_bits
  **until** error $<$ tol

# Outline

# Some Implementations of Higher Precision

- Fixed precision
  - Compiler supported quad
  - Double-double, Quad-double
- Arbitrary precision
  - GMP / MPFR
  - ARPREC

# Double-double / Quad-double Package

- `http://crd.lbl.gov/~dhbailey/mpdist/`.
- These represents a higher precision numbers as an unevaluated sum of 2 or 4 double precision numbers.
- Example: $2^{60} + 1$ is represented as the pair $(2^{60}, 1)$.
- Can represent about 32 / 64 digits of precision,
- Highly efficient algorithms due to their fixed, small size.
- Simple representation: fixed size array of doubles.
  Makes it easy to allocate arrays and use MPI.
- Somewhat fuzzy definition of "machine precision" for these numbers:
  $1 + 2^{-1000}$ can be represented exactly in double-double, but not $1 + 2^{-500} + 2^{-1000}$.
- Exponent range limited to that of double precision.

# Double-double / Quad-double Package

- C, C++ and Fortran 95 interfaces.

```
subroutine f_main
use qdmodule
type (qd_real) a, b
a = 1.d0
b = cos(a)**2 + sin(a)**2 - 1.d0
call qdwrite(6, b)
end subroutine
```

- Support for complex data types recently added.

# GMP/MPFR

- Multiple-precision floating-point computations with correct rounding.
- `http://www.mpfr.org/`
- Uses integer arithmetic instructions.
- Highly optimized for variety of platforms.
- Somewhat complicated data structure, mixing various types in a C struct.
  This makes inter-language operation, porting, and message passing somewhat harder.
- C, C++ interfaces. No Fortran 95 interface.

# ARPREC

- `http://crd.lbl.gov/~dhbailey/mpdist/`.
- Uses simple floating point array to represent data.
  This makes inter-language operation and message passing easier.
- Slower than GMP. (sometimes by factor of 10).
- C, C++, and Fortran 95 interfaces.

```
subroutine f_main
use mpmodule
type (mp_real) a, b
call mpinit (500)
a = 1.d0
b = cos(a)**2 + sin(a)**2 - 1.d0
call mpwrite(6, b)
end subroutine
```

# Outline

# Things to Consider

- Single source code for varying precision. This makes it much easier to maintain code.
- Workspace allocation. We need to tell the user how much workspace to allocate.
- Temporary variable allocation.
- Should we allow multiple precisions in one routine? Within one matrix?
- How to adjust precision.
- Can multiple versions co-exist? Naming issues.

# Workspace Allocation

Many LAPACK routines requires workspaces.
How should we specify the size?

- by number of bytes? Doesn't work well in Fortran.
- by number of bignum slots? Works OK if every bignum in the workspace has the same size.

We also like the workspace to have some contiguity for cache friendliness and ease of message passing.

# Temporary Variable Allocation

Memory for temporary variables need to be allocated somewhere:

$$x = a + b + c$$

The temporary result $(a + b)$ must be stored somewhere.

- Have the compiler automatically allocate (for fixed precisions),
- Use external malloc routine,
- Explicitly allocate.

# Fixed vs. Variable Precision

- Memory allocation: how much? when? where?
- Variable precision allows us to support codes like

n_bits $\leftarrow$ 32
**repeat**
  n_bits $\leftarrow$ n_bits $\times$ 2
  Solve with n_bits
**until** error $<$ tol

# Fixed Precision

This is the easiest approach.

- Many compilers (not all) provide support for quad precision.
- One compiler (OMF77) even supports multi-precision variables (precision is compile-time selected).
- Fortran 90 modules and interfaces can be used to provide operator overloading to custom types (double-double, quad-double).
- Memory allocation issues can be handled automatically.

# Maximum Precision

- User specifies maximum precision at compile time.
- At run time, the program specifies the precision to be used (less than the maximum specified at compile time).
- Memory allocation issues can be handled automatically.
- Wastes memory if precision used is much smaller than the maximum precision.

Fortran 95 interface of ARPREC currently works in this mode.

# Variable Precision

- User can specify precision to use at run-time.
- Memory allocation issues can get tricky:
  - Memory must be allocated dynamically.
  - For cache (and communication) efficiency, we want to allocate a single block for each matrix.
  - Do we allow differing precisions within one algorithm? within one matrix?
- Only allocates necessary memory.

C++ interface of ARPREC currently work in this mode. Work is currently being done to make Fortran 95 interface works in this mode as well.

# Outline

# Modules

The high precision library provides a module that declares what operators and functions are overloaded:

```
module mpmodule
  type mp_real
    sequence
    real*8 mpr(mpwds+5)
  end type

  interface operator (+)
    module procedure mpadd
  end interface

...

end module mpmodule
```

# Modules

LAPACK source is then modified slightly:

```
subroutine some_lapack_routine
  !  Use module for arbitrary precision
  use mpmodule

  !  Declare variables in the desired format
  type (mp_real) a, b, c

  !  LAMCH must be provided by the
  !  arbitrary precision package provider
  smlnum = lamch(a, 'Safe minimum')

  !  ...the rest of the code ...
end subroutine
```

# Assessment of Constants

Simple replacement of variable type is not enough:

```
x = 0.1d0 / 0.3d0
```

This line is interpreted by the compiler as double precision constants and computation. Needs to be replaced with something like,

```
x = mp_real(0.1d0) / mp_real(0.3d0)
```

or if dealing with constants not representable in double precision:

```
x = mp_real("0.1") / mp_real("0.3")
```

This is the most common "bugs" reported to us when people convert their code to use our higher precision packages.

# Naming Issues

How should a routine be named when using a new precision?

- Add a new prefix. DGESVX becomes QGESVX for quad, QD_GESVX for quad-double etc.
  Leads to name explosion, but easier for inter-language operations.
- Use generic names like "GESVX" and dispatch to correct routines using Fortran 95 module and interface facilities.
  LAPACK 95 does this.

# Limitations with Fortran 95

- Fortran 95 does not allow I/O routines to be overridden. LAPACK uses I/O only in the test code, so often we can just print out the double-precision approximation instead.
- Fortran 95 does not have a general destructor, making it hard to micromanage memory allocation / destruction. (Fortran 2003 fixes this).
- Annoying Fortran 77 formatting (e.g., line length).

# Implementation

Currently implemented as a Perl script to convert LAPACK sources to perform

- use appropriate module file (provided by the dd / qd / arprec packages),
- constant literal substitutions,
- handle Fortran data declarations,
- use a new prefix for each LAPACK routines,
- substitute something appropriate for read / write statements, and
- declare variables in appropriate types.

Passes many LAPACK tests for Quad and QD precisions, including linear systems.
Work being done on ARPREC implementations.

# Outline

# Iterative Refinement

Use Newton's iteration on $r(x) = b - Ax$ but compute the residual in double the working precision:

$\hat{x} \leftarrow A^{-1}b$ (using basic solution method)

**repeat**

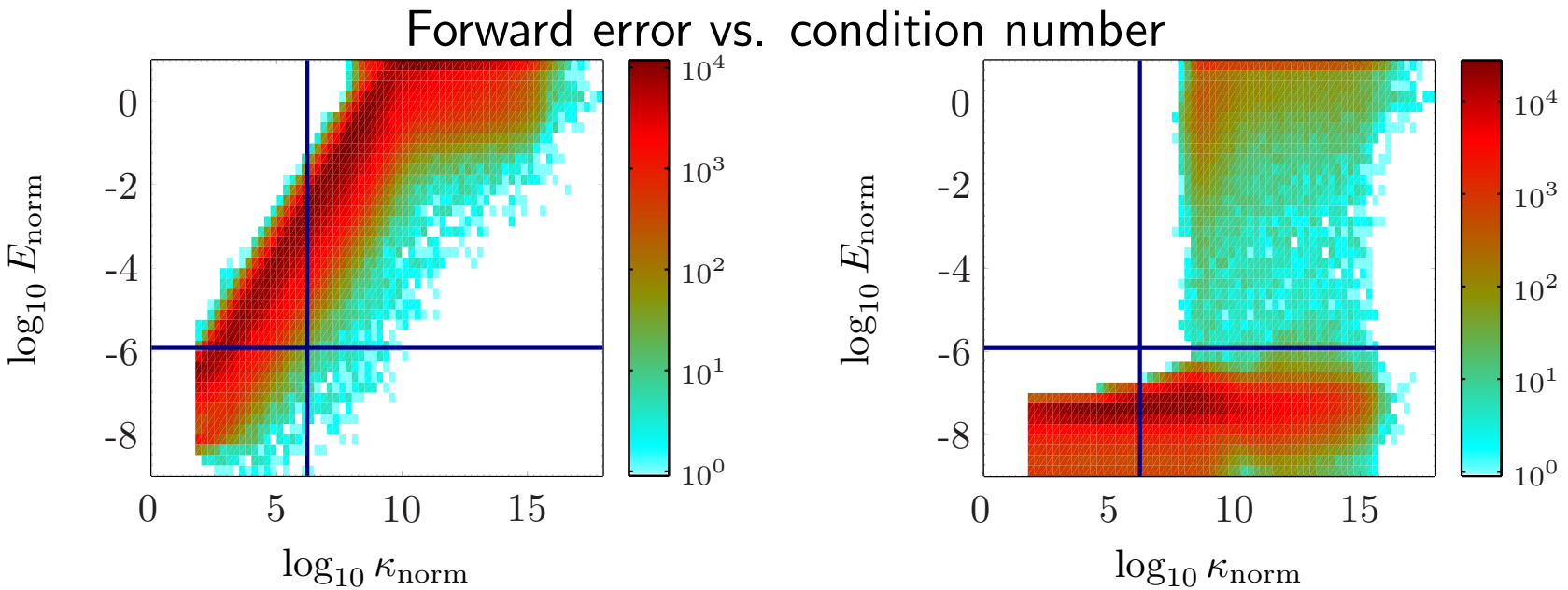
$r \leftarrow A\hat{x} - b$ (compute residual in doubled precision)

$dx \leftarrow A^{-1}r$ (solve for correction)

$x \leftarrow \hat{x} - dx$ (update solution)

**until** convergence **or** no progress

# Iterative Refinement




We can obtain small errors until condition number gets too bad; at which point we just need more precision.

# Accurate Summation

## Theorem

*Suppose we have $n$ $f$-digit, base-$b$ floating point numbers, and we sum them in decreasing order of their exponent using a $F$-digit accumulator. Then if $n \leq 1 + \frac{b^{F-f}}{1-b^{-f}} \equiv N$ then the computed sum $\hat{s}$ has a relative error bounded by*

$$\left| \frac{s - \hat{s}}{\hat{s}} \right| < \tfrac{1}{2} \left[ b + 1 + \frac{3b^{-f}}{1 - b^{1-f}} \right] b^{-f}$$

*For $b^f$ moderately large, relative error is bounded by just over $\frac{1}{2}(b+1)b^{-f}$. If $n \geq N + 2$, then relative error can be arbitrary.*

# Accurate Summation

If we use x86 80-bit floating point format as accumulator, we can add $2^{11} + 1 = 2049$ doubles accurately.

Previous theorem is especially useful when dealing with multi-precision variables, since

- sorting by exponent is often much faster, and
- we can often pick the size of the accumulator $(F)$ to achieve a desired accuracy, based on the number of terms to be added.

ARPREC uses $b = 2^{48}$ as the base, so we just need to keep an extra word to compute the sum accurately.

# Outline

# Conclusion

Once the high-precision library provides the appropriate module file declaring the overridden functions, most of LAPACK code can be transformed to use them without too much effort.

# Outline

# Future Work

- Exception handling.
- How much performance can we buy from using multi-core BLAS?
- Apply same technique to SCALAPACK.
- Provide F95 interface to MPFR. Anyone?

# Recent Results in Fast, Stable Matrix Computations

- Current record for fast matrix multiplication: $O(n^{2.38})$. (Coppersmith & Winograd, 1990).
- Strassen et al. showed $O(n^a)$ matmul implies $O(n^a)$ matrix inversion, determinant etc.
- New algorithm by Cohn, Kleinberg, Szegedy, Umans (2005)
  - Uses Wedderburn's Theorem to reduce matmul to FFT.
  - Equals Coppersmith & Winograd, beating it depends on finding right groups.
- New results (Demmel, Dumitriu, Holtz, Kleinberg, 2006)
  - Above algorithm is stable (normwise).
  - Any $O(n^a)$ algorithm can converted to a stable one (based on Raz, 2003)
  - There are stable algorithms for QR, LU, solve, determinant costing $O(n^a)$ or $O(n^{a+\epsilon})$ for any $\epsilon > 0$.