Cornell University

Department of Physics

Phys 480/680, Astro 690 April 8, 2002

# Computational Physics, Spring 2002

## Homework Assignment # 8

(Due Tuesday, April 16 at 8:40 am in Computer Lab.)

## Agenda and readings for the week of April 9:

**Goal:** Learn to use overloading to express scientific algorithms succinctly; solution of Poisson's equation in a single line of code:

```
phi=I(Linv((-4*pi)*O(J(rho))));
```

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:

- Lec18, 04/02 (Tue: mini-lecture, 25mins): Solution of Poisson's equation in $d > 1$ dimension in a single line of expressive software!
- Lab6, 04/02 (Tue): Fully optimized atomic software, FLOP rates of common operations
- Lec19, 04/04 (Thu): Implementation of expressive software for Poisson's equation: plane wave basis and coding in C++

Preview of coming week:

- Lec20, 04/09 (Tue: mini-lecture, 40mins): C++ continued, $d = 3$ DFT and IDFT
- Lab7, 04/09 (Tue): Timing computer FLOP performance for BLAS/LAPACK operations
- Lec21, 04/11 (Thu): Solution of Schrödinger's equation in three-dimensions, steepest descents, line-minimization
  **Reading: NR 10.0,10.5**
- Lec22, 04/16 (Tue: mini-lecture, 25mins): preconditioning
- Lab8, 04/16 (Tue): Solving Poisson's equation in $d > 1$ dimensions

# Contents


**1 Infrastructure** **3**
- 1.1 Working with classes . . . 3
  - 1.1.1 Using classes in the standard C++ library . . . 3
  - 1.1.2 Working with user defined classes: "vec.C" . . . 3
- 1.2 "pvector" class . . . 4
  - 1.2.1 C3tensor() and free_C3tensor() . . . 4
  - 1.2.2 sum() . . . 4
  - 1.2.3 operator-(), operator*() . . . 5
  - 1.2.4 I() . . . 5
  - 1.2.5 J() . . . 6
  - 1.2.6 O() and Linv() . . . 6

**2 Physics** **6**
- 2.1 evalr() . . . 6
  - 2.1.1 Debugging . . . 7
- 2.2 Potential from a sum of two Gaussian distributions . . . 7
  - 2.2.1 GaussR() . . . 7
  - 2.2.2 Computation of total electrostatic potential energy U . . . 8
  - 2.2.3 Computation of potential field $\phi(\vec{x})$ . . . 9

**A "complex.C"** **11**

**B "vec.C"** **12**

**C "pvector.C"** **13**

**D main() for "physics.C"** **20**

# 1 Infrastructure

To be able to solve Poisson's equation as a single line of code

```
phi=I(Linv((-4*pi)*O(J(rho))));
```

requires two capabilities not common in C programs. First, in the above expression, the calls to functions such as J(..) return not single numbers but entire arrays. Second, the expression (-4*pi)*O(..) represents not a simple scalar multiplication but rather multiplication of the scalar (-4*pi) with the *entire* array returned by O(..). We therefore need the ability to work with *objects* like arrays as easily as we work with numbers in c.

C++ supports these capabilities with exactly the same syntax as c, so that, if you know c, then programming in C++ (at least the kind we do in scientific programming) is *fairly* straightforward. The only tricky part is that you must explicitly program how your different objects behave.

For this first problem, we give you with some of this infrastructure as a guide for you to build the rest. This will take some time, but once it is working, it will make the rest of our work in the semester much nicer.

## 1.1 Working with classes

Each object in C++ belongs to a particular *class*. A particular double-precision complex variable "z", for instance, belongs to the "complex<double>" class.

### 1.1.1 Using classes in the standard C++ library

As an example of using classes, the program "complex.C" in Appendix A illustrates the use of the "complex<double>" class. To make sure that you have the C++ compiler working properly, compile and run the program. Note that this program #include's the standard C++ library for complex numbers (so we don't have to program them for ourselves) and, for convenience, makes "Complex" a short-hand for "complex<double>".

Note that the complex class is used just like any of the familiar variable types int, float, double. The one nice, extra feature is that the standard library includes a handy mechanism for *construct*ing a Complex number whenever we need one (such as $0 + 2i$ on the second line of main()).

To practice working with such classes, write a subroutine with the prototype

```
Complex cmult(Complex a,Complex b);
```

so that when you uncomment the statement "z2=cmult(..);" in "complex.C", the program produces the same output as before. In writing this subroutine, you should find that all the familiar rules of working with float and double apply!

### 1.1.2 Working with user defined classes: "vec.C"

The power of C++ is that we can program and define our own classes, and they will work just like int, float and double! Two very simple, but handy classes that we shall make for ourselves are i3vec and d3vec for three-vectors of integers and doubles, respectively.

The code for these classes appears in Appendix B. Note that each object of these classes has three members, x, y, and z, so that for a given vector "v", we can access its components as "v.x", "v.y" and "v.z". These two classes are so simple that we actually do not have to provide much code. We, however, do provide our own, handy constructors so that "i3vec(1,2,3)" and "d3vec(2.,4.,5.) construct the appropriate three-vectors.

Compile and check that the program "vec.C" in Appendix B runs. Next, write a subroutine with the prototype

```
d3vec scalarmult(double s,i3vec i);
```

which returns a double vector whose components are just the variable s times each of the components of i. Finally, uncomment the statement "dv=scalarmult(3.,iv);" in "vec.C" and verify that its output is correct.

## 1.2 "pvector" class

In expressions like "phi=I(Linv((-4*pi)*O(J(rho))));" we think of phi and rho as vectors of the values of the potential $\phi(\vec{x}_i)$ and charge density $\rho(\vec{x}_i)$ at a set of sample points $\{\vec{x}_i\}$. For our Fourier representation the data in these "vectors" are actually stored on a three-dimensional mesh of points. We shall call the class which represents these *physics vectors* the "pvector" class.

Appendix C contains most of the specification of the pvector class, along with a main program to test it. This section will teach you the basics of making such a class by having you complete its specification.

### 1.2.1 C3tensor() and free_C3tensor()

```
Complex ***C3tensor(long nrl,long nrh,long ncl,long nch,long ndl,long ndh);

void free_C3tensor(Complex ***t,
   long nrl,long nrh,long ncl,long nch,long ndl,long ndh);
```

Return value:

- C3tensor: pointer to three dimensional array of Complex

Input:

- nrl, nrh: lowest and highest values of indices for first index (rows)
- ncl, nch: lowest and highest values of indices for second index (columns)
- ndl, ndh: lowest and highest values of indices for third index (depth)

Write routines of the above prototypes. You should be able to do this with a simple replacement of "Complex" for every occurrence of "double" in your d3tensor routines.

**C++ Note#1:** The code in the appendix assumes that you have compiled nrutil.c with a standard c (not C++) compiler. This is what the statement at the top of pvector.C

```
extern "C"
{
#include "nrutil.h"
}
```

tells the compiler. If your nrutil.c is compiled with the C++ compiler, then remove the 'extern "C" {}' parts of the statement and use just '#include "nrutil.h"'.

**C++ Note#2:** Many compilers treat programs whose names end in ".c" as standard c and programs ending in ".C" as C++.

**Debugging** Place your C3tensor routines at the indicated location in the program "pvector.C" from Appendix C. You now should be able to compile and run the program for various box dimensions. At this stage, the main program allocates space for 3 pvectors (v1, v2, v3), fills them in with random numbers, computes the magnitude of each ($\vec{v}^\dagger\vec{v} \equiv$ v^v), and finally tests the mathematical identity that $\vec{v}_1^\dagger\vec{v}_2 = (\vec{v}_2^\dagger\vec{v}_1)^*$.

### 1.2.2 sum()

```
friend Complex sum(const pvector& in)
```

Return value:

- Sum of all complex components of in

Input:

- in: a pvector

Write a function of the above prototype using the code for "operator^" as a guide. You should place your new routine in "pvector.C" *within* the definition of the pvector class (immediately after the definition of "operator^" is a good location). **Debug** by uncommenting the "Test sum" code in main() and compiling and running.

**C++ Notes:** The modifier "friend" simply means that this function has access to all of the internal data of the objects upon which it acts. Also, for your coding, you can ignore the "const" and "&" for the input argument, and imagine we had declared simply sum(pvector in). The "const" and "&" are optimizations which tell the compiler not to make unnecessary copies of in.

### 1.2.3 operator-(), operator*()

```
friend pvector operator-(const pvector& in1,const pvector& in2);
friend pvector operator*(Complex c,const pvector& in1);
friend pvector operator*(const pvector& in1,Complex c);
```

Return value:

- operator-: in1-in2
- operator*: c*in1

Input:

- in1, in2: pvectors
- c: Complex number

Using the code provided for operator+() as a guide, write functions of the above prototype, again placing them *within* the definition of the pvector class. These are the functions that the compiler actually calls when it sees expressions like "4.*rho+phi" involving pvectors rho and phi. Note that you need to write two routines for multiplication, one for c*in1 and the other for in1*c because, from a compiler point of view, these really are distinct operations. **Debug** by uncommenting the tests for subtraction, scalar-pvector products and the algebra of the inner product.

### 1.2.4 I()

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

Return value:

- pvector I: Discrete (forward) Fourier Transform (DFT) of in1

Input:

- in1: pvector

Modify the *Numerical Recipes* FFT routine "fourn" by replacing all occurrences of "float" with "double", and rename the routine "fournp480". Then, uncomment the routine "friend pvector I(..)" in pvector.C, and **debug** with the "Simple test of I" in the main program.

**Numerical Recipes Note:** For debugging, keep in mind that *Numerical Recipes* requires that all FFT's be done with boxes with dimensions that are integer powers of two (1, 2, 4, 8, 16, etc.).

**C++ Note:** I found it easiest to compile fournp480 as a standard c code, thus it is declared as an extern "C" function at the top of pvector.C. You should change this if you decide to compile fournp480 as a C++ code.

### 1.2.5 J()

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

Return value:

- pvector J: Inverse Discrete Fourier Transform (DFT) of in1

Input:

- in1: pvector

Using the code provided in the appendix for I(..), write a function of the above prototype to perform the inverse Fourier transform. To **debug**, uncomment the "Tests of J and I" and run the program.

**Hint:** To get J() to be the inverse of I(), don't forget the normalization by $1/(N_x N_y N_z)$ where the box dimensions are accessible through in1.N.x, in1.N.y and in1.N.z. Also, don't forget to switch the sign of $i$ in the transform!

### 1.2.6 O() and Linv()

```
friend pvector O(const pvector& in1)
friend pvector Linv(const pvector& in1)
```

Return value:

- pvector O: Application of overlap matrix to in1
- pvector Linv: Application of inverse of L to in1

Input:

- in1: pvector

Uncomment the function L() in pvector.C and use it as a guide to construct functions of the above prototypes, recalling that $\mathcal{O} = V\mathbf{1}$, where $V$ is the volume of the cell and $\mathbf{1}$ is the identity matrix, and that Linv divides all components by $|\vec{G}|^2$, except for $\vec{G} = \vec{0}$, which it multiplies by zero. Note that the code provided for L() already takes proper account of aliasing, as discussed in lecture. To **debug**, uncomment the tests for O and L, and run the program.

# 2 Physics

Keep the present version of your pvector.C program, and make a new copy called physics.C. Replace main() in physics.C with the main() program in Appendix D. Add the new capabilities below to this physics.C program.

## 2.1 evalr()

```
void evalr(Complex (*func)(d3vec,double),double dparam)
```

Input:

- func: function to be evaluated for all grid points
- dparam: parameter to be passed to the function

Output:

- dat[i][j][k]: set to values of func(r,dparam) evaluated on real-space grid-point r corresponding to [i][j][k], wrapping the periodic boundary conditions around the point i=j=k=0.

Add a function of the above prototype to the definition of the pvector class; this utility will make our job of doing physics simpler.

**Important:** "Wrapping" around i=j=k=0 means that, for all grid points, you always take the closest periodic image to the origin. For instance, i=N-1 is the point just *before* the end of the array (i=N), which by periodic boundary conditions is equivalent to i=0. Thus, we interpret i=N-1 as the point x=-1*d, where d is the grid spacing. Thus, in general, to map points [i][j][k] to vectors in space, we use the following translation:

| i,j,k | 0 | 1 | 2 | ... | $N-3$ | $N-2$ | $N-1$ |
|---|---|---|---|---|---|---|---|
| $x, y, z$ | 0 | $d$ | $2d$ | ... | $-3d$ | $-2d$ | $-d$ |

### 2.1.1 Debugging

```
Complex constant(d3vec r,double dconst)
Complex planewave(d3vec r,double G)
```

Output:

- constant: for all values of r, returns the same value, dconst as a Complex number, Complex(dconst,0.)
- planewave: returns exp(i*G*r.x)

To debug evalr() above and to test that your transforms represent the correct physics, add the two functions above to your physics.C program (*not* as part of the pvector class, but as regular functions). Then, compile and run the program. Note how the program uses the statement "ones.evalr(constant,1.)" to fill the pvector "ones" with the value of 1. using your constant() function!

Next, based on the test for G=(0,0,0) provided in main(), use your planewave() function to write your own test for G=(1,0,0).

## 2.2 Potential from a sum of two Gaussian distributions

### 2.2.1 GaussR()

```
Complex GaussR(d3vec r,double sigma)
```

Output:

- GaussR: the value of an origin-centered, *normalized* Gaussian of root-mean-square (rms) width sigma in each coordinate direction

$$\left(\frac{1}{\sqrt{2\pi}\,\sigma}\right)^3 e^{-\frac{r.x^2+r.y^2+r.z^2}{2\sigma^2}}$$

Input:

- sigma: rms width
- r: d3vec with components r.x, r.y, r.z

To study the electrostatic potential from Gaussian charge distributions add a function of the above prototype to your physics.C code.

**Debugging:** Your Gaussian functions should contain one unit of charge. To debug GaussR, add the following code fragment (with appropriate initializations) to main() in physics.C

```
rho1.evalr(GaussR,1.);
rho2.evalr(GaussR,2.);
rho=rho1-rho2;
printf("Integral of rho1: (%f,%f)\n",sum(rho1)*V/N);
printf("Integral of rho2: (%f,%f)\n",sum(rho2)*V/N);
printf("Integral of rho: (%f,%f)\n",sum(rho)*V/N);
```

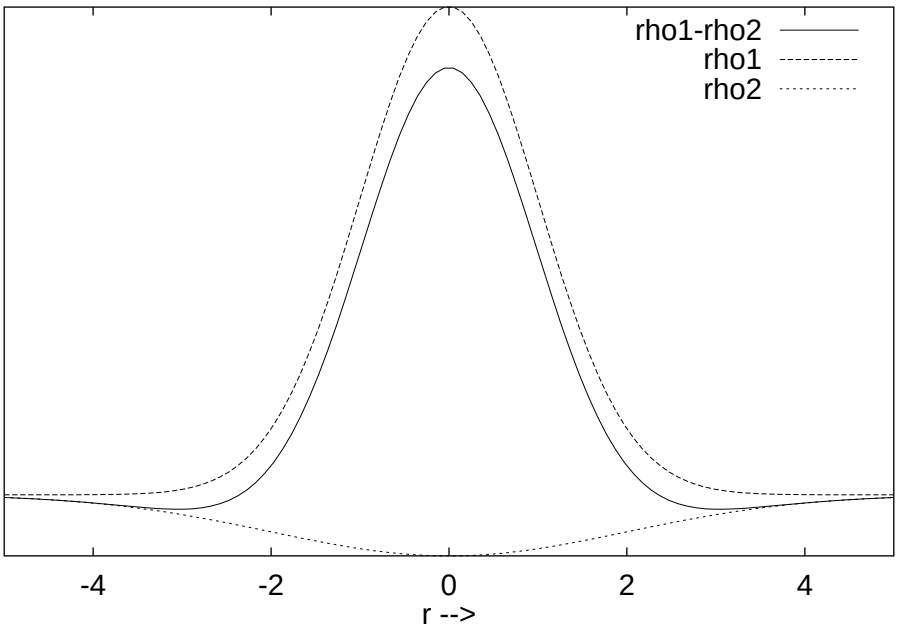



Figure 1: Distribution of charges for three-dimensional solution of Poisson's equation

With an FFT box of size n=32 and grid spacing of d=1 (bohrs), you should get the correct total charges to better than 0.000001 (electron charges).

### 2.2.2 Computation of total electrostatic potential energy U

We now solve Poisson's equation for the Gaussian charge distributions above. Because the energy of a periodic system is infinite if each unit cell contains a net charge, we will consider the charge distribution rho=rho1-rho2 from part 2.2.1. This net distribution has one unit of positive charge in a peak of width $\sigma = 1$ and one unit of negative charge in a more spread-out peak of width $\sigma = 2$, as in Figure 1.

The total electrostatic energy of any distribution of charges is

$$U = \frac{1}{2} \int \rho(\vec{x})^* \phi(\vec{x})\, dV. \tag{1}$$

Note that, because $\rho(\vec{x})$ is real, the complex conjugation '*' does not do anything. We include it for convenience so that we may use the $\mathcal{O}$ and $\mathcal{L}$ operators as defined in lecture.

If we have the coefficients, $\hat{\rho}_\alpha$ and $\hat{\phi}_\alpha$, for the expansions of these fields

$$\begin{aligned} \rho(\vec{x}) &= \sum_\alpha \hat{\rho}_\alpha b_\alpha(\vec{x}) \\ \phi(\vec{x}) &= \sum_\alpha \hat{\phi}_\alpha b_\alpha(\vec{x}), \end{aligned}$$

then, substituting into Eq. (1) we find (in the notation from lecture),

$$U = \frac{1}{2} \hat{\vec{\rho}}^\dagger \mathcal{O} \hat{\vec{\phi}}.$$

As discussed in lecture, we can always find the vectors of expansion coefficients from vectors of sample values by using our inverse transform operator,

$$\hat{\vec{\rho}} = \mathcal{J} \vec{\rho}$$

$$\hat{\vec{\phi}} \;=\; \mathcal{J}\vec{\phi}.$$

Thus our final result for the potential energy is

$$U = \frac{1}{2}(\mathcal{J}\vec{\rho})^{\dagger}\mathcal{O}(\mathcal{J}\vec{\phi}) \tag{2}$$

After the check of the integrals of rho1, rho2 and rho (from part 2.2.1) in physics.C, have your code determine and print out the total potential energy of the total charge-density distribution rho using the one-line solution of Poisson's equation from class

```
phi=I(Linv((-4*pi)*O(J(rho))));
```

and the computational representation of Eq. 2 for U,

```
U=0.5*real(J(rho)^O(J(phi)));
```

Note that we need to take the real part of the complex result of the '^' operator so that we can store the answer in U, a double precision value.

Compare your computed result for U with the analytic result for the total potential energy of a sum of two Gaussian distributions, one of width $\sigma_1$ with positive unit charge and one of width $\sigma_2$ with negative unit charge:

$$U = \frac{1}{\sqrt{\pi}}\left(\frac{1}{2}\left(\frac{1}{\sigma_1}+\frac{1}{\sigma_2}\right) - \frac{\sqrt{2}}{\sqrt{\sigma_1^2+\sigma_2^2}}\right)$$

**Debugging:** For n=64 and d=0.5, you should find agreement to 12 significant figures!

### 2.2.3 Computation of potential field $\phi(\vec{x})$

```
Complex ErfR(d3vec r,double sigma)
```

Output:

- ErfR: $\mathrm{erf}\,(r/(\sqrt{2}\sigma))/r$

Input:

- sigma: rms width
- r: d3vec with components r.x, r.y, r.z

**Important note for Windows users:** The standard error function $\mathrm{erf}\,x$ is not part of the math library for Visual Studio C++. You may down-load the source code for the erf() function from the course web site at http://www.ccmr.cornell.edu/~muchomas/P480/s_erf.c.

The analytic solution to Poisson's equation

$$\nabla^2\phi(\vec{x}) = -4\pi\rho(\vec{x})$$

for a Gaussian charge distribution of width $\sigma$

$$\rho_\sigma(r) = \left(\frac{1}{\sqrt{2\pi}\,\sigma}\right)^3 e^{-\frac{r^2}{2\sigma^2}},$$

where $r \equiv |\vec{x}|$, is

$$\phi_\sigma(r) = \frac{\mathrm{erf}\,(\frac{r}{\sqrt{2}\,\sigma})}{r}.$$

Write a function of the above prototype to evaluate this function.

To directly compare your computational solution for phi in part 2.2.2 with the analytic solution, use evalr and ErfR to construct pvectors phi1 and phi2 containing the analytic solutions corresponding to the

charge distributions rho1 and rho2. The analytic solution for rho=rho1-rho2 is then phianal=phi1-phi2. (Remember the principle of superposition!)

As discussed in class, your computational solution, phi, will differ from the analytic result phianal by an arbitrary constant shift in the potentials. Use the following code fragment to (a) compute the shift, (b) adjust the potentials by adding the correct shift, and (c) evaluate the remaining error:

```
pvector error(i3vec(n,n,n),d3vec(d,d,d));

shift=real(sum(phi-phianal)/N);
error=phi-phianal-shift*ones;

printf("Average shift between phi and phianal: %e\n",shift);
printf("rms error in phi: %e\n",(error^error)/N);
```

**For fun:** If you want to "see" your solutions, add the following statements to the end of your program

```
rho.printr("rho");
phi.printr("phi");
phianal.printr("phianal");
```

This will make three files that you can plot: "rho," "phi" and "phianal." The three columns in each file are (1) distance from the origin, and (2) the real and (3) imaginary parts of either rho, phi or phianal.

**Challenge physics question:** You will notice that the shift scales inversely with the volume of the cell. Can anyone compute *analytically* what the shift *should* be?

# A "complex.C"

```
/* Include standard C++ complex library */
/* And, make short-hand "Complex" for complex<double> */
#include <complex.h>
typedef complex<double> Complex;

/* C-style I/O functions */
#include <stdio.h>

main()
{
  Complex z1,z2;

  z1=Complex(0.,2.);
  z2=z1*Complex(1.,2.);
  // z2=cmult(z1,Complex(1.,2.));
  printf("(%f,%f)\n",z2);
}
```

## B "vec.C"

```
/* Include standard C++ complex library */
/* And, make short-hand "Complex" for complex<double> */
#include <complex.h>
typedef complex<double> Complex;

/* C-style I/O functions */
#include <stdio.h>

class i3vec {
 public:
  int x,y,z;
  // Default constructor
  i3vec() {};

  // Copy constructor: compiler provides automatically
  // Copy assignment: compiler provides automatically
  // Destructor: compiler provides automatically
  // Constructor
  i3vec(int i,int j, int k) {
    x=i; y=j; z=k;
  }
};

class d3vec {
 public:
  double x,y,z;
  // Default constructor
  d3vec() {};

  // Copy constructor: compiler provides automatically
  // Copy assignment: compiler provides automatically
  // Destructor: compiler provides automatically
  // Constructor
  d3vec(double i,double j, double k) {
    x=i; y=j; z=k;
  }
};

main()
{
  i3vec iv;
  d3vec dv;

  iv=i3vec(1,2,3);
  dv=d3vec(2.,4.,6.);

  //   dv=scalarmult(3.,iv);

  printf("iv=(%d,%d,%d)\n",iv);
  printf("dv=(%f,%f,%f)\n",dv);
}
```

# C "pvector.C"

```
/* Libraries needed by various routines */
#include <stdio.h>
#include <stdlib.h>

/* Include standard C++ complex library */
/* And, make short-hand "Complex" for complex<double> */
#include <complex.h>
typedef complex<double> Complex;

/* Make C functions in nrutil.h accessible to C++ */
extern "C"
{
void fournp480(double data[], unsigned long nn[], int ndim, int isign);
#include "nrutil.h"
}

/* Global definition of pi */
const double pi=4.*atan(1.);

/* Include your C3tensor routines between here ... */
/* ... and here! */


/* Simple three-vector classes */
class i3vec {
 public:
  int x,y,z;
  // Constructor
  i3vec() {};

  // Copy constructor
  // Copy assignment
  // Destructor
  // Initializer
  i3vec(int i,int j, int k) {
    x=i; y=j; z=k;
  }
};

class d3vec {
 public:
  double x,y,z;
  // Constructor
  d3vec() {};

  // Copy constructor
  // Copy assignment
  // Destructor
  // Initializer
  d3vec(double i,double j, double k) {
    x=i; y=j; z=k;
```

```
  }
};

/* The pvector class */
class pvector {
  struct i3vec N;
  struct d3vec d;
 public:
  Complex ***dat;

  // Constructor
  pvector(i3vec iN, struct d3vec id) {
    int i,j,k;

    N.x=iN.x; d.x=id.x;
    N.y=iN.y; d.y=id.y;
    N.z=iN.z; d.z=id.z;

    dat=C3tensor(0,N.x-1,0,N.y-1,0,N.z-1);
  }

  // Copy constructor
  pvector(const pvector& in) {
    int i,j,k;

    N.x=in.N.x; d.x=in.d.x;
    N.y=in.N.y; d.y=in.d.y;
    N.z=in.N.z; d.z=in.d.z;

    dat=C3tensor(0,N.x-1,0,N.y-1,0,N.z-1);

    for (i=0; i<N.x; i++)
      for (j=0; j<N.y; j++)
for (k=0; k<N.z; k++)
  dat[i][j][k]=in.dat[i][j][k];
  }

  // Copy assignment: cleanup and copy
  pvector& operator=(const pvector& in) {
    int i,j,k;

    for (i=0; i<N.x; i++)
      for (j=0; j<N.y; j++)
for (k=0; k<N.z; k++)
  dat[i][j][k]=in.dat[i][j][k];
  }

  // Destructor
  ~pvector() {
    free_C3tensor(dat,0,N.x-1,0,N.y-1,0,N.z-1);
  }

  // Print out as 3 dimensional array
```

```
  print3d(char filename[]) {
    FILE* ostream;
    if ( (ostream=fopen(filename,"w"))==NULL ) {
      printf("Unable to open file '%s'!\n",filename);
      exit(1);
    }

    int i,j,k;
    for (i=0; i<N.x; i++) {
      fprintf(ostream,"%4d ----------------------------------------------n",i);
      for (j=0; j<N.y; j++) {
for (k=0; k<N.z; k++)
  fprintf(ostream,"(%9f,%9f) ",
 real(dat[i][j][k]),imag(dat[i][j][k]));
fprintf(ostream,"\n");
      }
    }
    fprintf(ostream,"\n");
  }

  // Print out as radial data
  printr(char filename[]) {
    FILE* ostream;
    if ( (ostream=fopen(filename,"w"))==NULL ) {
      printf("Unable to open file '%s'!\n",filename);
      exit(1);
    }

    // Loop over entire array
    for (int i=0; i<N.x; i++)
      for (int j=0; j<N.y; j++)
for (int k=0; k<N.z; k++) {
  // Wrap box around so end of array is "near" origin
  // (box dimensions are in an i3vec called "N")
  int ii = ( i<N.x/2 ? i : i-N.x);
  int jj = ( j<N.y/2 ? j : j-N.y);
  int kk = ( k<N.z/2 ? k : k-N.z);

  // Print distance to origin and data
  fprintf(ostream,"%e %e %e\n",
 sqrt( (ii*d.x)*(ii*d.x)+(jj*d.y)*(jj*d.y)+(kk*d.z)*(kk*d.z) ),
 dat[i][j][k]);
}

  }

  // vector Hermitian inner product
  friend Complex operator^(const pvector& in1,const pvector& in2) {
    Complex out;

    // Initialize sum
    out=Complex(0.,0.);
    // Loop over all components, doing sum for dagger operation
```

```
    // (box dimensions are in both i3vec's, "in1.N" or "in2.N")
    for (int i=0; i<in2.N.x; i++)
      for (int j=0; j<in2.N.y; j++)
for (int k=0; k<in2.N.z; k++)
  out+=conj(in1.dat[i][j][k])*in2.dat[i][j][k];
    return out;
  }

  // vector addition
  friend pvector operator+(const pvector& in1,const pvector& in2) {
    // Create a pvector called "out" which is a copy of pvector "in1"
    pvector out(in1);

    // Loop over all components, adding in2 onto out
    // (which already contains in1)
    for (int i=0; i<in2.N.x; i++)
      for (int j=0; j<in2.N.y; j++)
for (int k=0; k<in2.N.z; k++)
  out.dat[i][j][k]+=in2.dat[i][j][k];
    return out;
  }

  // Initialize according to a function
  eval(Complex (*func)()) {
    for (int i=0; i<N.x; i++)
      for (int j=0; j<N.y; j++)
for (int k=0; k<N.z; k++)
  dat[i][j][k]=func();
  }

  // I (forward) transform
/*   friend pvector I(const pvector& in) { */
/*     pvector out(in); */
/*     long unsigned int dim[4]; */

/*     dim[1]=in.N.x;  */
/*     dim[2]=in.N.y; */
/*     dim[3]=in.N.z;  */

/*     fournp480((double *)&out.dat[0][0][0]-1, dim, 3, 1); */
/*     return out; */
/*   } */

/*   // L  */
/*   friend pvector L(const pvector& in) { */
/*     // Get nornalization constants for G-vectors */
/*     d3vec g=d3vec( */
/*    2.*pi/(in.N.x*in.d.x), */
/*    2.*pi/(in.N.y*in.d.y), */
/*    2.*pi/(in.N.z*in.d.z) */
/*    ); */
/*     d3vec g2=d3vec( */
/*     g.x*g.x, */
```

```
/*     g.y*g.y, */
/*     g.z*g.z */
/*     ); */

/*     // Creat space for output and copy input into it     */
/*     pvector out(in); */

/*     // Calculate volume of unit cell */
/*     double V=(in.N.x*in.d.x)*(in.N.y*in.d.y)*(in.N.z*in.d.z); */

/*     // Loop over entire box */
/*     for (int i=0; i<in.N.x; i++) */
/*       for (int j=0; j<in.N.y; j++) */
/*  for (int k=0; k<in.N.z; k++) { */
/*    // Treat aliasing properly!! */
/*    int ii = ( i<in.N.x/2 ? i : i-in.N.x); */
/*    int jj = ( j<in.N.y/2 ? j : j-in.N.y); */
/*    int kk = ( k<in.N.z/2 ? k : k-in.N.z); */

/*    out.dat[i][j][k]*=-V*(g2.x*(ii*ii)+g2.y*(jj*jj)+g2.z*(kk*kk)); */
/*  } */

/*     return out; */
/*   } */
};

// Function to return random, complex numbers
Complex rnd() {
  return Complex(
 ((double)rand())/RAND_MAX-0.5,
 ((double)rand())/RAND_MAX-0.5);
}

main() {
  int n; /* Dimensions of FFT box */
  double d=1.; /* Grid spacing in real space */

  // Scan problem size and store in N
  printf("n: "); scanf("%d",&n);

  printf("Testing on %dx%dx%d grid ...\n",n,n,n);

  // Initialize
  pvector v1(i3vec(n,n,n),d3vec(d,d,d));
  pvector v2(i3vec(n,n,n),d3vec(d,d,d));
  pvector v3(i3vec(n,n,n),d3vec(d,d,d));

  // Fill in with random data
  v1.eval(rnd);
  v2.eval(rnd);
  v3.eval(rnd);

  // Test that random filling is working
```

```
  printf("Ok to be real, but better not be all zeros!\n");
  printf("    |v1|^2=(%f,%f)\n",v1^v1);
  printf("    |v2|^2=(%f,%f)\n",v2^v2);
  printf("    |v3|^2=(%f,%f)\n\n",v3^v3);

  // "Random" complex number for testing
  Complex c1=Complex(2.,3.);


  printf("These SHOULD be ALL zeros!!!\n");
  // Test (v1^v2)=conj(v2^v1))
  printf("    Hermitian inner product is Hermitian: (%15.12f,%15.12f)\n",
   (v1^v2)-conj(v2^v1));

/*   // Test sum(v1+v2)=sum(v1)+sum(v2) */
/*   printf("    sum() is a linear operation:          (%15.12f,%15.12f)\n", */
/*   sum(v1+v2)-sum(v1)-sum(v2)); */

/*   // Test pvector subtraction */
/*   printf("    subtraction test:                     (%15.12f,%15.12f)\n", */
/*   sum(v1-v2)-(sum(v1)-sum(v2))); */

/*   // Test pvector products */
/*   printf("    scalar-pvector product test:          (%15.12f,%15.12f)\n", */
/*   sum(c1*v1)-c1*sum(v1)); */
/*   printf("    pvector-scalar product test:          (%15.12f,%15.12f)\n", */
/*   sum(v1*c1)-c1*sum(v1)); */

/*   // Test algebra of inner product */
/*   printf("    Hermitian inner product is linear:    (%15.12f,%15.12f)\n",  */
/*    (v1^(v2+c1*v3))-((v1^v2)+c1*(v1^v3)) */
/*    ); */
/*   printf("    Hermitian inner product is co-linear: (%15.12f,%15.12f)\n\n", */
/*    ((v2+c1*v3)^v1)-((v2^v1)+conj(c1)*(v3^v1)) */
/*    ); */

/*   // Simple test of I */
/*   v3=I(v1+c1*v2)-(I(v1)+c1*I(v2)); */
/*   printf("    I operator is linear:                 (%15.12f,%15.12f)\n", */
/*   v3^v3); */

/*   // Tests of J and I */
/*    v3=J(I(v1))-v1; */
/*    printf("    J is inverse of I:                    (%15.12f,%15.12f)\n", */
/*    v3^v3); */

/*    v3=I(J(v1))-v1; */
/*    printf("    I is inverse of J:                    (%15.12f,%15.12f)\n", */
/*    v3^v3); */

/*    printf("    Unitarity:                            (%15.12f,%15.12f)\n", */
/*    (v1^I(v2))/conj(v2^J(v1))-n*n*n); */
```

```
/*    // Test of O */
/*    v3=O(v1)-(pow(n*d,3)*v1); */
/*    printf("    Test of O:                              (%15.12f,%15.12f)\n", */
/*    v3^v3); */

/*    // Test of Linv */
/*    v1.dat[0][0][0]=Complex(0.,0.); /* Zero out G=0 component! */ */
/*    v3=L(Linv(v1))-v1; */
/*    printf("    Test of Linv:                           (%15.12f,%15.12f)\n", */
/*    v3^v3); */

}
```

# D main() for "physics.C"

```
main() {
  int n,N;
  double d,V,U,shift;

  // Scan grid size (n) and grid spacing (d)
  printf("n:  "); scanf("%d",&n);
  printf("dx: "); scanf("%lf",&d);

  // Compute total number of points in and volume of cell
  N=n*n*n; V=pow(n*d,3);
  printf("%d^3 grid of size %f^3=%lf (bohr^3)...\n",n,n*d,V);

  // Initialize physics vectors on n^3 grid with spacing d in each direction
  pvector ones(i3vec(n,n,n),d3vec(d,d,d));
  pvector v1(i3vec(n,n,n),d3vec(d,d,d));
  pvector v2(i3vec(n,n,n),d3vec(d,d,d));
  pvector v3(i3vec(n,n,n),d3vec(d,d,d));

  // Set pvector ones to all 1's.
  ones.evalr(constant,1.);

  // Does b_{G=0}(x_i)=ones?
  v1.evalr(constant,0.); // Set v1=all zeros
  v1.dat[0][0][0]=Complex(1.,0.); // Set G=[0][0][0] component to 1

  v2=I(v1)-ones; // Compute difference
  printf("b_{G=(0,0,0)}(x_i)=ones?  %e\n",v2^v2); // Print magnitude of error

}
```