Cornell University

Department of Physics

Phys 480/680, Astro 690 April 15, 2005

Computational Physics, Spring 2005

Homework Assignment # 6

(Due Thursday, March 31)

Agenda and readings for the week of April 5:

Goal: Optimize to produce *efficient*, high-precision software to calculate the electronic structure of atoms within density-functional theory; explore FLOP rates of various operations and cache performance.

Readings marked NR are from *Numerical Recipes: The Art of Scientific Computing*, 2^{nd} edition (in C). Readings marked LN are from the course lecture notes to be found at http://www.ccmr.cornell.edu/~muchomas/P480.

Recap of last three weeks:

- 03/08 Lec 14: Recap modified Broyden method; Introduction to numerical linear algebra; Benefits of LAPACK; Solution of linear systems through LU decomposition; Crout's algoritm for LU decomposition
  Reading: NR 2.0, 2.3
- 03/10 Lec 15: Crout's Algorithm with pivoting
  Reading: NR 2.3
- 03/10 Lab 5: "Hot-wired" solver for atoms within Hartree theory, reproducing analytic Hydrogenic solutions to Schrödinger's equation $(-(1/2)Z^2/n^2)$.
- 03/15 Lec 16: Inner loop tricks; Cache optimization, ...
- 03/17 Lab 6 (Meet in Rock B3): Profiling and real-time optimization of full atomic solver: using gprof; avoiding malloc; inlining optimizations.
- 03/21-03/25: SPRING BREAK

Preview of coming week:

- 03/29 Lec 17 Avoiding cache misses; BLAS1-3; blocked matrix multiplies; Automated empirical optimization of software (AEOS): automatically tuned linear algebra software (ATLAS), fastest Fourier Transform in the West (FFTW).
- 03/31 Lec 18: Solution of Poisson's equation in d¿1 dimensions; expressive software
- 03/31 Lab 7: FLOP performance of basic operations; FLOP rates in octave.

Contents

1 Optimization **3**
- 1.1 Calls to dvector() and free_dvector() . . . 3
- 1.2 Calls to rk4p480() . . . 4
- 1.3 Calls to derivs_Schrodinger() . . . 5
- 1.4 Calls to exc() and excp() . . . 5

2 Uranium **5**

3 Inner-loop operations **6**
- 3.1 Expected peak processing rate . . . 6
- 3.2 timer() . . . 7
- 3.3 Other inner-loop operations . . . 7

4 Basic Linear Algebra Subroutines (BLAS) **8**
- 4.1 BLAS2 . . . 8
- 4.2 BLAS3 . . . 8

5 Linear Algebra Package (LAPACK) **9**

6 Matlab/Octave **10**
- 6.1 Timing matrix-matrix multiplication . . . 10

A "timer.c" **10**

B timer() for Windows **14**

C "lapack.c" **16**

D "lapack.c" for Windows **18**

E LAPACK **21**

F "octtime.m" **22**

1 Optimization

The most dramatic improvements (often by many orders of magnitude) in computational science almost always come from the development of better algorithms, such as the modified Broyden algorithm above.

Important gains (one or two orders of magnitude) also may be achieved through making proper use of the CPU and its architecture. Among such optimizations, often the greatest gains come from eliminating unnecessary operations in those parts of the software which are executed many times over and over. These operations are most frequently buried in the deepest loop in the code, the so-called "inner-most" loop.

As we learned in lab, among the most wasteful operations in the inner-most loop which should be avoided at all costs are subroutine calls.

To ascertain the impact of the improvements we are about to make, please "#define Itmx 10" in your code and time it. (For us, at this stage, broyden took 10 sec to run the initial five plus the ten Broyden iterations.)

Running our initial code through the gnu profiler (compiling with the -pg -O3 flags, running the code and then typing "gprof"), we found the data below.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
56.54	1.08	1.08	25	43.20	76.40	getg
27.75	1.61	0.53	2419	0.22	0.27	schint
6.81	1.74	0.13	25	5.20	5.25	getphi
3.66	1.81	0.07	4737925	0.00	0.00	rk4p480
1.57	1.84	0.03	14063775	0.00	0.00	derivs_Schrodinger
1.57	1.87	0.03	200050	0.00	0.00	excp
1.05	1.89	0.02	100025	0.00	0.00	exc
0.52	1.90	0.01	14221421	0.00	0.00	dvector
0.52	1.91	0.01	14221421	0.00	0.00	free_dvector
0.00	1.91	0.00	150000	0.00	0.00	derivs_Poisson
0.00	1.91	0.00	1880	0.00	0.27	func_Schrodinger
0.00	1.91	0.00	389	0.00	0.27	func_SchrodingerNodes
0.00	1.91	0.00	150	0.00	0.00	simpint
0.00	1.91	0.00	125	0.00	0.83	rtbisp480
0.00	1.91	0.00	75	0.00	0.54	getPsi
0.00	1.91	0.00	75	0.00	6.72	zriddrp480
0.00	1.91	0.00	30	0.00	0.00	dmatrix
0.00	1.91	0.00	30	0.00	0.00	free_dmatrix
0.00	1.91	0.00	25	0.00	0.00	d3tensor
0.00	1.91	0.00	25	0.00	0.00	free_d3tensor
0.00	1.91	0.00	10	0.00	0.00	lubksbp480
0.00	1.91	0.00	10	0.00	0.00	ludcmpp480
0.00	1.91	0.00	2	0.00	0.00	free_ivector
0.00	1.91	0.00	2	0.00	0.00	ivector
0.00	1.91	0.00	1	0.00	1910.00	main

1.1 Calls to dvector() and free_dvector()

Above, it is clear that among the most frequently called routines are dvector and free_dvector. These calls are particularly wasteful because they involve requesting memory from the operating system. Gprof also tells us that almost all of these calls come from rk4p480!

```
-----------------------------------------------
                0.00    0.00       4/14221421     main [1]
                0.00    0.00      10/14221421     ludcmpp480 [17]
```

```
                0.00    0.00      75/14221421     getphi [7]
                0.00    0.00     150/14221421     getPsi [11]
                0.00    0.00     150/14221421     getg [3]
                0.00    0.00    7257/14221421     schint [4]
                0.01    0.00 14213775/14221421     rk4p480 [8]
[15]     0.5    0.01    0.00 14221421         dvector [15]
-----------------------------------------------
```

To eliminate these calls comment out the lines

```
dym=dvector(1,n);
dyt=dvector(1,n);
yt=dvector(1,n);
       .
       .
free_dvector(yt,1,n);
free_dvector(dyt,1,n);
free_dvector(dym,1,n);
```

from rk4p480(), and replace the declarations of dym, dyt and yt with

```
double dym[3],dyt[3],yt[3];
```

This simple change reduced our run time (recompiling with just the -O3 flag to get an accurate timing) from 10 sec to 5 sec!!!

1.2 Calls to rk4p480()

The calls to derivs_Schrodinger() come from the inner loop of schint() calling rk4p480(). We can avoid these calls by including the derivs_Schrodinger() code directly into rk4p480() and the rk4p480() code directly into schint(). We can do this by hand, at the risk of introducing many bugs, or we can induce the compiler to do this.

To get the gnu compiler to do this you must include code for all three functions, derivs_Schrodinger(), rk4p480() and schint(), in the same file, right next to each other, and in this same sequence. Also, to allow "in-lining" you must compile the code with the optimization flag "-O3". So that we may further modify rk4p480(), please keep the original copy and call your new copy of the routine rk4p480Sch(). Also, be sure to update the call in schint() to call the new routine rk4p480Sch().

Upon our making these changes, gprof now reports the following.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.90      1.91     1.91     2419     0.79     1.18  schint
 25.82      2.86     0.95 14063775     0.00     0.00  derivs_Schrodinger
 12.50      3.32     0.46       25    18.40   147.20  getg
  4.35      3.48     0.16   200050     0.00     0.00  excp
  2.45      3.57     0.09   100025     0.00     0.00  exc
  1.36      3.62     0.05       25     2.00     3.20  getphi
  0.54      3.64     0.02   150000     0.00     0.00  derivs_Poisson
  0.54      3.66     0.02       75     0.27     2.63  getPsi
  0.27      3.67     0.01    50000     0.00     0.00  rk4p480
  0.27      3.68     0.01       75     0.13    29.77  zriddrp480
  0.00      3.68     0.00     7646     0.00     0.00  dvector
  0.00      3.68     0.00     7646     0.00     0.00  free_dvector
  0.00      3.68     0.00     1880     0.00     1.18  func_Schrodinger
```

```
  0.00      3.68     0.00      389     0.00     1.18  func_SchrodingerNodes
  0.00      3.68     0.00      150     0.00     0.00  simpint
  0.00      3.68     0.00      125     0.00     3.68  rtbisp480
  0.00      3.68     0.00       30     0.00     0.00  dmatrix
  0.00      3.68     0.00       30     0.00     0.00  free_dmatrix
  0.00      3.68     0.00       25     0.00     0.00  d3tensor
  0.00      3.68     0.00       25     0.00     0.00  free_d3tensor
  0.00      3.68     0.00       10     0.00     0.00  lubksbp480
  0.00      3.68     0.00       10     0.00     0.00  ludcmpp480
  0.00      3.68     0.00        2     0.00     0.00  free_ivector
  0.00      3.68     0.00        2     0.00     0.00  ivector
  0.00      3.68     0.00        1     0.00  3680.00  main
```

Note that there are now far fewer calls to rk4p480() and no calls to rk4p480Sch()! This is because the compiler has in-lined rk4p480Sch() and included it explicitly into schint, avoiding all of the unnecessary associated subroutine calls. There are still, however, calls to derivs_Schrodinger().

Important note: If the compiler *still* did not inline your routines, the default compiler limit on size of functions to be inlined may be too small. To fix this, try adding the compiler option "-finline-limit=10000".

1.3 Calls to derivs_Schrodinger()

The compiler could not eliminate the calls to derivs_Schrodinger() because rk4p480() takes a variable function (*derivs) as input and so the compiler does not know that we always intend to call derivs_Schrodinger() from rk4p480Sch. To inline derivs_Schrodinger() as well, replace all occurrences of the text "(*derivs)" with the text "derivs_Schrodinger" in the body of the routine rk4p480Sch(). (**Note:** do *not* do this in the function header, that will turn derivs_schrodinger back into a variable!)

With this, all calls to derivs_Schrodinger() vanished, and our run time reduced to 3.8 sec!

1.4 Calls to exc() and excp()

The next most common calls are to exc() and excp(). To inline these, make sure that they are in the same file with getVxc() and getDExc(). To induce the compiler to inline them, you may need to add the keyword "inline" to the prototypes:

```
inline double exc(double rs)

inline double excp(double rs)
```

This reduced our time to 3.3 sec. From the original solution time of 26 sec at the start of this problem set, we now have nearly an order of magnitude improvement!

Write down your final run time to show the grader and discuss during lab.

2 Uranium

As a last exercise for your tuned atomic software, for fun (not required), you may wish to see if your code can handle uranium, the largest naturally occurring element. The specifications for it are

```
/* Specs for U */
Z=92;
lmax=3;
```

```
nmax=ivector(0,lmax);
nmax[0]=6;
nmax[1]=4;
nmax[2]=3;
nmax[3]=1;

nmaxmax=0;
for (l=0; l<=lmax; l++)
  if (nmax[l]>nmaxmax) nmaxmax=nmax[l];

F=dmatrix(0,lmax,0,nmaxmax);
F[0][0]=2.;
F[0][1]=2.;
F[0][2]=2.;
F[0][3]=2.;
F[0][4]=2.;
F[0][5]=2.;
F[0][6]=2.;

F[1][0]=6.;
F[1][1]=6.;
F[1][2]=6.;
F[1][3]=6.;
F[1][4]=6.;

F[2][0]=10.;
F[2][1]=10.;
F[2][2]=10.;
F[2][3]=1;

F[3][0]=14.;
F[3][1]=3.;
```

3 Inner-loop operations

The preceding basic optimizations complete our work with the atomic software. To further explore the issue of floating point performance in general, this problem allows you to evaluate for yourself, on your own computer, the compute rate and relative cost of different operations in the innermost loop.

To help you with the problem, Appendix A contains a function timer() which computes the FLOP rate for the various subroutines which you will write; a main() program which prints out various information; and a function mult() as an example of a routine to be timed. At present, much of the main() program is commented out. By the end of this problem set, you should have all of main() program functioning.
Note: Because of operating system dependencies in timing routines, those using the Windows compiler should replace timer() from Appendix A with the routine in Appendix B.

3.1 Expected peak processing rate

As discussed in lecture, the peak processing rate you should expect to see should be approximately one-half to two times the clock speed. To put the results of your runs below in context, have ready for the grader the clock speed of the processor on which you did your runs. You may know this directly if you are working

on your own machine. Or, if you are running linux, you should be able to find it on the line "cpu MHz" in the file /proc/cpuinfo.

3.2 timer()

Compile the program in Appendix A (replacing timer() with Appendix B if you are using Windows).

When you have the timer() routine working, compile the program with the highest optimization level of your compiler (options "-O3 -fomit-frame-pointer" on the gnu compiler, or "release" version for the Windows compiler) and compare the MFLOPS rate with the clock speed of your machine in MHz. Be sure to try problem sizes (128..512) that produce enough operations to get reasonable, repeatable measures of the rate. Optional: You may also which to compare the MFLOPS rate when you compile with or without optimization.

3.3 Other inner-loop operations

Write subroutines, with prototypes identical to mult(), which perform the following operations:

1. divide(): works just like mult (doing n^3 operations) but, after initializing x to 1, does n^3 divisions of x by 1.000001.

2. multsub(): performs the same operation as mult, but where multsub() makes a subroutine call

   ```
   sub(x,1.000001)
   ```

   to perform the computation. This measures the time needed to make the subroutine calls. To prevent the compiler from automatically in-lining the subroutine, you should put sub() in a separate file.

3. multif(): within the same loop as for mult(), test whether x¡1.00001. If so, multiply by 1.000001; otherwise multiply by 0.99. Note that you must return the value of x: If you forget to return x, many compilers are smart enough to realize that you don't need the loop, and they'll skip it entirely!

4. multal(): works just like mult, but, within the loop, makes a call to allocate/deallocate memory

   ```
   p=dvector(0,1);
   free_dvector(p,0,1);
   ```

   This gives a feel for the cost of making calls to the operating system.

5. multadd(): works like mult(), but performs multiply-add operations instead,

   ```
   x+=x*0.000001;
   ```

   Note that each of these counts as two FLOP (one add and one multiply).

6. cmat1(): loops through all n×n elements of the matrix M1 and copies them into matrix M3 with a loop which steps through memory sequentially, and repeats this operation n times:

   ```
   for (k=0; k<n; k++)
     for (i=0; i<n; i++)
       for (j=0; j<n; j++)
         M3[i][j]=M1[i][j];
   ```

   Note that this loop will run a lot *slower* than mult(), which does a number of multiplies equal to the number of copies here. This gives an idea of the cost of accessing the main memory.

7. cmat2(): The same as cmat1(), but now with the order of the loops changed. This gives an idea of the cost of a cache-miss, equivalently the cache-line length (measured in number of double's). Note that, because some part of your matrices fit in cache, you'll have to run large problems to see the true cost of a cache miss.

Finally, in main, uncomment the timer() calls to your new routines as well as the section of print statements labeled "Print out time ratios." Running your code for various sizes, you should find that most operations proceed at a constant rate, except for the cmat operations. The speed of these depends on the size of your matrices compared to the size of the cache memory!

4 Basic Linear Algebra Subroutines (BLAS)

This problem develops experience using cache in the context of basic linear algebraic subroutines (BLAS).

4.1 BLAS2

As a typical BLAS2 operation, we here consider vector-matrix multiplication (the example from lecture),

$$v_2[i] = \sum_k v_1[j] M_1[j][i]. \tag{1}$$

To assess the importance of the ordering of the loops, write the following routines (same prototype as mult()), uncomment their calls through timer in the main program in "timer.c", comment out preceding calls to timer, and run the resulting program.

1. vecmat1(): code the multiplication (1) using the naïve sequence of loops (first i, then j)
2. vecmat2(): code the multiplication (1) reversing the sequence of the loops

4.2 BLAS3

The BLAS3 routines, which involve n^3 operations (and only n^2 data) are where the great gains from cache may be had. Write the routines below (same prototype as mult()), and uncomment the rest of the main program. Note that, for debugging, the main program checks your two matrix-matrix multiplications routines against each other. This is useful because block-matrix multiply is somewhat intricate and prone to coding bugs.

1. matmat1(): standard implementation of the matrix multiplication

$$M_3[i][j] = \sum_{k=1}^{n} M_1[i][k] M_2[k][j]$$

2. matmat3(): matrix multiplication $M_3 = M_1 M_2$ via blocks of size Nb×Nb, where Nb is set in a #define statement. (Nb=32 is a reasonable number.)

   Note: To simplify the exercise, you may assume that Nb divides the total matrix size evenly.

Hint: Pseudo-code for the block-matrix multiplication follows:

```
double b1[Nb][Nb],b2[Nb][Nb],b3[Nb][Nb]; /* Regular C arrays are faster
                                             -- use them! */

for (I=0; I<n/Nb; I++)
  for (J=0; J<n/Nb; J++) { /* Compute IJ block of M3: */


    -> YOUR CODE: zero out b3, an Nb x Nb array to store IJ block of M3
```

```
    for (K=0; K<n/Nb; K++) { /* Add to b3 product of IK block of M1
                                with KJ block of M2 */
      /* Copy IK block of M1 into b1, an Nb x Nb matrix */
      for (i=0; i<Nb; i++)
        for (k=0; k<Nb; k++)
         b1[i][k]=M1[I*Nb+i][K*Nb+k];

      /* Copy KJ block of M2 into b2, an Nb x Nb matrix */
      -> YOUR CODE

      /* Do b1*b2, adding result into b3 */
      for (i=0; i<Nb; i++)
        for (j=0; j<Nb; j++)
          for (k=0; k<Nb; k++)
            b3[i][j]+=b1[i][k]*b2[k][j];
    }
  -> YOUR CODE: Copy b3 into IJ block of M3
  }
}
```

5 Linear Algebra Package (LAPACK)

This problem demonstrates the power of a highly optimized package such as LAPACK over *Numerical Recipes*. Compile and run the main program in Appendix C(Appendix D if you are using Windows), which compares the MFLOPS rates of Numerical Recipes routines ludcmpp480() and lubksbp480() for solving a linear system of N=512 equations in 512 unknown against the LAPACK routine dgesv(). The program also checks that the equations are solved correctly. You may wish to set the problem size N=512 (#define at top of code) to a smaller number while you are debugging.

Because LAPACK is written in Fortran 77, much of the work in calling LAPACK is to get the code to compile. You will note, for instance, that because Fortran 77 accepts all arguments as pointers to the corresponding data, the arguments in the call to dgesv() each represent the *address* of the first element of a set of data. (The &'s to the left of each argument accomplish this.)

Because of the above subtleties, Appendix C/D gives a completed program for you. You still likely will have to spend some time hacking with the code to get it to compile. Compilers on UNIX systems, for instance, usually put underscore characters ('_') to the right of the names of Fortran 77 routines (as we have done in the appendix). Your system may do something different and you may have to experiment with moving the '_' or, perhaps, eliminating it!

On the physics educational cluster, ATLAS LAPACK and BLAS routines are the default -llapack and -lblas libraries. Also, because lapack is written in fortran, you will need to include the fortran library -lg2c as well. Thus, the following statement should compile your code on the cluster:

```
cc -O3 lapack.c p480.c nrutil.c -llapack -lblas -lg2c -lm
```

If you cannot find the LAPACK or BLAS libraries for your machine, Appendix E gives links to the P480 home page where you can down-load C code for the LAPACK and BLAS routines that you will need, and some hints for compiling. (Thanks to Nick Bailey, the grader for Spring 2000, for getting this for you!) Note that the code will run a lot faster if you can use the ATLAS BLAS.

Important Note: To use LAPACK from C in real applications, it is important to keep in mind that LAPACK is written in Fortran and expects matrices to be stored in memory in the opposite order from C. This means that, *if you start in C, you should send LAPACK the transposes of any matrices (and expect to get transposes back).* For the test program in this example, we work with symmetric matrices — so this transposing isn't needed!

6 Matlab/Octave

For the last phase of this course, we will develop software using the octave (almost identical to the proprietary software matlab) programming language. Octave calls ATLAS BLAS and thus should approach theoretical peak machine performance. This problem is designed to familiarize you with this language and to verify the high performance attained using the language.

A very good introduction and reference for `octave` is http://www.octave.org/doc/octave_toc.html. If you are not already familiar with the language, you should read and try the examples in Section 1, "A Brief Introduction to Octave" of this document.

6.1 Timing matrix-matrix multiplication

Cut and paste the code in Appendix F into the file "octtime.m" and insert your own octave code at the two locations marked "Place your own code here" to compute the MFLOPS rate for matrix-matrix multiplication and solution of a linear system via LU decomposition (which requires $2N^3/3$ operations), the same computation carried out in lapack.c. Then, start octave, and at the octave prompt, type "more off" so that you can see the output appear as it is computed, and then type "octtime" to execute the octave program. Compare to the microprocessor clock speed in /proc/cpuinfo!

A "timer.c"

```
#include <stdio.h>
#include <sys/time.h>
#include "nrutil.h"
#include <math.h>

double mult(int n,double **M1,double **M2,double **M3,
          double *v1,double *v2,double *v3)
{
  int i;
  double x=1.;

  for (i=0; i<n*n*n; i++)
    x=x*1.000001;

  return x;
}

double timer(char *name,double flop,
             double (*func)(int,double **,double **,double **,
                            double *,double *,double *),
             int n)
{
  /* Timer variables */
  struct timeval time1,time2;
  double ctime,mflop;
  double **M1,*v1;
  double **M2,*v2;
  double **M3,*v3;
  int i,j;
```

```
M1=dmatrix(0,n,0,n);
M2=dmatrix(0,n,0,n);
M3=dmatrix(0,n,0,n);

v1=dvector(0,n);
v2=dvector(0,n);
v3=dvector(0,n);

/* Fill in values for input vectors */
for (i=0; i<n; i++) {
  v1[i]=3.14159;
  v2[i]=0.314159;
}

/* Fill in values for input matrices */
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    M1[i][j]=1./(1+i+j);
    M2[i][j]=1./(2+i+j);
  }

/* Record time of function call */
gettimeofday(&time1,NULL);

/* Call function */
func(n,M1,M2,M3,v1,v2,v3);

/* Record time of function return */
gettimeofday(&time2,NULL);

/* Coupute time in seconds */
ctime=
  (time2.tv_sec-time1.tv_sec) /* Integer seconds */
  +(time2.tv_usec-time1.tv_usec)/1.e6; /* Integer giving microseconds part */

/* Convert from FLOP to MFLOP */
mflop=flop/1e6;

/* Output result */
fprintf(stderr,"%20s: %4.0lf MFLOPS  = %.1lf MFLOP / %.3f sec\n",
        name,mflop/ctime,mflop,ctime);

/* Free memory before return */
free_dvector(v1,0,n);
free_dvector(v2,0,n);
free_dvector(v3,0,n);

free_dmatrix(M1,0,n,0,n);
free_dmatrix(M2,0,n,0,n);
free_dmatrix(M3,0,n,0,n);
```

```
  return ctime;
}

main()
{
  /* Place to store run times of various routines */
  double tm,td,ts,ti,tal,tac1,tac2;

  /* Variables for verification of matmat3 */
  int i,j;
  double **M1,**M2,**M3,**M4;
  double err=0.;

  /* Problem size (n x n) */
  int n;

  printf("Problem size: ");
  scanf("%d",&n);

  /* Calls to timers which don't access much memory */
  tm=timer("mult",(double) n*n*n,mult,n);
/*    td=timer("divide",(double) n*n*n,divide,n); */
/*    ts=timer("sub ",(double) n*n*n,multsub,n); */
/*    ti=timer("if ",(double) n*n*n,multif,n); */
/*    tal=timer("alloc",(double) n*n*n,multal,n); */
/*    timer("multadd",(double) 2*n*n*n,multadd,n); */

  /* Time strides through memory */
/*    tac1=timer("cmat1",(double) n*n*n,cmat1,n); */
/*    tac2=timer("cmat2",(double) n*n*n,cmat2,n); */

  /* Print out time ratios */
/*    printf("Notes:\n"); */
/*    printf("Division      costs %4.1f mults\n",td/tm); */
/*    printf("Subroutine    costs %4.1f mults\n",ts/tm-1); */
/*    printf("If            costs %4.1f mults\n",ti/tm-1); */
/*    printf("malloc        costs %4.1f mults\n",tal/tm-1); */
/*    printf("Memory access costs %4.1f mults\n",tac1/tm-1); */
/*    printf("Cache miss    costs %4.1f accesses\n",tac2/tac1); */
/*    printf("\n"); */

  /* Matrix-vector operations */
/*    timer("vecmat1",(double) 2*n*n*n,vecmat1,n); */
/*    timer("vecmat2",(double) 2*n*n*n,vecmat2,n); */
/*    timer("matmat1",(double) 2*n*n*n,matmat1,n); */
/*    timer("matmat2",(double) 2*n*n*n,matmat2,n); */

  /* Verify matmat2 */
/*    M1=dmatrix(0,n,0,n); */
/*    M2=dmatrix(0,n,0,n); */
/*    M3=dmatrix(0,n,0,n); */
```

```
/*    M4=dmatrix(0,n,0,n); */

  /* Fill in values for input matrices */
/*    for (i=0; i<n; i++) */
/*      for (j=0; j<n; j++) { */
/*        M1[i][j]=1./(1+i+j); */
/*        M2[i][j]=1./(2+i+j); */
/*      } */

  /* Note, the matmat routines don't use their last three arguments anyway */
/*    matmat1(n,M1,M2,M3,M1[0],M2[0],M3[0]); */
/*    matmat2(n,M1,M2,M4,M1[0],M2[0],M3[0]); */

/*    for (i=0; i<n; i++) */
/*      for (j=0; j<n; j++) */
/*        err+=pow(M4[i][j]-M3[i][j],2); */
/*    printf("\nIMPORTANT: Test of matmat2; should be near zero... %e\n\n",err); */

  /* Free up space */
/*    free_dmatrix(M1,0,n,0,n); */
/*    free_dmatrix(M2,0,n,0,n); */
/*    free_dmatrix(M3,0,n,0,n); */
/*    free_dmatrix(M4,0,n,0,n); */
}
```

B timer() for Windows

.

```
#include <sys/timeb.h>
double timer(char *name,double flop,
             double (*func)(int,double **,double **,double **,
                            double *,double *,double *),
             int n)
{
 /* Timer variables */
        struct _timeb time1,time2;
  double ctime,mflop;
  double **M1,*v1;
  double **M2,*v2;
  double **M3,*v3;
  int i,j;

  M1=dmatrix(0,n,0,n);
  M2=dmatrix(0,n,0,n);
  M3=dmatrix(0,n,0,n);

  v1=dvector(0,n);
  v2=dvector(0,n);
  v3=dvector(0,n);

  /* Fill in values for input vectors */
  for (i=0; i<n; i++) {
    v1[i]=3.14159;
    v2[i]=0.314159;
  }

  /* Fill in values for input matrices */
  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      M1[i][j]=1./(1+i+j);
      M2[i][j]=1./(2+i+j);
    }

  /* Record time of function call */
        _ftime(&time1);

  /* Call function */
  func(n,M1,M2,M3,v1,v2,v3);

  /* Record time of function return */
        _ftime(&time2);


  /* Compute time in seconds */
        ctime = (time2.time-time1.time) /* Integer seconds */
         +(time2.millitm-time1.millitm)/1.e3; /* Integer giving milliseconds part */
```

```
  /* Convert from FLOP to MFLOP */
  mflop=flop/1e6;

  /* Output result */
  fprintf(stderr,"%20s: %4.0lf MFLOPS  = %.1lf MFLOP / %.3f sec\n",
          name,mflop/ctime,mflop,ctime);

  /* Free memory before return */
  free_dvector(v1,0,n);
  free_dvector(v2,0,n);
  free_dvector(v3,0,n);

  free_dmatrix(M1,0,n,0,n);
  free_dmatrix(M2,0,n,0,n);
  free_dmatrix(M3,0,n,0,n);

  return ctime;
}
```

C "lapack.c"

```
#include <stdio.h>
#include <sys/time.h>
#include "nrutil.h"
#include <math.h>
#define N 512

main()
{
  /* Timer stuff */
  struct timeval time1,time2;
  double rtime,mflop;

  /* Stuff for LAPACK call */
  int NL=N,NRHS=1,LDA=N,*IPIV,LDB=N,INFO;

  /* Declare arrays and variables */
  double **M1,*v1;
  double **M2,*v2;
  double **M3,*v3;

  double tmp,error=0.;
  int *indx;
  int i,j;

  M1=dmatrix(1,N,1,N);
  M2=dmatrix(1,N,1,N);
  M3=dmatrix(1,N,1,N);

  v1=dvector(1,N);
  v2=dvector(1,N);
  v3=dvector(1,N);

  indx=ivector(1,N);
  IPIV=ivector(1,N);

  /* Matrices and vectors */
  for (i=1; i<=N; i++)
    for (j=1; j<=N; j++) {
      M1[i][j]=1./(1+i+j);
      M2[i][j]=1./(1+i+j);
      M3[i][j]=1./(1+i+j);
    }
  for (i=1; i<=N; i++) {
    v1[i]=1./(1+i);
    v2[i]=1./(1+i);
    v3[i]=1./(1+i);
  }

  /* Numerical Recipes */
  gettimeofday(&time1,NULL);
```

```
  ludcmpp480(M1,N,indx,&tmp); /* Get LU decomposition */
  lubksbp480(M1,N,indx,v1); /* Use LU decomposition to solve equations */
  gettimeofday(&time2,NULL);

  /* Output timing */
  mflop=(double) 2*N*N*N/3./1e6;
  rtime=
    (time2.tv_sec-time1.tv_sec)+(time2.tv_usec-time1.tv_usec)/1.e6;

  printf("\n");
  printf("Numrec:  %.0lf MFLOP in %.3f sec ==> %.0lf MFLOPS.\n",
          mflop,rtime,mflop/rtime);

  /* LAPACK */
  gettimeofday(&time1,NULL);
  dgesv_(&NL,&NRHS,&M2[1][1],&LDA,&IPIV[1],&v2[1],&LDB,&INFO);
  gettimeofday(&time2,NULL);
  if (INFO!=0) {
    printf("Error in LAPACK call: %d",INFO);
    exit(1);
  }

  /* Output timing info */
  mflop=(double) 2*N*N*N/3./1e6;
  rtime=
    (time2.tv_sec-time1.tv_sec)+(time2.tv_usec-time1.tv_usec)/1.e6;
  printf("LAPACK:  %.0lf MFLOP in %.3f sec ==> %.0lf MFLOPS.\n",
          mflop,rtime,mflop/rtime);

  /* Check results */
  printf("\n");
  for (i=1; i<=N; i++) {
    tmp=-v3[i];
    for (j=1; j<=N; j++)
      tmp+=M3[i][j]*v1[j];
    error+=tmp*tmp;
  }
  printf("Numerical Recipes rms error: %e\n",sqrt(error/N));

  for (i=1; i<=N; i++) {
    tmp=-v3[i];
    for (j=1; j<=N; j++)
      tmp+=M3[i][j]*v2[j];
    error+=tmp*tmp;
  }
  printf("LAPACK            rms error: %e\n",sqrt(error/N));
  printf("\n");
}
```

D "lapack.c" for Windows

```
#include <stdio.h>
#include <sys/timeb.h>
#include "nrutil.h"
#include "p480.h"
#include <math.h>
#define N 512


main()
{
  /* Timer stuff */
  struct _timeb time1,time2;
  double rtime,mflop;

  /* Stuff for LAPACK call */
  int NL=N,NRHS=1,LDA=N,*IPIV,LDB=N,INFO;

  /* Declare arrays and variables */
  double **M1,*v1;
  double **M2,*v2;
  double **M3,*v3;

  double tmp,error=0.;
  int *indx;
  int i,j;

  M1=dmatrix(1,N,1,N);
  M2=dmatrix(1,N,1,N);
  M3=dmatrix(1,N,1,N);

  v1=dvector(1,N);
  v2=dvector(1,N);
  v3=dvector(1,N);

  indx=ivector(1,N);
  IPIV=ivector(1,N);

  /* Matrices and vectors */
  for (i=1; i<=N; i++)
    for (j=1; j<=N; j++) {
      M1[i][j]=1./(1+i+j);
      M2[i][j]=1./(1+i+j);
      M3[i][j]=1./(1+i+j);
    }
  for (i=1; i<=N; i++) {
    v1[i]=1./(1+i);
    v2[i]=1./(1+i);
    v3[i]=1./(1+i);
  }
```

```
  /* Numerical Recipes */
  _ftime(&time1);
  ludcmpp480(M1,N,indx,&tmp); /* Get LU decomposition */
  lubksbp480(M1,N,indx,v1); /* Use LU decomposition to solve equations */
  _ftime(&time2);


  /* Output timing */
  mflop=(double) 2*N*N*N/3./1e6;
  rtime=
    (time2.time-time1.time)+(time2.millitm-time1.millitm)/1.e6;

  printf("\n");
  printf("Numrec:  %.0lf MFLOP in %.3f sec ==> %.0lf MFLOPS.\n",
          mflop,rtime,mflop/rtime);

  /* LAPACK */
  _ftime(&time1);
  //dgesv_(&NL,&NRHS,&M2[1][1],&LDA,&IPIV[1],&v2[1],&LDB,&INFO);
  _ftime(&time2);
  /*if (INFO!=0) {
    printf("Error in LAPACK call: %d",INFO);
    exit(1);
  }*/

  /* Output timing info */
  mflop=(double) 2*N*N*N/3./1e6;
  rtime=
    (time2.time-time1.time)+(time2.millitm-time1.millitm)/1.e3;
  printf("LAPACK:  %.0lf MFLOP in %.3f sec ==> %.0lf MFLOPS.\n",
          mflop,rtime,mflop/rtime);

  /* Check results */
  printf("\n");
  for (i=1; i<=N; i++) {
    tmp=-v3[i];
    for (j=1; j<=N; j++)
      tmp+=M3[i][j]*v1[j];
    error+=tmp*tmp;
  }
  printf("Numerical Recipes rms error: %e\n",sqrt(error/N));

  for (i=1; i<=N; i++) {
    tmp=-v3[i];
    for (j=1; j<=N; j++)
      tmp+=M3[i][j]*v2[j];
    error+=tmp*tmp;
  }
  printf("LAPACK            rms error: %e\n",sqrt(error/N));
  printf("\n");
}
```

```
MAIN__()
{
  printf("Who am I?  And, why am I here?\n");
  exit(1);
}
```

E LAPACK

If you cannot locate the LAPACK library on your machine, please see
http://www.ccmr.cornell.edu/~muchomas/P480/LAPACK for instructions as to how to proceed.

Please let us (muchomas@ccmr.cornell.edu, freedman@physics.cornell.edu) know if you have any difficulties!

F "octtime.m"

```
for N=[100:100:500, 512, 600:100:1000] %# Loop for sizes N=100, 200, ..., 1000
  N %# Output N to terminal
  A=randn(N,N); %# Random (normally distributed) N x N matrix;
  B=randn(N,N); %# A second random N x N matrix;
  v=randn(N,1); %# A random column vector of length N;

  tic; %# Start an automatic timer
  C=A*B; %# Perform matrix multiplication
  dt=toc; %# End timing, store time interval in dt
          %# <== Place your own code here to display time and MFLOPS rate

  tic; %# Restart automatic timer
  x=A\v; %# Solve linear system (LU decomp and backsubs)
  dt=toc; %# End timing, store time interval in dt
          %# <== Place your own code here to display time and MFLOPS rate

end %# End loop
```