Cornell University

Department of Physics

Phys 480/680, Astro 690 April 9, 2000

Computational Physics, Spring 2000

Homework Assignment # 8

(Due Thursday, April 13 at 8:40 am in Lab.)

Agenda and readings for the remainder of the term

All readings from *Numerical Recipes: The Art of Scientific Computing*, 2nd edition (in Fortran *or* C).

- – Lab6/HW#7, 04/06: Broyden solution; optimization
- – Lec12, 04/04: Optimization, LAPACK; Poisson's equation in multiple dimensions
  – Lab7/HW#8, 04/13: Inner loop optimization; blocking
- – Lec13, 04/11: Fast Fourier Transform (FFT) **Reading: 12.0–12.2, 12.4**
  – Lab8/HW#9, 04/20: Solution of Poisson's equation in multiple dimensions
- – Lec14, 04/18: Schrödinger's equation in multiple dimensions; conjugate-gradient minimization **Reading: 10.0–10.6**;
  – Lab9/HW#10, 04/27: Schrödinger's equation in multiple dimensions
- – Lec15, 04/25: Kohn–Sham equations in multiple dimensions
  – Lab10/HW#11, 05/04: DFT solutions of molecules
- – Lec16, 05/02: Tie up any loose-ends

Goals for this assignment: Develop a feel for computational speeds, learn to code for peak performance and how to call optimized libraries such as LAPACK.

Contents

1 Inner-loop operations **3**
1.1 timer() . . . 3
1.2 Other inner-loop operations . . . 3

2 Basic Linear Algebra Subroutines (BLAS) **4**
2.1 BLAS2 . . . 4
2.2 BLAS3 . . . 4

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

A "timer.c" **7**

B timer() for Windows **10**

C "lapack.c" **12**

D "lapack.c" for Windows **14**

E LAPACK **17**

1 Inner-loop operations

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 problems 1 and 2, you should have the entire 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.

1.1 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.

1.2 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^2 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!

2 Basic Linear Algebra Subroutines (BLAS)

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

2.1 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", 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

2.2 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
  }
}
```

3 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!

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 getting this for you!) Note that the code will run a lot faster if you can use the vendor-supplied BLAS provided there for Pentium/Linux machines.

Finally, if you are compiling under Linux, something like the following compile statement (all on one command line!) will *probably* get the code to compile:

```
cc -O3 -fomit-frame-pointer lapack.c p480.o nrutil.o -o lapack -L... -llapack -lblas -lf2c -lm
```

where "..." represents the full path name of the directory where the "liblapack.a" and "libblas.a "files appears. Don't hesitate to ask for help in getting this code to compile!

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!

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;
  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");
}

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

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 for your machine, please see http://www.ccmr.cornell.edu/~muchomas/P480/LAPAC for instructions as to how to proceed.

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