A few words about **gcov**. This command tells us:

- how often each line of code executes
- what lines of code are actually executed

Compile without optimization. It works only with **gcc**. So it should work with **g95** and **gfortran** as well. There may, however, be problems with different versions of **gcc** and the **gcc**-libraries. See the web-page for the assignment for the latest details.

To use **gcov** on the student system (not Intel in this case) one should be able to type:

```
   g95  -fprofile-arcs -ftest-coverage prog.f90 sub.f90
   ./a.out

   gcov prog.f90  creates prog.f90.gcov
   gcov sub.f90   creates sub.f90.gcov

   less prog.f90.gcov   etc.
```

and for C

```
      gcc  -fprofile-arcs -ftest-coverage prog.c sub.c
```

similarly for **gfortran** and **g++**.

Example: Arpack, a package for solving large and sparse eigenvalue problems, $Ax = \lambda x$ and $Ax = \lambda Bx$. I fetched a compressed tar-file, unpacked, read the README-file, edited the configuration file, and compiled using make. After having corrected a few Makefiles everything worked. I then recompiled using the compiler options for **gprof** and **tcov** (on a Sun; I have not run this one the AMD-system).

I used the **f90**-compiler even though Arpack is written in Fortran77. (There is also Arpack++, a collection of classes that offers C++ programmers an interface to Arpack.)

First **gprof**:

```
% gprof | less   (1662 lines, less is a pager)
or
% gprof | more   (or m with    alias m more)
                 (I have       alias m less)
or
% gprof > file_name  (emacs file_name, for example)
etc.
```

The first part of the output is the flat profile, such a profile can be produced by **prof** as well. Part of it, in compressed form, comes on the next page. The flat profile may give a sufficient amount of information.

```
Each sample counts as 0.01 seconds.
 %   cumulative   self           self    total
time   seconds  seconds  calls  s/call  s/call  name
79.10    8.10     8.10    322    0.03    0.03  dgemv_
 8.50    8.97     0.87     60    0.01    0.01  dger_
 4.10    9.39     0.42     58    0.01    0.01  dgttrs_
 3.22    9.72     0.33    519    0.00    0.00  dcopy_
 2.25    9.95     0.23    215    0.00    0.00  dnrm2_
 0.49   10.00     0.05    562    0.00    0.00  __open
        ...  lots of lines deleted  ...
 0.00   10.24     0.00      1    0.00   10.14  main
        ...  lots of lines deleted  ...
 0.00   10.24     0.00      1    0.00    0.00  strchr
```

**name** is the name of the routine (not the source file). The Sun-compiler converts the routine name to lower case and adds _ . **___open** is a system (compiler?) routine.

The columns are:
**% time** the percentage of the total running time of the program used by this function. Not the one it calls, look at **main**.

**cumulative seconds** a running sum of the number of seconds accounted for by this function and those listed above it.

**self seconds** the number of seconds accounted for by this function alone. This is the major sort for this listing.

**calls** the number of times this function was invoked, if this function is profiled, else blank.

**self ms/call** the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

**total ms/call** the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank. Note **main**.

**dgemv** is a BLAS routine, double general matrix vector multiply:

```
 dgemv - perform one of the matrix-vector operations
 y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y
```

I have compiled the Fortran code instead of using a faster performance library so we can look at the source code.

Let us run **tcov** on **dgemv**.

```
Part of the output (compressed):
                   ...
   168 ->       DO 60, J = 1, N
  4782 ->          IF( X(JX).NE.ZERO )THEN
  4740 ->             TEMP = ALPHA*X(JX)
                      DO 50, I = 1, M
77660160 ->               Y(I) = Y(I) + TEMP*A(I, J)
          50          CONTINUE
                   END IF
  4782 ->          JX = JX + INCX
          60    CONTINUE

                Top 10 Blocks

                Line      Count

                 211   77660160
                 238   50519992
                 177     871645
                   ...
```

Note that this code is very poor. Never use the simple Fortran BLAS- or Lapack routines supplied with some packages. One lab deals with this issue.

## More about gprof

**gprof** produces a call graph as well. It shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines called by each function. This list is edited.

```
index %time self  children  called     name
                                      ...  rm lines
-----------------------------------------------
           0.01   10.13     1/1      main [1]
[3]   99.0  0.01   10.13     1      MAIN_ [3]
           0.00    7.19    59/59     dsaupd_ [5]
           0.00    2.45     1/1      dseupd_ [8]
           0.42    0.00    58/58     dgttrs_ [14]
 ... lines deleted
-----------------------------------------------
           0.83    0.00    33/322    dsapps_ [11]
           1.48    0.00    59/322    dlarf_ [9]
           5.79    0.00   230/322    dsaitr_ [7]
[4]   79.1  8.10    0.00   322      dgemv_ [4]
           0.00    0.00  1120/3179   lsame_ [50]
-----------------------------------------------
```

Each routine has an index (see table at the end) and is presented between `---`lines. 8.10s was spent in **dgemv** itself, 79.1% of total (including calls from **dgemv**). **dsapps**, **dlarf**, **dsaitr** (parents) called **dgemv** which in turn called **lsame**, a child. **dsapps** made 33 out of 322 calls and **dgemv** took 0.83s for the calls. **dgemv** called **lsame** 1120 of 3179 times, which took no measurable time (**self**).

**children**: For **dgemv** it is the total amount of time spent in all its children (**lsame**). For a parent it is the amount of that time that was propagated, from the function's children (**lsame**), into this parent. For a child it is the amount of time that was propagated from the child's children to **dgemv**.

## Profiling in Matlab

Matlab has a built-in profiling tool. **help profile** for more details. Start Matlab (must use the GUI).

```
>> profile on
>> run        % The assignment
Elapsed time is 1.337707 seconds.
Elapsed time is 13.534952 seconds.
>> profile report          % in mozilla or netscape
>> profile off
```

You can start the profiler using the GUI as well
(click in "Profiler" using "Desktop" under the main meny). The output comes in a new window and contains what looks like the flat profile from **gprof**.

One can see the details in individual routines by clicking on the routine under **Function Name** This produces a **gcov**-type of listing. It contains the number of times a line was executed and the time it took.

## Using Lapack from Fortran and C

Use Lapack to solve a problem like:

$$\begin{bmatrix} 1 & -1 & -2 & -3 & -4 \\ 1 & 1 & -1 & -2 & -3 \\ 2 & 1 & 1 & -1 & -2 \\ 3 & 2 & 1 & 1 & -1 \\ 4 & 3 & 2 & 1 & 1 \end{bmatrix} x = \begin{bmatrix} -9 \\ -4 \\ 1 \\ 6 \\ 11 \end{bmatrix}$$

The solution is the vector of ones. We use the Lapack-routine **dgesv** from Lapack. Here is a man-page:

```
NAME
DGESV  - compute the solution to a real system of
         linear equations A * X = B,

SYNOPSIS
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
INTEGER       INFO, LDA, LDB, N, NRHS
INTEGER       IPIV( * )
DOUBLE        PRECISION A( LDA,* ), B( LDB, * )

PURPOSE
DGESV computes the solution to a real system of linear
equations A * X = B, where A is an N-by-N matrix and X
and B are N-by-NRHS matrices.
The LU decomposition with partial pivoting and row
interchanges is used to factor A as  A = P* L * U,
where P is a permutation matrix, L is unit lower
triangular, and U is upper triangular.  The factored
form of A is then used to solve the system of equations
A * X = B.

ARGUMENTS
N       (input) INTEGER
        The number of linear equations, i.e., the order
        of the matrix A.  N >= 0.
```

```
NRHS    (input) INTEGER
        The number of right hand sides, i.e., the numbe
        of columns of the matrix B.  NRHS >= 0.

A       (input/output) DOUBLE PRECISION array, dimensio
        (LDA,N) On entry, the N-by-N coefficient matrix
        A.  On exit, the factors L and U from the
        factorization A = P*L*U; the unit diagonal
        elements of L are not stored.

LDA     (input) INTEGER
        The leading dimension of the array A.
        LDA >= max(1,N).

IPIV    (output) INTEGER array, dimension (N)
        The pivot indices that define the permutation
        matrix P; row i of the matrix was interchanged
        with row IPIV(i).

B       (input/output) DOUBLE PRECISION array, dimensio
        (LDB,NRHS) On  entry, the N-by-NRHS matrix of
        right hand side matrix B.  On exit, if INFO = 0
        the N-by-NRHS solution matrix X.

LDB     (input) INTEGER
        The leading dimension of the array B.
        LDB >= max(1,N).

INFO    (output) INTEGER
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an
              illegal value
        > 0:  if INFO = i, U(i,i) is exactly zero.  The
              factorization has been completed, but the
              factor U is exactly singular, so the
              solution could not be computed.
```

In Fortran90, but using the F77 interface, and F77-type declarations (to get shorter lines) this may look like:

```
program main
  integer, parameter ::  n   = 10, lda = n, &
                        ldb = n, nrhs = 1
  integer            ::  info, row, col, ipiv(n)
  double precision   ::  A(lda, n), b(ldb)

  do col = 1, n
    do row = 1, n
      A(row, col) = row - col
    end do
    A(col, col) = 1.0d0
    b(col)      = 1 + (n * (2 * col - n - 1)) / 2
  end do

  call dgesv ( n, nrhs, A, lda, ipiv, b, ldb, info )

  if ( info == 0 ) then
    print*, "Maximum error = ", maxval(abs(b - 1.0d0))
  else
    print*, "Error in dgesv: info = ", info
  end if

end program main

% Compile and link, somehow, to Lapack
% a.out
 Maximum error =  4.218847493575595E-15
```

Where can we find **dgesv**? There are several options. Fetching the Fortran-code from Netlib, using a compiled (optimized) library etc. One of the assignments, Lapack (Uniprocessor optimization), deals with these questions.

The following optimized libraries contain Lapack and BLAS (and perhaps routines for fft, sparse linear algebra, etc. as well).

- AMD: ACML (AMD Core Math Library).
- Intel: MKL (Intel Math Kernel library).
- SGI: complib.sgimath (Scientific and Mathematical Library).
- IBM: ESSL (Engineering and Scientific Subroutine Library).
- Sun: Sunperf (Sun Performance Library).

There may be parallel versions.

Now for C and C++

Fairly portable (do not use local extensions of the compiler). Think about: In C/C++

- matrices are stored by row (not by column as in Fortran)
- matrices are indexed from zero
- call by reference for arrays, call by value for scalars
- the Fortran compiler MAY add an underline to the name
- you may have to link with Fortran libraries (mixing C and Fortran I/O may cause problems, for example)
- C++ requires an **extern**-declaration, in C you do not have to supply it (but do)
- make sure that C and Fortran types are compatible (number of bytes)
- some systems have C-versions of Lapack

In the example below I have linked with the Fortran-version since not all systems have C-interfaces. Make sure not to call **dgesv** from C on the Sun, if you want the Fortran-version (**dgesv** gives you the C-version).

```
#include <math.h>
#include <stdio.h>
#define  _N   10

#ifdef __cplusplus
extern "C" void                 /* For C++ */
#else
extern void                     /* For C   */
#endif
   dgesv_(int *, int *, double *, int *, int[],
          double[], int *, int *);
/*
 * int [] or int *. double [][] is NOT OK but
 * double [][10] is, provided we
 * call dgesv_ with A and not &A[0][0].
 */

int main()
{
  int      n = _N, lda = _N, ldb = _N, nrhs = 1,
           info, row, col, ipiv[_N];
  double   A[_N][_N], b[_N], s, max_err;

  /* Make sure you have the correct mix of types.*/
  printf("sizeof(int) = %d\n", sizeof(int));

  /* Indexing from zero. */
  for (col = 0; col < n; col++) {
    for (row = 0; row < n; row++)
      A[col][row] = row - col;  /* Note TRANSPOSE */
    b[col] = 1 + (n * (1 + 2 * col - n)) / 2;
    A[col][col] = 1;
  }
```

```
  /* Note underline and & for the scalar types.
   * &A[0][0] not to get a
   * conflict with the prototype.
   */
  dgesv_(&n, &nrhs, &A[0][0], &lda, ipiv, b,
         &ldb, &info);

  if (info) {
    printf("Error in dgesv: info = %d\n", info);
    return 1;
  } else {
    max_err = 0.0;
    for (row = 0; row < n; row++) {
      s = fabs(b[row] - 1.0);
      if (s > max_err)
        max_err = s;
    }
    printf("Maximum error = %e\n", max_err);
    return 0;
  }
}

On a Sun. See the lab for AMD.

% cc -fast extern.c -xlic_lib=sunperf
% a.out
sizeof(int) = 4
Maximum error = 4.218847e-15

% CC -fast extern.c -xlic_lib=sunperf
% a.out
sizeof(int) = 4
Maximum error = 4.218847e-15

% a.out    If you call dgesv and not dgesv_
sizeof(int) = 4
** On entry to DGESV , parameter number 1 has an
   illegal value.  Error in dgesv: info = -1
```

## Java

It is possible to mix Java with other languages using JNI, the Java Native Interface. Wikipedia is a good starting point (look for jni).

Here are a few words on Java.

```
% cat test.java
public class test {
  public static void main (String[] args) {
    int n = 10;
    double[] a = new double[n];

    for(int j = 0; j < n; j++)
      a[j] = j;

    System.out.println("a[n-1] = " + a[n-1]);
  }
}

% javac test.java       Produces test.class
% java test             Execute (can optimize, later...
a[n-1] = 9.0
```

`javac` produces the file `test.class` containing the bytecode.

`java` is the Java interpreter that reads and executes `test.class`. We can study the bytecode (instructions) using `javap`, the Java class file disassembler. The interpreter uses a stack and has local variables; I have called them `var_1` etc. To make the bytecode easier to read I have used our variable names. Half of the listing (mostly dealing with the print, has been deleted). I have not printed all the pops from the stack.

See Wikipedia (java bytecode) for more details.

```
% javap -verbose test

  ...
public static void main(java.lang.String[]);
Code:
  0: bipush  10       10    -> stack
  2: istore_1         stack -> var_1
  3: iload_1          var_1 -> stack
  4: newarray double  create double a[10], &a[0]->stack
  6: astore_2         &a[0] -> var_2
  7: iconst_0         0 -> stack
  8: istore_3         0 -> var_3 (corresponds to j)

  9: iload_3          j -> stack
 10: iload_1          n -> stack
 11: if_icmpge  25    if ( j >= n ) goto line 11+25
 14: aload_2          &a[0] -> stack
 15: iload_3          j -> stack
 16: iload_3          j -> stack  (used as index)
 17: i2d              double(j) -> stack
 18: dastore          a[j] = double(j), "index reg"
 19: iinc3, 1         j++
 22: goto9            goto line 9:
...
 54:return
}
```

To speed things up the bytecode interpreter (`java`) often uses a JIT (Just In Time) technique. A JIT compiler converts all of the bytecode into native machine code just as a Java program is run. This results in run-time speed improvements over code that is interpreted by a Java virtual machine.

`java -client test` or
`java -server test` (usually much faster; default).

One can profile Java programs. from `man java`:

```
-Xprof
  Profiles the running program, and sends profiling
  data to standard output. This option is provided
  as a utility that is useful in program development
  and is not intended to be be used in production
  systems.

-Xrunhprof[:help][:suboption=value,...]
  Enables cpu, heap, or monitor profiling. This
  option is typically followed by a list of
  comma-separated suboption=value pairs. Run the
  command   java -Xrunhprof:help   to obtain a
  list of suboptions and their default values.
```

## Interfacing Matlab with C

It is not uncommon that we have a program written in C (or Fortran) and need to communicate between the program and Matlab.

The simplest (but not the most efficient) way the fix the communication is to use ordinary text files. This is portable and cannot go wrong (in any major way). The drawback is that it may be a bit slow and that we have to convert between the internal binary format and text format. We can execute programs by using the `unix`-command (or `!` or `system`).

One can do more, however:

- Reading and writing binary MAT-files from C
- Calling Matlab as a function (Matlab engine)
- Calling a C- or Fortran-function from Matlab (using MEX-files, compiled and dynamically linked C- or Fortran-routines)

In the next few pages comes a short example on how to use MEX-files.

### MEX-files

Let us write a C-program that can be called as a Matlab-function. The MEX-routine will call a band solver, written in Fortran, from Lapack for solving an Ax=b-problem. The routine uses a Cholesky decomposition, where A is a banded, symmetric and positive definite matrix.

b contains the right hand side(s) and x the solution(s).
I fetched the routines from `www.netlib.org`

Matlab has support for solving unsymmetric banded systems, but has no special routines for the positive definite case.

We would call the function by typing:

```
>> [x, info] = bandsolve(A, b);
```

where **A** stores the matrix in compact form. **info** returns some status information (**A** not positive definite, for example).

**bandsolve** can be an m-file, calling a MEX-file. Another alternative is to let **bandsolve** be the MEX-file. The first alternative is suitable when we need to prepare the call to the MEX-file or clean up after the call.

The first alternative may look like this:

```
function [x, info] = bandsolve(A, b)
A_tmp = A;  % copy A
b_tmp = b;  % copy b
% Call the MEX-routine
[x, info] = bandsolve_mex(A_tmp, b_tmp);
```

I have chosen to make copies of A and b. The reason is that the Lapack-routine replaces A with the Cholesky factorization and b by the solution. This is not what we expect when we program in Matlab. If we have really big matrices, and if we do not need A and b afterwards we can skip the copy (although the Matlab-documentation says that it "may produce undesired side effects").

I will show the code for the second case where we call the MEX-file directly. Note that we use the file name, **bandsolve**, when invoking the function. There should always be a **mexFunction** in the file, which is the entry point. This is similar to a C-program, there is always a **main**-routine.

It is possible to write MEX-files in Fortran, but is more natural to use C.

First some details about how to store the matrix (for the band solver). Here an example where we store the lower triangle. The dimension is six and the number of sub- (and super-) diagonals is two.

```
a11  a22  a33  a44  a55  a66
a21  a32  a43  a54  a65   *
a31  a42  a53  a64   *    *

Array elements marked * are not used by the routine.
```

The Fortran-routine, **dpbsv**, is called the following way:

```
call dpbsv( uplo, n, kd, nB, A, lda, B, ldb, info )
```

where

```
uplo = 'U':  Upper triangle of A is stored
       'L':  Lower triangle of A is stored
```

We will assume that **uplo = 'L'** from now on

```
n    = the dimension of A
kd   = number of sub-diagonals
nB   = number of right hand sides (in B)
A    = packed form of A
lda  = leading dimension of A
B    = contains the right hand side(s)
ldb  = leading dimension of B
info = 0, successful exit
     < 0, if info = -i, the i-th argument had
          an illegal value
     > 0, if info = i, the leading minor of order i
          of A is not positive definite, so the
          factorization could not be completed,
          and the solution has not been computed.
```

Here comes **bandsolve.c** (I am using C++-style comments):

```
#include <math.h>
// For Matlab
#include "mex.h"

void dpbsv_(char *, int *, int *, int *, double *,
            int *, double *, int *, int *);

void mexFunction(int nlhs,       mxArray*plhs[],
                 int nrhs, const mxArray *prhs[])
{
  double    *px, *pA, *pb, *pA_tmp;
  mxArray   *A_tmp;
  char uplo = 'L';
  int k, A_rows, A_cols, b_rows, b_cols, kd, info;

  // Check for proper number of arguments
  if (nrhs != 2) {
    mexErrMsgTxt("Two input arguments required.");
  } else if (nlhs > 2) {
    mexErrMsgTxt("Too many output arguments.");
  }

  A_rows = mxGetM(prhs[0]);
  kd     = A_rows - 1;       // # of subdiags
  A_cols = mxGetN(prhs[0]);  // = n

  b_rows = mxGetM(prhs[1]);
  b_cols = mxGetN(prhs[1]);

  if (b_rows != A_cols || b_cols <= 0)
    mexErrMsgTxt("Illegal dimension of b.");
```

```
  // Create a matrix for the return argument
  // and for A. dpbsv destroys A and b).
  // Should check the return status.
  plhs[0]=mxCreateDoubleMatrix(b_rows, b_cols, mxREAL);
  plhs[1]=mxCreateDoubleMatrix(1, 1, mxREAL);
  A_tmp  =mxCreateDoubleMatrix(A_rows, A_cols, mxREAL);

  px = mxGetPr(plhs[0]);     // Solution x
  pA = mxGetPr(prhs[0]);     // A
  pA_tmp = mxGetPr(A_tmp);   // temp for A
  pb = mxGetPr(prhs[1]);     // b

  for (k = 0; k < b_rows * b_cols; k++) // b -> x
    *(px + k) = *(pb + k);

  for (k = 0; k < A_rows * A_cols; k++) // A -> A_tmp
    *(pA_tmp + k) = *(pA + k);

  dpbsv_(&uplo, &A_cols, &kd, &b_cols, pA_tmp,
         &A_rows, px, &b_rows, &info);

  *mxGetPr(plhs[1]) = info; // () higher prec. than*
  if (info)
    mexWarnMsgTxt("Non zero info from dpbsv.");

  // Should NOT destroy plhs[0] or plhs[1]
  mxDestroyArray(A_tmp);
}
```

Some comments:
**nrhs** is the number of input arguments to the MEX-routine. **prhs** is an array of pointers to input arguments. **prhs[0]** points to a so-called, **mxArray**, a C-struct containing size-information and pointers to the matrix-elements.
**prhs[0]** corresponds to the first input variable, **A** etc.

Since one should not access the member-variables in the struct directly, there are routines to extract size and elements.
**A_rows = mxGetM(prhs[0]);** extracts the number of rows and
**A_cols = mxGetN(prhs[0]);** extracts the number of columns.

The lines

```
  plhs[0]=mxCreateDoubleMatrix(b_rows, b_cols, mxREAL);
  plhs[1]=mxCreateDoubleMatrix(1, 1, mxREAL);
```

allocate storage for the results (of type **mxREAL**, i.e. ordinary **double**).

**A_tmp = mxCreateDoubleMatrix(A_rows, A_cols, mxREAL);**
allocates storage for a copy of **A**, since the Lapack-routine destroys the matrix.
**px = mxGetPr(plhs[0]);** extracts a pointer to the (real-part) of the matrix elements and stores it in the pointer variable, **px**.

The first for-loop copies **b** to **x** (which will be overwritten by the solution). The second loop copies the matrix to the temporary storage, pointed to by **A_tmp**. This storage is later deallocated using **mxDestroyArray**.

Note that neither the input- nor the output-arguments should be deallocated.

It is now time to compile and link. This is done using the Bourne-shell script **mex**. We must also make a symbolic link. Since we would like to change some parameters when compiling, we will copy and edit an options file, **mexopts.sh**.

```
% which matlab
/chalmers/sw/sup/matlab-2008b/bin/matlab
(ls -ld /chalmers/sw/sup/matlab* to see the versions)
```

Make the link:

```
% ln -s /usr/lib/libstdc++.so.6.0.3 libstdc++.so
```

Copy **mexopts.sh**:

```
% cp /chalmers/sw/sup/matlab-2008b/bin/mexopts.sh .
```

and edit the file (after **glnx86**):

```
  change CC='gcc' to CC='gcc4'
```

if you are using the latest Matlab-version. In the **CFLAGS**-line, change **-ansi** to **-Wall**, to use C++-style comments and to get more warnings.

Add **-L.** to **CLIBS**, and add linker-info. to get Goto-blas:

```
   CLIBS="$RPATH $MLIBS -lm  -L. -lstdc++
          -L/chalmers/sw/unsup/libgoto/lib
          -lgoto_opt32-r0.96"    NOTE: in one long line
  change -O to -O3 in FOPTIMFLAGS
```

Make sure your **LD_LIBRARY_PATH** contains the name of the directory where Goto-blas resides.

I have fetched the lapack-routines from Netlib:

```
% ls lapack
dpbsv.f   dpbtf2.f  dpbtrf.f  dpbtrs.f  dpotf2.f
ieeeck.f  ilaenv.f  lsame.f   xerbla.f
```

Now it is time to compile:

```
% mex -f ./mexopts.sh bandsolve.c lapack/*.f
```

which creates **bandsolve.mexglx**

Now we can test a simple example in Matlab:

```
>> A = [2 * ones(1, 5); ones(1, 5)]
A =
     2     2     2     2     2
     1     1     1     1     1

>> [x, info] = bandsolve(A, [3 4 4 4 3]')
x =
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
info =
     0
```

Here a case when A is not positive definite:

```
>> A(1, 1) = -2;  % Not positive definite
>> [x, info] = bandsolve(A, [3 4 4 4 3]')
Warning: Non zero info from dpbsv.
x =               % Since b is copied to x
     3
     4
     4
     4
     3
info =
     1
```

Note that the first call of **bandsolve** may take much more time, since the mex-file has to be loaded. Here a small test when n=10000, kd=10:

```
>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.147128 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.034625 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.034950 seconds.
```

Now to some larger problems:
With n=100000 and kd=10, **dpbsv** takes 0.25 s and sparse backslash 0.41 s on a student AMD-computer.
kd=20 gives the times 0.48 s and 0.77 s respectively.

On an Opteron with more memory:
with n=1000000, kd=10 the times are 2.9 s, 4.7 s.
Increasing kd to 50, the times are 15.4 s and 27.6 s.

## Libraries, ar, ld

Numerical (and other software) is often available in libraries. To use a subroutine from a library one has to use the linker to include the routine. Advantages:

- Fewer routines to keep track of.
- There is no need to have source code for the library routines that a program calls.
- Only the needed modules are loaded.

These pages deal with how one can make libraries and use the linker, link-editor, **ld**.

```
% cat sub1.f90
subroutine sub1
  print*, 'in sub1'
end

% cat sub2.f90
subroutine sub2
  print*, 'in sub2'
end

% cat sub3.f90
subroutine sub3
  print*, 'in sub3'
  call sub2
end

% cat main.f90
program main
  call sub3
end
```

```
% ls sub*.f90
sub1.f90  sub2.f90  sub3.f90

% g95 -c sub*.f90
sub1.f90:
sub2.f90:
sub3.f90:

% ls sub*
sub1.f90  sub1.o  sub2.f90  sub2.o  sub3.f90  sub3.o

% ar -r libsubs.a sub*.o

% ar -t libsubs.a
sub1.o
sub2.o
sub3.o

% g95 main.f90 -L. -lsubs
% a.out
 in sub3
 in sub2
```

**g95** calls the link-editor, **ld**, to combine **main.o** and the object files in the library to produce the executable **a.out**-file. Note that the library routines become part of the executable.

If you write **-lname** the link-editor looks for a library file with name **libname.a** (or **libname.so**).

On some systems you may have to give the location of the library using the flag **-L** (**ld** does not look everywhere). **.** means current working directory, but you could have a longer path, of course. You can have several **-L** flags.

From **man ar**:

ar creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier s.
...
An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

**ar** seems to do this even with **ar -r ...** as well.
If your library does not have this index:

```
% g95 main.f90 -L. -lsubs
./libsubs.a: could not read symbols:
 Archive has no index; run ranlib to add one
% ranlib libsubs.a
% g95 main.f90 -L. -lsubs
```

The order of libraries is important:

```
% g95 -c sub4.f90 sub5.f90
sub4.f90:
sub5.f90:

% ar -r libsub45.a sub[45].o

% ar -t libsub45.a
sub4.o
sub5.o
```

```
% cat sub4.f90
subroutine sub4
  print*, 'in sub4'
  call sub2
end

% cat main.f90
program main     ! A NEW main
  call sub4
end

% g95 main.f90 -L. -lsubs -lsub45
./libsub45.a(sub4.o)(.text+0x6f): In function `sub4_':
: undefined reference to `sub2_'
```

**ld** does <u>not</u> go back in the list of libraries.

```
% g95 main.f90 -L. -lsub45 -lsubs
% a.out
 in sub4
 in sub2
```

The compiler uses several system libraries, try **g95 -v ...**.
One such library is the C math-library, **/usr/lib/libm.a**

```
% ar -t /usr/lib/libm.a | grep expm1 | head -1
s_expm1.o

% man expm1
NAME   expm1, expm1f, expm1l  - exponential minus 1

       #include <math.h>
       double expm1(double x);
...
```

```
% cat main.c
#include <math.h>
#include <stdio.h>

int main()
{
  double x = 1.0e-15;

  printf("expm1(x)   = %e\n", expm1(x));
  printf("exp(x) - 1 = %e\n", exp(x) - 1.0);

  return 0;
}

% gcc main.c
/tmp/cc40PH1o.o(.text+0x2b): In function ‘main’:
: undefined reference to ‘expm1’
/tmp/cc40PH1o.o(.text+0x53): In function ‘main’:
: undefined reference to ‘exp’

% gcc main.c -lm
% a.out
expm1(x)   = 1.000000e-15
exp(x) - 1 = 1.110223e-15
```

## Shared libraries

More about **libm**. The following output has been shortened.

```
% ls -l /usr/lib/libm.*
  /usr/lib/libm.a
  /usr/lib/libm.so -> ../../lib/libm.so.6

% ls -l /lib/libm.*
  /lib/libm.so.6 -> libm-2.3.4.so

% ls -l /lib/libm-2.3.4.so
-rwxr-xr-x  1 root root 176195 Aug 20 03:21
    /lib/libm-2.3.4.so
```

What is this last file?

```
% ar -t /lib/libm-2.3.4.so
ar: /lib/libm-2.3.4.so: File format not recognized

Look for symbols (names of functions etc.):
% objdump -t /lib/libm-2.3.4.so | grep expm1
 ...
00009420  w    F .text  0000005c   expm1
 ...
```

**so** means shared object. It is a library where routines are loaded to memory during runtime. This is done by the dynamic linker/loader **ld.so**. The **a.out**-file is not complete in this case, so it will be smaller.

One problem with these libraries is that they are needed at runtime which may be years after the executable was created. Libraries may be deleted, moved, renamed etc.

One advantage is shared libraries can be shared by every process that uses the library (provided the library is constructed in that way).

It is easier to handle new versions, applications do not have to be relinked.

If you link with **-lname**, the first choice is **libname.so** and the second **libname.a**

**/usr/lib/libm.so -> ../../lib/libm.so.6** is a soft link (an “alias”).

```
% ln -s full_path alias
```

The order is not important when using shared libraries (the linker has access to all the symbols at the same time).

A shared library is created using **ld** (not **ar**) or the compiler, the **ld**-flags are passed on to the linker.

```
% g95 -o libsubs.so -shared -fpic sub*.f90
% g95  main.f90 -L. -lsubs
% ./a.out
 in sub4
 in sub2

From man gcc (edited):

-shared
  Produce a shared object which can then be linked with
  other objects to form an executable. Not all systems
  support this option. For predictable results, you mus
  also specify the same set of options that were used
  to generate code (-fpic, -fPIC, or model suboptions)
  when you specify this option.[1]

-fpic
  Generate position-independent code (PIC) suitable for
  use in a shared library, if supported for the target
  machine. Such code accesses all constant addresses
  through a global offset table (GOT). The dynamic
  loader resolves the GOT entries when the program
```

```
  starts (the dynamic loader is not part of GCC; it is
  part of the operating system). ...
```

Since the subroutines in the library are loaded when we run the program (they are not available in **a.out**) the dynamic linker must know where it can find the library.

```
% cd ..
% Examples/a.out
Examples/a.out: error while loading shared libraries:
 libsubs.so: cannot open shared object file: No such
 file or directory

% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:Examples
% Examples/a.out
 in sub4
 in sub2
```

**LD_LIBRARY_PATH** contains a colon separated list of paths where **ld.so** will look for libraries. You would probably use a full path and not **Examples**.

**$LD_LIBRARY_PATH** is the old value (you do not want to do **setenv LD_LIBRARY_PATH Examples** unless **LD_LIBRARY_PATH** is empty to begin with.

The backslash is needed in **[t]csh** (since colon has a special meaning in the shell). In **sh** (Bourbe shell) you may do something like:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:Example
$ export LD_LIBRARY_PATH      (or on one line)
```

Some form of **LD_LIBRARY_PATH** is usually available (but the name may be different). The SGI uses the same name for the path but the linker is called **rld**. Under HPUX 10.20, for example, the dynamic loader is called **dld.sl** and the path **SHLIB_PATH**.

It is possible to store the location of the library when creating **a.out**.

```
% unsetenv LD_LIBRARY_PATH
% g95 -o libsubs.so -shared -fpic sub.f90
% g95 main.f90 -L. -lsubs
% a.out
a.out: error while loading shared libraries:
 libsubs.so: cannot open shared object file:
 No such file or directory
```

Add the directory in to the runtime library search path (stored in **a.out**):

```
 -Wl, means pass -rpath 'pwd' to ld

% g95 -Wl,-rpath 'pwd' main.f90 -L. -lsubs

% cd ..   or cd to any directory
% Examples/a.out
 in sub4
 in sub2
```

A useful command is **ldd** (print shared library dependencies):

```
% ldd a.out
  libsubs.so => ./libsubs.so (0x00800000)
  libm.so.6 => /lib/tls/libm.so.6 (0x009e2000)
  libc.so.6 => /lib/tls/libc.so.6 (0x008b6000)
  /lib/ld-linux.so.2 (0x00899000)
```

Used on our **a.out**-file it will, in the first case, give:

```
% ldd Examples/a.out
    libsubs.so => not found
```

In the second case, using **rpath**, **ldd** will print the full path.

And now to something related:

Large software packages are often spread over many directories. When distributing software it is customary to pack all the directories into one file. This can be done with the **tar**-command (tape archive). Some examples:

```
% ls -FR My_package
bin/        doc/      install*  lib/      README
configure*  include/  INSTALL   Makefile  src/

My_package/bin:      binaries

My_package/doc:      documentation
userguide.ps   or in pdf, html etc.

My_package/include:  header files
params.h  sparse.h

My_package/lib:      libraries

My_package/src:      source
main.f  sub.f
```

Other common directories are **man** (for manual pages), **examples**, **util** (for utilities).

**README** usually contains general information, **INSTALL** contains details about compiling, installation etc. There may be an **install**-script and there is usually a **Makefile** (probably several).

If the package is using X11 graphics there may be an **Imakefile**. The tool **xmkmf** (using **imake**) can generate a Makefile using local definitions and the **Imakefile**.

In a Linux environment binary packages (such as the Intel compilers) may come in RPM-format. See **http://www.rpm.org/** or type **man rpm**, for details.

Let us now create a tar-file for our package.

```
% tar cvf My_package.tar My_package
My_package/
My_package/src/
My_package/src/main.f
My_package/src/sub.f
My_package/doc/
 ...
My_package/Makefile
```

One would usually compress it:

```
% gzip My_package.tar  (or using bzip2)
```

This command produces the file **My_package.tar.gz**.
**.tgz** is a common suffix as well (**tar.bz2** or **.tbz2** for **bzip2**).

To unpack such a file we can do (using gnu **tar**) (**z** for **gunzip**, or **zcat**, **x** for extract, **v** for verbose and **f** for file):

```
% tar zxvf My_package.tar.gz
My_package
My_package/src/
 ...
```

Using **tar**-commands that do not understand **z**:

```
% zcat      My_package.tar.gz | tar vxf -    or
% gunzip -c My_package.tar.gz | tar vxf -    or
% gunzip <  My_package.tar.gz | tar vxf -    or
% gunzip    My_package.tar.gz            followed by
% tar xvf   My_package.tar
```

I recommend that you first try:

```
% tar ztf My_package.tar.gz
My_package/  ...
```

To see that files are placed in a new directory (and that are no name conflicts).
Under GNOME there is an Archive Manager (File Roller) with a GUI. Look under **Applications/System Tools**

## An Overview of Parallel Computing

Flynn's Taxonomy (1966). Classification of computers according to number of instruction and data streams.

- SISD: Single Instruction Single Data, the standard uniprocessor computer (workstation).
- MIMD: Multiple Instruction Multiple Data, collection of autonomous processors working on their own data; the most general case.
- SIMD: Single Instruction Multiple Data; several CPUs performing the same instructions on different data.
  The CPUs are synchronized.
  Massively parallel computers.
  Works well on regular problems. PDE-grids,
  image processing.
  Often special languages and hardware. Not portable.

  Typical example, the Connection Machines from Thinking Machines (bankruptcy 1994).
  The CM-2 had up to 65536 (simple processors).
  PDC had a 16384 proc. CM200.

  Often called "data parallel".

Two other important terms:

- fine-grain parallelism - small tasks in terms of code size and execution time
- coarse-grain parallelism - the opposite

We talk about granularity.