Cornell University

Department of Physics

Phys 480/680, Astro 690 April 15, 2002

Computational Physics, Spring 2002

Homework Assignment # 9

(Due Thursday, April 25 at 8:40 am in Computer Lab.)

Agenda and readings for the weeks of April 16 & 23:

Goals: Solution of single particle Schrödinger equation using *unconstrained* minimization techniques; minimization using steepest descents, line minimization, preconditioning and conjugate gradients; solution of the harmonic oscillator in $d = 3$ dimensions.

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:

- Lec20, 04/09 (Tue: mini-lecture, 40mins): C++ continued, $d = 3$ DFT and IDFT
- Lab7, 04/09 (Tue): SKIPPED
- Lec21, 04/11 (Thu): Aliasing of G-vectors in the DFT, eigenvalue problems in $d > 1$ dimension through variational principles.

Preview of coming weeks:

- Lec22, 04/16 (Tue: mini-lecture, 25mins): Variational principle for Schrödinger's equation, analytic continuation for treatment of constraints.
- Lab8, 04/16 (Tue): Solving Poisson's equation in $d > 1$ dimensions.
- Lec23, 04/18 (Thu): DFT++ representation of Schrodinger's equation; numerical minimization, steepest descents, line-minimization, preconditioning.
  Reading: NR 10.0,10.5
- Lec24, 04/23 (Tue: **regular lecture only**): Conjugate gradients
  Reading: NR 10.0,10.6
- Lec25, 04/25 (Thu: minilecture, 25mins): Kohn-Sham equations in $d > 1$ dimensions
- Lab9, 04/25 (Thu): Solution of Schrodinger's equation in $d > 1$ dimensions
- Lec26, 04/30 (Tue): FFT
  Reading: NR 12.0,12.1,12.2

Contents

1 Infrastructure **3**
1.1 Idag() and Jdag() . . . 3
1.2 vector-vector product: pvector '*' pvector . . . 3
1.3 sho() . . . 3

2 Physics and steepest descents: sch.C **5**
2.1 Gathering data . . . 6

3 Line minimization **7**
3.1 Theory of line minimization . . . 7
3.2 Implementation . . . 7
3.3 Debugging . . . 7
3.4 Gathering data . . . 7

4 Conjugate gradients **8**
4.1 Debugging . . . 8
4.2 Gathering data . . . 8

5 Preconditioning **9**
5.1 Implementation: K() . . . 9
5.2 Debugging and Gathering data . . . 9
5.3 Debugging . . . 9
5.4 Gathering data . . . 9

1 Infrastructure

Going beyond Poisson's equation to the solution of Schrödinger's equation introduces a few new operators which we must code into the pvector class. These all have forms similar to the operators we have already coded, and so may be coded by copying the similar routines and editing.

1.1 Idag() and Jdag()

```
friend pvector Idag(const pvector& in1)
friend pvector Jdag(const pvector& in1)
```

Return value:

- Idag: action of complex-conjugate of discrete Fourier transform on in1. (Code is just like I() but sends "-1" to fournp480 instead of "1".)
- Jdag: action of complex-conjugate of inverse discrete Fourier transform on in1. (Code is just like J() but sends "1" to fournp480 instead of "-1".)

Input:

- in1: a pvector

Add the above two operators to the definition of the pvector class.

Note: Along with L(), O(), I(), and J(), this completes the set of operators needed to solve any set of non-linear, partial differential equations with spatially local non-linear couplings. This includes the Kohn-Sham equations of density-functional theory as well as problems in fluid mechanics and a wide range of other problems in the physics sciences.

1.2 vector-vector product: pvector '*' pvector

```
friend pvector operator*(const pvector& in1,const pvector& in2)
```

Return value:

- $\left(\mathbf{Diag}\,\vec{\text{in1}}\right)\vec{\text{in2}}$: a pvector, each of whose elements is the product of the corresponding element of in1 with the corresponding element of in2; i.e., out.dat[i][j][k]=in1.dat[i][j][k]*in2.dat[i][j][k] for all i,j,k.

Input:

- in1, in2: pvectors

Add a function of the above prototype to the pvector class.

Note: This algebraic function gives a short-hand for the operation $\left(\mathbf{Diag}\,\tilde{V}\right)\vec{\psi}$ (which appears in density functional theory) so that we do not have to actually form the (very large!) diagonal matrix $\left(\mathbf{Diag}\,\tilde{V}\right)$. With this notation, we can now code the above operation as simply "V*psi".

1.3 sho()

```
Complex sho(d3vec r,double k)
```

Return value:

- sho: Value of the potential energy from a three-dimensional spring –

$$\frac{1}{2}k|\vec{r}|^2$$

Input:

- k: spring constant
- r: d3vec with components r.x, r.y, r.z

In the next problem, we will solve for the ground state of the simple harmonic oscillator. Provide the above function (*not* as part of the pvector class definition!) to your program so that it may be used with evalr() in your main() program to create the potential pvector V.

2 Physics and steepest descents: sch.C

Write a program called sch.C based upon the code fragment below (and your physics.C program from HW#9) to solve Schrödingers's equation for the three-dimension harmonic oscillator. Use the following parameters:

- n=16 (size of FFT grid)
- d=0.75 bohr (grid spacing)
- k=0.25 hartree/bohr^2 (spring constant)

Hint: In quantum mechanics, the analytic solution for the energy of the three-dimensional harmonic oscillator is $(3/2)\hbar\omega$ where $\omega = \sqrt{k/m}$. In our units $\hbar = m = 1$, so the expected result is 3/4 hartree. With the above resolution (d) and box size (n), the numerical result which you should find in the end is $E = 0.749\,999\,895\,138\,61$. (The fact that d and E are equal is just a coincidence in this problem. You can try running with different d's and see how the results for E change.) After about 400 iterations of the steepest descent algorithm (with alpha=0.0004 as defined below), you should reach an energy within about 1 micro-hartree of this final result.

Note: The algebraic results derived/to be derived in lecture for the energy and gradient are

$$\begin{aligned} E &= \frac{N}{D} \\ N &= \int \left\{ u^*(\vec{x}) \left(-\frac{1}{2}\nabla^2 u(\vec{x}) \right) + V(\vec{x}) u^*(\vec{x}) u(\vec{x}) \right\} dV \\ &= -\frac{1}{2}\vec{u}^\dagger \mathbf{L}\vec{u} + \vec{u}^\dagger \mathcal{I}^\dagger \left(\mathbf{Diag}\, \tilde{V} \right) \mathcal{I}\vec{u} \\ D &= \int \left(u^*(\vec{x}) u(\vec{x}) \right) dV \\ &= \vec{u}^\dagger \mathcal{O}\vec{u} \end{aligned}$$

$$\nabla_{\vec{u}} E = \frac{1}{D}\left(-\frac{1}{2}\mathbf{L}\vec{u} + \mathcal{I}^\dagger \left(\mathbf{Diag}\, \tilde{V} \right) \mathcal{I}\vec{u} - E\mathcal{O}\vec{u} \right),$$

where

$$\tilde{V} \equiv \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \vec{V}$$

and $\vec{V}$ are the sample values of the potential on the real-space grid points, $V_i = V(\vec{x}_i)$.

Code fragment:

```
                          .
                          .
                          .
const int Itmx=400;
main() {
                          .
                          .
                          .
  // Open file to store convergence results
  FILE *fp;
  if ( (fp=fopen("conv.dat","w"))==NULL ) {
    printf("Unable to open file!\n");
    exit(1);
  }
                          .
                          .
```

```
                        .
  n=16; d=0.75;  // Parameters for the calculation
                        .
                        .
                        .
  V.evalr(sho,0.25); // SHO potential stored in pvector V

  Vtilde= ... // See formula above

  // Initialize with normalized, but otherwise random, wave function
  u.eval(rnd); // Random elements
  u=u*(1./sqrt(u^O(u))); // Normalize u for starting
                        .
                        .
                        .
  alpha=4e-4; // Step size for going down the gradient
  for (int i=0; i<Itmx; i++) {
    // Compute energy and gradient for wave function u
    Denom= ... // See formula for ``D'' above.  Hint: don't forget to use real()
    Num= ... // See formula for ``N'' above.  Hint: use definition of Vtilde
    E=Num/Denom;
    grad= ...

    // Output:
    //   iteration number, energy, length of u, cosine of angle between grad and last dir
    printf("%4d %20.15f %e %e\n",
   i,E,sqrt(Denom),real(grad^dir)/sqrt(real(grad^grad)*real(dir^dir))
   );
    fprintf(fp,"%4d %20.15f %e %e\n",
   i,E,sqrt(Denom),real(grad^dir)/sqrt(real(grad^grad)*real(dir^dir))
   );
    fflush(fp);

    // Choice of search direction (opposite to the gradient, for now...)
    dir=(-1.)*grad;

    // Put code for computing best alpha between here...

    // ... and here when you do Part 3.2.


    u=u+alpha*dir; // Take a step of size alpha down along the search direction
    printf("         ===> alpha=%e\n",alpha); // Remind us of step size

  }

}
```

2.1 Gathering data

When you run your program, it will create a file called "conv.dat". For future reference and plotting, copy this to a file called "convSD.dat" for *steepest descents*. Make a plot of $\log(E - 0.749\,999\,895\,138\,61)/\log(10.)$ (log base ten of the error) as a function of iteration number.

3 Line minimization

If you wish, you may experiment with the value of alpha in part 2 to see if you can obtain somewhat better convergence. This part of the homework implements an algorithm which ensures that you always take just the right step to minimize the energy as much as possible at each iteration.

3.1 Theory of line minimization

The best choice is to pick alpha to give the minimum possible energy along the search direction dir. Mathematically, we wish to minimize the function

$$E(\alpha) \equiv E(\vec{u} + \alpha\vec{d}),$$

where $\vec{d}$ represents the search direction dir and we write α for alpha. Clearly, this function is a minimum when the function

$$E'(\alpha) = \vec{d} \cdot \nabla_{\vec{u}} E(\vec{u})$$

has the value zero.

One simple, but effective approach to finding a good estimate of the zero of $E'(\alpha)$ is to expand this function near $\alpha = 0$,

$$E'(\alpha) \approx E'(0) + E''(0)\alpha \Longrightarrow \alpha \approx -\frac{E'(0)}{E''(0)}$$

and then to estimate $E''(0)$ by a simple finite difference

$$E''(0) = \frac{d}{d\alpha}E'(0) \approx \frac{E'(\alpha_t) - E'(0)}{\alpha_t},$$

where α_t is some small "trial" value for α.

Putting this all together gives the formula

$$\alpha = -\alpha_t \frac{\vec{d} \cdot \nabla E(\vec{u})}{\vec{d} \cdot \nabla E(\vec{u} + \alpha_t\vec{d}) - \vec{d} \cdot \nabla E(\vec{u})} \tag{1}$$

3.2 Implementation

Using the value alphat=1e-5 for α_t, put code which implements Eq. (1) at the location indicated in the code for this problem.

Hints:

- Eq. (1) requires $\nabla E(\vec{u}+\alpha_t d)$, the value of the gradient evaluated for the vector $\vec{u}+\alpha_t d$. You may wish to define a pvector ut=u+alphat*dir, and then replicate the code your wrote previously for computing N, D and the gradient, calling the new result gradt instead of grad.
- Dot products like $\vec{d}\cdot\nabla E(\vec{u})$ are equal to the real part of the "dagger product", $\vec{d}^\dagger \nabla E(\vec{u})$. So your code will have fragments that look something like "real(dir^gradt)-real(dir^grad)".

3.3 Debugging

Run your new code. When your steepest descents is working properly, you will find that (after a few initial steps) the fourth column of the output is generally small, something on the order of 1e-2. Also, you will find that, as you approach the minimum, the above expansion for $E''(\alpha)$ gets better and better, so that the fourth column should continue getting closer to zero as the algorithm converges.

3.4 Gathering data

For future reference and plotting, copy the convergence data file ("conv.dat") to a file called "convLM.dat" for *line minimization*. Plot the convergence ($\log(E - 0.749\,999\,895\,138\,61)/\log(10.)$ as a function of iteration number) of both convSD.dat and convLM.dat *on the same graph* to see the improvement.

4 Conjugate gradients

The method of conjugate-gradients (*Numerical Recipes*, Sec. 10.6) can make dramatic improvements in the convergence.

While the theory behind conjugate-gradients is subtle, the implementation is straight-forward. The theory states that at step n, rather than choosing the search direction ($\vec{d}_n$) to be along the down-hill gradient

$$\vec{d}_n = -\nabla E(\vec{u}_n),$$

one should search along a direction which is a linear combination of the previous search direction (d_{n-1}) and the down-hill gradient,

$$\vec{d}_n = \beta \vec{d}_{n-1} - \nabla E(\vec{u}_n).$$

The "magic" linear-combination coefficient β is

$$\beta \equiv \frac{\nabla E(\vec{u}_n) \cdot \nabla E(\vec{u}_n)}{\nabla E(\vec{u}_{n-1}) \cdot \nabla E(\vec{u}_{n-1})}.$$

Because conjugate-gradients depends upon being relatively near a minimum, it pays to first do a few (say ten) standard steps of line-minimization first. Thus, you should replace the statement

```
dir=(-1.)*grad;
```

in your code with the following code fragment

```
if (i<=10)
  dir=(-1.)*grad;
else {
  beta= ...
  dir=beta*dir+(-1.)*grad;
}
```

where beta is defined as above.

Hint: Don't forget to compute dot products by taking the real part of the corresponding "dagger product".

4.1 Debugging

When your conjugate-gradients is working, you should reach *machine precision* in about 80 iterations!

4.2 Gathering data

Copy the convergence data file to a file called "convCG.dat" for *conjugate gradients*. Plot the convergence ($\log(E - 0.749\,999\,895\,138\,61)/\log(10.)$ as a function of iteration number) of convSD.dat, convLM.dat, and convCG.dat *on the same graph* to see the improvements you have made!

5 Preconditioning

Yet further gains may be made by using a technique known as preconditioning. "Preconditioning" basically means applying an operator to your gradient in order to make the search direction point more toward the minimum. To implement this, the only changes to your code are in the formation of your search direction.

If we call the preconditioning operator K(), then the only changes you need to make appear in the code fragment below:

```
if (i<10)
  dir=(-1.)*K(grad);
else {
  beta= ...
  dir=beta*dir+(-1.)*K(grad);
}
```

The formula for β for preconditioned conjugate-gradients now becomes

$$\beta \equiv \frac{\nabla E(\vec{u}_n) \cdot \mathbf{K} \nabla E(\vec{u}_n)}{\nabla E(\vec{u}_{n-1}) \cdot \mathbf{K} \nabla E(\vec{u}_{n-1})}.$$

Make these changes for preconditioning to your main() program.

5.1 Implementation: K()

```
friend pvector K(const pvector& in1)
```

Return value:

- Result of applying preconditioner to in1: out.dat[i][j][k]=in1.dat[i][j][k]/(1.+ $|\vec{G}|^2$)

Input:

- in1: pvector

To implement preconditioning, add a function of the above prototype to the definition of the pvector class.

You may model K() on the Linv function. The only differences between Linv() and K() are that, now, (a) 1. is added to $|\vec{G}|^2$ in the denominator to keep things finite for $\vec{G} = 0$, and (b) you shouldn't set the result at $\vec{G} = 0$ to zero.

Hint if you run into trouble: a good test for debugging is that if you just have K() return in1, then you should recover your original conjugate-gradient results.

5.2 Debugging and Gathering data

5.3 Debugging

When your preconditioned conjugate-gradients is working, you should reach approximately machine precision in about 50 iterations!!!

5.4 Gathering data

Copy the convergence data file to a file called "convPCCG.dat" for *preconditioned conjugate gradients*. Plot the convergence ($\log(E - 0.749\,999\,895\,138\,61)/\log(10.)$ as a function of iteration number) of convSD.dat, convLM.dat, convCG.dat and convPCCG.dat *all on the same graph* to see the improvements you have made!

Have this final plot ready to show the grader to get credit for this assignment.