nag®

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Libraries II

David Quigley

LAPACK and Fast Fourier Transforms

Libraries II

D. Quigley

Centre for Scientific Computing
University of Warwick

Plan

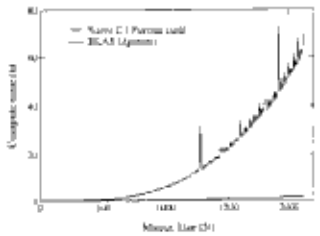
LAPACK functionality and usage

LAPACK interface in C and Fortran

Fast Fourier Transforms (FFTs)

But first.....

In yesterday's practical we measured the performance of BLAS vs naive code for matrix multiplication on our IBM cluster



BLAS brings speedups measured in orders of magnitude for large matrices

LAPACK

Introduction

Linear Algebra PACKage – written in Fortran.

Developed from LINPACK and EISPACK.

Uses the BLAS extensively for speed and efficiency.

LAPACK is usually included within vendor-provided maths libraries (MKL, ESSL, ACML – i.e. wherever you find BLAS)

- all the information you'll every need

Functionality

- Solve systems of linear equations
- Solve linear least squares problems
- Eigenvalue and singular value problems

Dense and banded matrices are catered for in real and complex flavours

General sparse matrices are the domain of other libraries, see libraries lectures IV and V

LAPACK Routines

LAPACK Routines can be classified as either:

Driver routines: These solve a particular computational problem, e.g. computing eigenvalues or solving a system of linear equations

Computational routines: Perform a distinct computational task, e.g. factorise a matrix, reduce a matrix to tridiagonal form.

Auxiliary routines: Include extensions to the BLAS, routines for common subtasks, various low level computations.

Each driver routine calls a sequence of computational routines to implement a particular algorithm

Routine names

Similar naming convention applies as BLAS.

Routine name = XYYZZ or XYYZZZ

X	YY	ZZ or ZZZ
Specifies precision	Specifies matrix type, e.g.	Computation, e.g.
S = single D = double C = complex Z = double complex	GE = general matrix HB = Hermitian banded BD = bidiagonal TP = triangular, packed storage SY = symmetric	BRD = bidiagonal reduction TRF = LU factorisation SV = solve linear system EV = solve eigensystem LQF = LQ factorisation

Again, too many to list. Consult the manual.

LAPACK arguments

An example.

```
call [illegible]
```

Find eigenvalues and eigenvectors of a real symmetric NxN matrix A
Eigenvalues are stored in the vector w, eigenvectors overwrite A

What are all the extra arguments?

We've already met lda . Leading dimension of A.

Special Matrices

Symmetric / Hermitian / Triangular matrices can be stored as "packed" to save memory and computation

$$\begin{pmatrix} 1 & 4 & 7 \\ 0 & 5 & 8 \\ 0 & 0 & 9 \end{pmatrix}$$

e.g. upper triangular

Note column-major ordering

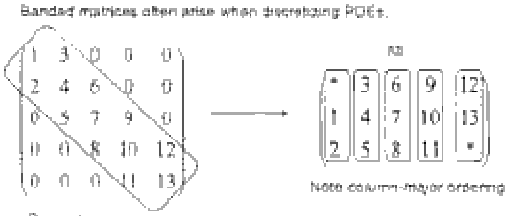
Extra argument uplo specifies which half we are storing 'U' or 'L'

Computer memory ⟶

	1	4	5	7	8	9						

Special Matrices

Banded matrices often arise when discretising PDEs.



Note column-major ordering

Computer memory ⟶

*	1	2	3	4	5	6	7	8	9	10	11	12	13	*

Special Matrices

Extra arguments are again required.

```
call dgbmv(trans,m,n,kl,ku,alpha,AB,ldab,x,incx,beta,y,incy)
```

m and n are dimensions of original A
kl is the number of non-zero subdiagonals
ku is the number of non-zero ~~sub~~diagonals super

ldab is the leading dimension of AB, NOT A

$$\begin{pmatrix} * & 3 & 6 & 9 & 12 \\ 1 & 4 & 7 & 10 & 13 \\ 2 & 5 & 8 & 11 & * \end{pmatrix}$$

AB

The wasted elements in the top left and bottom right triangles are never referenced

Extra reduction in storage costs for symmetric banded matrices

Special Matrices

Packed and banded storage are also used in BLAS and LAPACK

LAPACK allows for an additional storage scheme.

Tridiagonal matrices can be specified as 3 (2 if symmetric) 1D arrays.

Similarly bidiagonal matrices can be specified in this way.

Specify a job....

An example

```
call dsyevd(jobz,uplo,N,A,lda,w,work,lwork,iwork,liwork,info)
```

If we want eigenvalues only : `jobz = 'N'`
If we want eigenvectors also : `jobz = 'V'`

This is just a particular example. the documentation for each routine will detail any options which must be specified by character arguments.

Info

BLAS will blindly perform operations on whatever we give it.
A lot more can go wrong in LAPACK – are the inputs valid?

```
call dsyevd(jobz,uplo,N,A,lda,w,work,lwork,iwork,liwork,info)
if (info/=0) then
    write(0 ,'("Error: on dsyevd returned info = ",I3)')info
end if
```

info = 0	Routine completed as normal
info < 0	Argument number -1 x info is invalid
info > 0	Failure in course of computation

Note that info = 0 does not imply your answer is correct.
Garbage in = Garbage out, even if your garbage is a valid input!

Work and more work

LAPACK will often need to use temporary work arrays.
We need to provide this storage, this avoids any nasty surprises!

From netlib.org (dsyevd documentation):

```
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.
*          If N <= 1,                (work must be at least 1.
*          If JOBZ = 'N' and N > 1, LWORK must be at least 2*N+1.
*          If JOBZ = 'V' and N > 1, LWORK must be at least 1 + 6*N + 2*N**2.
```

+ similar for the LIWORK, the size of the integer work array

Optimal work array size

Q. I' m genetically predisposed to ignore user manuals. How can I safely choose the size of my LAPACK work arrays?
A. You can query the optimal size within your code.

```
allocate(work(1),iwork(1),stat=ierr)
if (ierr/=0) stop "Error allocating work arrays"

! Query optimal size of work array
call dsyevd(jobz,uplo,N,A,lda,w,work,-1,iwork,-1,info)
lwork  = work(1)     ! Optimal size stored in first element
liwork = iwork(1)    ! of the work and iwork arrays

deallocate(work,iwork)
allocate(work(1:lwork),iwork(1:liwork),stat=ierr)
if (ierr/=0) stop 'Error allocating work arrays'

call dsyevd(jobz,uplo,N,A,lda,w,work,lwork,iwork,liwork,info)
```


I see no C

- C programmers will have spotted that everything we've seen so far on LAPACK is in FORTRAN
- Unlike BLAS – there is no widely accepted standard for calling LAPACK routines from C
- Many wrappers exist for specific purposes
- This is why we have to worry about array storage in C vs Fortran, how arrays are passed behind the scenes

You may encounter CLAPACK – this is just the LAPACK source translated to C with f2c, and "tidied up a bit". It' s primarily intended as a means for people without a Fortran compiler to use LAPACK.

LAPACKe

- Since LAPACK 3.3 (October 2010) the LAPACK web pages have included LAPACKe as a separate download.
- This is a free C interface to (most) LAPACK routines, based on that used in the Intel Math Kernel Library (MKL) version 10.3 onwards
- Yet to be widely adopted in other packages that include LAPACK, e.g. ACML/NAG, and probably won't be anytime soon
- Unlike CBLAS (included in GSL), LAPACKe hasn't yet filtered its way into the popular Linux distros.
- MKL is currently the only source of (limited) documentation

Calling LAPACK from C

```
dsyevd_(&jobz,&uplo,&n,A,&lda,w,work,&lwork,iwork,&liwork,&info);
```

- Who compiled your LAPACK library? Did their compiler append underscores to subroutine/function names? Use nm to find out.
- Again, we assume that Fortran is passing by reference behind the scenes (A, w, work and iwork are assumed to be pointers).
- We must supply A in column-major order!
- work and iwork must be allocated arrays of length lwork and liwork respectively

LAPACKe example

```
lapack_complex_double *lpcx_A = (lapack_complex_double *)A;
info = LAPACKE_dsyevd(LAPACK_COL_MAJOR,job,uplo,N,lpcx_A,lda,w);
```

- First argument specifies array storage, either LAPACK_COL_MAJOR or LAPACK_ROW_MAJOR.
- job, uplo, n now passed by value
- A and w are pointers to allocated storage.
- Note that we don't need work arrays!
- Complex variables must be cast into a format which LAPACKe understands, which is configurable at build time

MPACK

```
dsyevd_f77(&jobz,&uplo,&N,A,&lda,w,work,&lwork,iwork,&liwork,&info);
```

- MPACK is a package for performing linear algebra to arbitrary numerical precision. Usefully, it contains header files blas.h and lapack.h which define prototypes for calling Fortran
- Everything is passed by reference, and Fortran routine names have _f77 appended
- This is the approach used in Tuesday's practical
- **WARNING** – these headers assume certain compiler-dependent behaviour for complex function return values (works only with gcc)

Worked Example

Lets look at what we just saw in the maths lecture.

Let's solve , $Ax = b$ for the following.

```
// Matrix A in column major format
double A[] = { [illegible],   // 1st column
               [illegible],   // 2nd column
               [illegible],   // 3rd column
               [illegible],   // 4th column
               [illegible] }; // 5th column
```

We'll treat this as a double precision ("d") general ("ge") matrix. First we should think about storage

Storage in C

As discussed yesterday, we must be very careful how we store matrices in C, i.e. row versus column major ordering and contiguity of storage

For the purposes of this example we will work with 1D arrays explicitly as on the previous slide, stored in column major order.

There are various ways we can ease converting from row and column indices to 1D indices, for example:

```
int [illegible] (int row,int col,int lda) {
  return col*lda+row;  // swap row and col if row-major
}
```

Storage in C

Usage:

```
for (j=0;j<5;j++) {                    // loop over columns
    for (i=0;i<5;i++) {                // loop over rows
      printf("%5.2f ",A[i+j*5]);       // print A_ij
    }
    printf("\n");                      // new row
  }
}
```

Produces:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Next we compute the partially pivoted LU factorisation ("trf")

Worked Example

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Original A

```
// Factorise using dgetrf
dgetrf_(&m,&n,A,&ldA,ipiv,&info);
```

n = m = ldA = 5

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	-1.20	-3.19	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	-3.40
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

A on exit contains L and U (unit diagonals of L not stored)

Worked Example

Check – split L & U and multiply together.

L:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

x

U:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

=

LU:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

$= P^{-1}A \neq A$

Worked Example

ipiv contains information required to form P.

[illegible]	Row 1 was interchanged with row [illegible]
[illegible]	Row 2 was interchanged with row [illegible]
[illegible]	Row 3 was interchanged with row [illegible]
[illegible]	Row 4 was interchanged with row [illegible]
[illegible]	Row 5 was interchanged with row [illegible]

Now we have $A = PLU$ we can solve for any number of vectors b. For example,

```
double b[] = { [illegible] };
```

Next task is to perform the substitution and get the solution ("trs").

Worked Example

```
// Get the solution
dgetrs_(&trans,&n,&nrhs,A,&ldA,ipiv,b,&ldb,&info);
```

trans can be 'n', 't' or 'c' as before

nrhs is the number of b vectors to compute solution for.

b is a matrix to store the nrhs b vectors (as columns)

A and ipiv are those output from dgetrf, passing the original A will produce nonsense

On exit from dgetrs, b contains the solution x:

[illegible]
[illegible]
[illegible]
[illegible]
[illegible]

So are we done?

- No way! We may have a code which gives numbers, and we may trust the authors of the library, but can we be sure we've used it properly?
- VALIDATE your usage of the library, e.g. restore original A and compute Ax and check against original value of b

```
#ifdef DEBUG

double alpha = 1.0 , beta = 0.0 ; int one = 1
double check[n];
// x is currently stored in b
dgemv_(&trans,&m,&n,&alpha,A,&ldA,b,&one,&beta,check,&one);
for (i=0;i<n;i++) {printf("%5.2f ",check[i]);} printf("\n");

#endif /* DEBUG */
```

Good enough?

- Maybe - we might not be happy with the quality of our solution
- We could apply iterative refinement (`dgerfs`) – last lecture, requires passing the original A plus the LU factorisation (storage implications)
- We might want to estimate the condition number of A (see later) and equilibrate if poorly scaled
- There are also routines to compute forward and backward error bounds (See Sven Hammarling's lecture tomorrow)

Driver routines

- We used two computational routines to get our solution
- We could have used a single driver routine to do the same thing

```
dgesv_( &n, &nrhs, A, &lda, ipiv, B, &ldb, &info)
```

This is the "simple" version of the relevant linear systems driver routine

The "expert" version also invokes the condition / error analysis computational routines.

The computational routines are instructive, but production codes would normally use the driver routines unless there was a reason not to

Other systems / matrix types

- We've focussed on the mechanics of using LAPACK here by looking at a familiar example
- We've only touched on the functionality available
- As ever, the user manual and Google will help

Fast Fourier Transforms

A reminder

- The one-dimensional Fourier transform

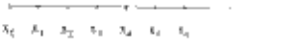
$$F(k) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx \qquad \text{Forward transform}$$

$$f(x) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{\infty} F(k) e^{ikx} dk \qquad \text{Backward transform}$$

- Fourier transforms are everywhere: signal processing, image analysis, solid state physics and just about anywhere else!
- Here we are interested in the mechanics of performing the FT
- We will look at a specific example in molecular simulation tomorrow

Discrete Fourier Transform

- We need to work on a discrete grid of N points in a computer.
- Domain of length L in real space: grid spacing is $\delta = L/(N-1)$



$x_n = n\delta \qquad f_n = f(x_n)$

- Grid in reciprocal space of length $2\pi/L$, spacing $2\pi/L(N-1) = 2\pi\delta/L^2$

$k_n = n\,2\pi\delta/L^2 \qquad F_n = F(k_n)$

- Applying the DFT implies this domain is one period sampled from a periodic function (this is true in the workshop tomorrow). Otherwise may need to think about windowing functions and associated artefacts

Discrete Fourier Transform

- For discrete data, the integrals become sums over grid points
- Adopting a few conventions

$$F_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f_n e^{-2\pi i n k / N}$$

k is now an integer running over the reciprocal space grid points

$$f_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} F_k e^{2\pi i n k / N}$$

n runs over the real space grid

f_n and F_k are the values of the function and its transform at the grid points

N.B. This makes f_0 the value of the function at $x = 0$, this is important to remember when interpreting transforms of symmetric functions. Most implementations drop normalising prefactor.

The Fast Fourier Transform

- Computing the DFT requires computing N complex exponentials for each of N data points. It should therefore require $O(N^2)$ work.
- FFTs are a class of algorithm which reduce this to $O(N \log N)$
- The simplest example is the original Cooley-Tukey (1965) algorithm

The DFT at each point k can be expressed as sums over two interleaved sub-grids of odd / even grid points of length $M=N/2$.

$$F_k = \sum_{n=0}^{M-1} f_{2n} e^{-2\pi i (2n) k / N} + \sum_{n=0}^{M-1} f_{2n+1} e^{-2\pi i (2n+1) k / N}$$

The Fast Fourier Transform

We recognise this as a sum of length $M=N/2$, plus the product of a second sum of length M and a "twiddle factor"

$$F_k = \sum_{n=0}^{M-1} f_{2n} e^{-2\pi i n k / M} + e^{-2\pi i k / N} \sum_{n=0}^{M-1} f_{2n+1} e^{-2\pi i n k / M}$$

$$F_k = E_k + e^{-2\pi i k / N} O_k$$

E_k and O_k are sums over terms which are periodic in k, but with a period of M rather than N – so we only need calculate them for $k<M$ to compute F_k at all N grid points

Need to evaluate $(N+1)$ complex exponentials $N/2$ times i.e. $(N^2+N)/2$

Lather, Rinse, Repeat

For each $k<M$ we have effectively performed two DFTs of length $N/2$, plus one DFT of length 2

This is too good a trick to use only once, so lets further subdivide the each DFT of length M into DFTs of length $M/2 = N/4$

This becomes four DFTs of length $N/4$ plus one DFT of length 4 for each $k<N/4$. We now need evaluate only $(N^2+3N/4)$ complex exponentials

We can keep recursively applying this trick until we run out of points to divide up. The eventual cost is $O(N \log N)$

The choice of two as the radix restricts us to using $N=2^p$ grid points where p = integer

Generalisations and Libraries

- Generalisations to use various small prime factors as the radix exist, and allow for less restrictive choice of grid size
- Further efficiency gains can be made if the input data is known to be real, with some slight complications in usage
- Hardware tuned implementations of FFT algorithms exist in most vendor-supplied maths libraries, e.g. MKL, ACML, ESSL etc.
- Popular (non vendor specific) FFT libraries include FFTW (Fastest Fourier Transform in the West) and the Temperton GPFA (Generalised Prime Factor Algorithm)
- We will cover the interface to these libraries in tomorrow's workshop.