Cornell University

Department of Physics

Phys 480/680, Astro 690 April 2, 2001

Computational Physics, Spring 2001

Homework Assignment # 6

(Due Friday, March 30 at 8:00 am in Computer Lab.)

Agenda and readings for the week of March 26:

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.

Readings marked NR are from *Numerical Recipes: The Art of Scientific Computing*, 2nd 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 week:

- Lec12, 03/05 [1 hr]: Non-spherical states (higher angular momenta); total energy in density functional theory
  Reading: Sections "Introduction," "Procedure" (you can skip the relativistic part), "Results," and "The LDA approximation," from http://math.nist.gov/DFTdata/atomdata/narrative.html
- Lec13, 03/07 [1 hr]: Forms in the literature for exchange and correlation energy; solution of many non-linear equations in many unknowns (Broyden approaches); our need for a "Modified" Broyden's Method
  Reading: 9.6–9.7
- Lab5, 03/09 [1 hr]: Full, self-consistent solutions for density functional theory of atoms; using gprof; avoiding malloc; inlining optimizations.

Preview of coming week:

- Lec14, 03/26 [1 hr]: Relations among self-consistent iteration, charge mixing, and Broyden's method; line optimization
- Lec15, 03/27 [1 hr]: Full, modified Broyden's method; Introduction to Numerical linear algebra; benefits of LAPACK
- Lab 6, 03/29 [1 hr]: Analysis of convergence of charge mixing; condition numbers; exploration of FLOP rates within inner loops.

Contents

1 Packaging **3**

1.1 "getg" . . . 3

1.1.1 Debugging . . . 3

1.2 "ludcmpp480" and "lubksbp480" . . . 3

1.2.1 Debugging . . . 4

2 Initial Performance **4**
2.1 Convergence Rate . . . 4
2.2 Run time . . . 4

3 Broyden Solver **4**
3.1 Bookkeeping . . . 4
3.1.1 Debugging . . . 5
3.2 Improved initial guess . . . 5
3.3 "broyden" . . . 6
3.3.1 Matrix G and vector u . . . 6
3.3.2 Solution of **G** b = u . . . 7
3.3.3 Formation of final Rho[it+1] . . . 7
3.4 Convergence rate . . . 7

4 Optimization **7**
4.1 Calls to dvector() and free_dvector() . . . 8
4.2 Calls to rk4p480() . . . 9
4.3 Calls to derivs_Schrodinger() . . . 10
4.4 Calls to exc() and excp() . . . 10

5 Uranium **10**

6 Inner-loop operations **11**
6.1 timer() . . . 11
6.2 Other inner-loop operations . . . 11

7 Basic Linear Algebra Subroutines (BLAS) **12**
7.1 BLAS2 . . . 12
7.2 BLAS3 . . . 12

A "test_getg.c" **14**

B "test_lu.c" **16**

C "timer.c" **18**

D timer() for Windows **21**

1 Packaging

1.1 "getg"

```
double getg(double g[], double Rho[],
 double Z, int lmax, int nmax[], int nmaxmax, double **F,
 double r[], double dr[], int N)
```

Return value:

- Etot: total energy of atom

Output:

- g[k]: $\vec{f}(\vec{n}) - \vec{n}$ at the grid points r[k]

Input:

- Rho[k]: $4\pi r^2 n(r[k])$ for all grid points r[k]
- Z: nuclear charge
- lmax: maximum angular momentum to consider (l=0..lmax)
- nmax[l]: maximum number of nodes sought for angular momentum l
- nmaxmax: maximum of all nmax[l]
- F[l][n]: shell occupancies ($F_{n\ell} \equiv (2\ell+1) f_{n\ell}$)
- r[], dr[], N: standard grid information

Using the pieces of your main routine from the Homework #6, construct a subroutine of the above prototype, which takes Rho[] as input, and then computes from it the total potential V=Vnuc+phi+Vxc, the wave functions Psi[][][], and the density corresponding to those wave functions, Rhonew[]. Finally, the routine should put the difference Rhonew[]-Rho[] in g[] and return the value Etot.

Hint: Note that getg() will have to allocate space for many variables such as Vxc[], phi[], Psi[][][]. You will find the *Numerical Recipes* utilities dvector, dmatrix, d3tensor very useful for this. Now that you are allocating this space in a subroutine, it is *essential* that you *deallocate* everything before you exit the routine.

1.1.1 Debugging

The main program in Appendix A solves for the total energy of the *oxygen* atom using your getg() routine. It initializes with zero density, Rho[]=0, and then iterates Rho_{i+1}[]=Rho[]+alpha*g[], as discussed in class, with alpha=0.25. This program should converge to a value of -74.473076 hartree in about 30 iterations.

1.2 "ludcmpp480" and "lubksbp480"

```
void ludcmpp480(double **a, int n, int *indx, double *d);

void lubksbp480(double **a, int n, int *indx, double b[]);
```

Output:

- a[n][m]: LU decomposition of A ("permuted")
- n: size of matrix
- index[n]: permutation of LU decomposition
- d: sign of determinant of A

Input:

- a[n][m]: matrix elements, A_{nm}

- n: size of matrix

As discussed in lecture, our new algorithm for solving large systems of non-linear equations requires us to solve small systems of linear equations of the form $\mathbf{G}\vec{b} = \vec{u}$. *Numerical Recipes* provides two routines, "ludcmp" and "lubksb", which together solve such equations. Generalize them to double precision by making the replacements "float"→"double" and "vector"→"dvector", and rename them with our standard p480 extension.

1.2.1 Debugging

Run your "ludcmpp480" and "lubksbp480" routines with the main program in Appendix B. This test builds a $6 times 6$ linear system, uses the LU routines to solve it, and then tests the result. You may use this main program as a model of how to use the LU routines to solve the linear system $\mathbf{G}\vec{b} = \vec{u}$ in 3.3.2.

2 Initial Performance

The focus of this problem set is enhancing the performance of your atomic solver. To get a sense of the impact of improvements which we make, please gather the following information.

2.1 Convergence Rate

As each iteration, "test_getg" prints both the total energy and the difference of this total from the final converged result. (If your code converges to a slightly different result, simply change the print statement to subtract your final value.)

So that we may assess the effectiveness of the algorithm, make plots to show the grader and to bring to lab of (a) the error as a function of iteration number and (b) the log (preferably, base 10) of the error as a function of iteration number.

2.2 Run time

Write down to show to the grader and at lab (a) how many iterations and (b) how many seconds "test_it.c" requires to reach convergence to all twelve digits printed to the output. (For our code and machine, this was 53 iterations and 26 seconds.)

3 Broyden Solver

You are now ready to use a better algorithm to speed your code by an order of magnitude! Implementing the new algorithm requires attention to detail, but is well worth it in the end.

Programming strategy: We are also now at a stage in our software project where our programming strategy must change. This is because our software (Appendix A) is now fully functional, albeit not optimized. Rather than building up the working pieces one at a time and carefully verifying each one, we now can make incremental changes and rerun the code with each change to make sure that it continues to function properly. For the exercises below, please begin with the main program in Appendix A and make the appropriate modifications to it, calling the new program "test_it".

3.1 Bookkeeping

Important note: The *Numerical Recipes* routines for linear systems are all "one based" (all array and matrix elements begin with one, not zero). The simplest way to work with this is to build all of our linear algebra objects (such as $\mathbf{G}$) also as one based and to keep our physics grid objects (such as phi) as zero based. This mixed strategy allows us to use the *Numerical Recipes* routines with minimal modification and requires no changes to our preexisting code.

The new algorithm requires that we keep track of the charge densities Rho_i and their residuals g_i from all previous iterations. Accordingly, we shall require two dimensional arrays (dmatrix's) for their storage in the main program.

To do this, modify the main program to store the Rho_i and g_i as it goes along, producing the new program "test_it". An appropriate code fragment is

```
#define Itmx 100
            .
            .
  Rho=dmatrix(1,Itmx+1,0,Nmx);
  g=dmatrix(1,Itmx+1,0,Nmx);
            .
            .
  for (it=1; it<=Itmx; it++) {
    Etot=getg(g[it],Rho[it],Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
           it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[it+1][k]=Rho[it][k]+alpha*g[it][k];
  }
            .
            .
  free_dmatrix(Rho,1,Itmx+1,0,Nmx);
  free_dmatrix(g,1,Itmx+1,0,Nmx);
```

Note how we begin the matrices Rho and g with it=1 so that we can conform to the one-based routines from *Numerical Recipes*. Also, because the final iteration (Itmx) needs space for Rho[it+1], we declare space for 1..Itmx+1 in the dmatrix statements.

3.1.1 Debugging

Verify that "test_it" produces output identical to "test_getg".

3.2 Improved initial guess

The basis of the Broyden algorithm is linearization of the equations, which requires that we start not too far from the initial solution. Presently we begin with the very poor starting point Rho[k]=0. To improve upon this, we will start the Broyden solver with the density Rho after five iterations of our present charge-mixing algorithm. To accomplish this, *add* the following loop to "test_it" just before the main iteration loop (which starts with "for (it=1; it¡=Itmx; it++)"):

```
  for (it=1; it<=5; it++) {
    Etot=getg(g[1],Rho[1],Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
           it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[1][k]=Rho[1][k]+alpha*g[1][k];
  }
```

Rerun your "test_it" to again verify that it produces the same output. At this stage, this is a trivial reorganization of the code, but it is important to ensure that everything is working before we implement the modified Broyden algorithm.

3.3 "broyden"

The modified Broyden algorithm discussed in class is:

1. Trial step —
   - Form trial step $\mathrm{Rho}_{it+1} = \mathrm{Rho}_{it} + \alpha \mathrm{g}_{it}$
2. Prediction step —
   - Form matrix

$$\mathbf{G} = \begin{pmatrix} \vec{g}_1 \cdot g_1 & \cdots & \vec{g}_1 \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ \vec{g}_2 \cdot g_1 & \cdots & \vec{g}_2 \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ & \vdots & & \vdots \\ \vec{g}_{it+1} \cdot g_1 & \cdots & \vec{g}_{it+1} \cdot \vec{g}_{it+1} & -\frac{1}{2} \\ 1 & \cdots & 1 & 0 \end{pmatrix}$$

   - Form vector

$$\vec{\mathrm{u}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

   - Solve

$$\mathbf{G}\vec{\mathrm{b}} = \vec{\mathrm{u}}$$

   - Form final prediction

$$\mathrm{Rho}_{it+1} \longleftarrow \sum_{itp=1}^{it} \mathrm{b}_{itp}\mathrm{Rho}_{itp} + \mathrm{b}_{it+1}\mathrm{Rho}_{it+1} \tag{1}$$

Important note: This formulation of the algorithm uses Rho_{it+1} and g_{it+1} in two different ways: to store the trial step and residual (which we called Rho' and g' in lecture) *and* to store the final prediction step. This becomes important for coding only at the stage where we form the final Rho_{it+1}. At the final step, you should first multiply the contents of Rho_{it+1} by b_{it+1}, and then loop over all of the preceding Rho_{itp}, multiplying by b_{itp} and accumulating the final result into Rho_{it+1}.

Beginning with your code "test_it", make a new program "broyden" with the modifications below.

3.3.1 Matrix G and vector u

Declarations: In the above algorithm, **G** is an (it+2) × (it+2) matrix. Thus, at the top of your main program, you should declare space for **G** as a one-based, (Itmx+2) × (Itmx+2) dmatrix and also declare space for u as a one-based dvector of length Itmx+2. Finally, be good and free these variables at the end of main().

Residual vector $\vec{g}_{it+1}$: Once the trial Rho[it+1] is computed, you must also compute the corresponding g[it+1] with a call such as

```
getg(g[it+1],Rho[it+1],Z,lmax,nmax,nmaxmax,F,r,dr,N);
```

Formation: After forming Rho[it+1] and g[it+1] in the main iteration loop, add code to your main program to compute the matrix **G** and the vector u. A suitable code fragment (not the most efficient, but quite workable) for computing the $\vec{g}_{itp} \cdot \vec{g}_{itpp}$ components of **G** is

```
for (itp=1; itp<=it+1; itp++)
  for (itpp=1; itpp<=it+1; itpp++) {
    G[itp][itpp]=0.; /* Trust no one */
    for (k=0; k<=N; k++)
      G[itp][itpp]+=g[itp][k]*g[itpp][k];
  }
```

Note that you will also have to set the final column G[..][it+2]=-0.5, the final row G[it+2][..]=1, and the bottom corner element G[it+2][it+2]=0 to complete the matrix. Finally, be sure to properly set the values of u[..] as described in the algorithm above.

3.3.2 Solution of G b = u

Use your modified numerical recipes routines from 1.2.1 to solve the matrix equation G*b=u for b, taking the program in Appendix B as a model. Note that, for this, you will have to declare (and free!) a dmatrix LU with the same dimensions as G, a dvector b with the same dimensions as u, an ivector indx with the same dimensions as b, and a dummy double variable sgn. For debugging purposes, we highly recommend that you include the "Testing G*b=u?" part of the code from the appendix.

Note that, at this stage, your code should still output the same energies because we have not yet put the final prediction in Rho_{it+1}. Finally, once you are certain that the linear solver is functioning properly, comment out the test G*b=u part.

3.3.3 Formation of final Rho[it+1]

Use your elements of b[itp] from the lu solver to form the final value of Rho[it+1] according to the formula (1). (See the note in Section 3.3 for hints on how to do this.)

This completes the implementation of the modified Broyden algorithm! You now should be able to reach complete convergence within about ten Broyden steps!!!

Hint: For larger atoms, the modified Broyden approach sometimes induces NaN errors. This happens because some the coefficients b[itp] can become negative and the formation of Rho[it+1] sometimes results in very small, negative densities (on the order of -1e-12). To prevent these negative densities from causing NaNs in your exc() routines, it is best to simply take the absolute value after forming Rho[it+1] with a loop like

```
for (k=0; k<=N; k++)
  Rho[it+1][k]=fabs(Rho[it+1][k]);
```

3.4 Convergence rate

Make a new plot of the log of the error versus iteration number to show the TA and to bring to lab.

4 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

4.1 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!!!

4.2 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.

% time	cumulative seconds	self seconds	calls	self ms/call	total 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().

4.3 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().

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

4.4 Calls to exc() and excp()

The next most common calls are to excp() 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.

5 Uranium

Finally, 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.;
```

6 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 C 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 6 and 7, 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 C with the routine in Appendix D.

6.1 timer()

Compile the program in Appendix C (replacing timer() with Appendix D 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.

6.2 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!

7 Basic Linear Algebra Subroutines (BLAS)

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

7.1 BLAS2

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

$$v_2[i] = \sum_k v_1[j] M_1[j][i]. \tag{2}$$

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

7.2 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
  }
}
```

A "test_getg.c"

```
#define Nmx 40000
#define Itmx 100
#define alpha 0.25
main()
{
  /* Change of variables info */
  double *r,*dr;
  int N;

  /* Physics variables */
  double *Rho,*g,Z;
  int lmax,*nmax,nmaxmax;
  double **F;
  double Etot;

  /* Working variables */
  int n,l,k;
  double x;

  /* Solver iteration variables */
  int it;
  /* Value of pi */
  const double pi=4.*atan(1.);

  /* Specs for O */
  Z=8.;
  lmax=1;

  nmax=ivector(0,lmax);
  nmax[0]=1;
  nmax[1]=0;

  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[1][0]=4.;

  /* The rest is now general for ANY case */
  Rho=dvector(0,Nmx);
  g=dvector(0,Nmx);

  /* Grid vectors */
  r=dvector(0,Nmx);
  dr=dvector(0,Nmx);

  N=4000;
  /* Set up grid */
```

```
  for (k=0; k<=N; k++) {
    x=((double) k)/((double) N);
    r[k]=1/(1-x)-1-x;
    dr[k]=1/(1-x)/(1-x)-1;
  }
  dr[N]=0.;

  /* Initialize charge density */
  for (k=0; k<=N; k++)
    Rho[k]=0.;

  /* Iteration loop */
  for (it=1; it<=Itmx; it++) {
    Etot=getg(g,Rho,Z,lmax,nmax,nmaxmax,F,r,dr,N);
    printf("Iter:%4d  Etot:%20.12f  Error:%20.12f\n",
   it,Etot,fabs(Etot-(-74.473076803203738)));

    for (k=0; k<=N; k++)
      Rho[k]=Rho[k]+alpha*g[k];
  }

  /* Be a good citizen and clean up... */
  free_dvector(r,0,Nmx);
  free_dvector(dr,0,Nmx);

  free_dmatrix(F,0,lmax,0,nmaxmax);
  free_dvector(Rho,0,Nmx);
  free_dvector(g,0,Nmx);
}
```

B "test_lu.c"

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#include "p480.h"

#define N 4
main()
{
  double **G,**LU,*b,*u;

  int *indx;
  double sgn,tmp;

  int i,j;

  /* Declare space */
  G=dmatrix(1,N,1,N); /* Matrix */
  LU=dmatrix(1,N,1,N); /* Space for LU decomposition of matrix */
  b=dvector(1,N); /* Space for answer to G*b=u */
  u=dvector(1,N); /* Space for right-hand side, u */
  indx=ivector(1,N); /* Space for integer vector needed by ludcmp */

  /* Make matrix Gij=1/(i+j) */
  for (i=1; i<=N; i++)
    for (j=1; j<=N; j++)
      G[i][j]=1./( (double) i+j );

  /* Make right-hand side u=(1 1 1 ...) for equation G*x=u */
  for (i=1; i<=N; i++) /* u=0 for all but the last component */
    if (i<N)
      u[i]=0.;
    else
      u[i]=1.;


  /* Print out the problem */
  printf("\nSolving...\n\n");
  for (i=1; i<=N; i++) {
    printf("| ");
    for (j=1; j<=N; j++)
      printf("%9.2e ",G[i][j]);
    printf("|   |b[%d]|  =  | %9.2e |\n",i,u[i]);
  }

  /* Solve G*b=u */
  for (i=1; i<=N; i++) /* Copy G into LU because ludcmp destroys its input */
    for (j=1; j<=N; j++)
      LU[i][j]=G[i][j];
  for (i=1; i<=N; i++) /* Copy u into b because lubksb destroys its input */
    b[i]=u[i];
```

```
  /* Actual solution of equations */
  ludcmpp480(LU,N,indx,&sgn); /* Get LU decomposition */
  lubksbp480(LU,N,indx,b); /* Use LU decomposition to solve equations */

  /* Does it work? */
  printf("\nTesting G*b=u?:\n");
  for (i=1; i<=N; i++) {
    tmp=0.;
    for (j=1; j<=N; j++)
      tmp+=G[i][j]*b[j];
    printf("%20.12f  =%20.12f  ?\n",tmp,u[i]);
  }
  printf("\n");

  /* Be good, and clean up! */
  free_dmatrix(G,1,N,1,N); /* Matrix */
  free_dmatrix(LU,1,N,1,N); /* Space for LU decomposition of matrix */
  free_dvector(b,1,N); /* Space for answer to G*b=u */
  free_dvector(u,1,N); /* Space for right-hand side, u */
  free_ivector(indx,1,N); /* Space for integer vector needed by ludcmp */
}
```

C "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); */
}
```

D 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;
}
```