APPENDIX E

USEFUL PROGRAMS

THIS appendix contains a few programs and functions that are used in the main text. All of these are also available for download in the on-line resources.

E.1 GAUSSIAN QUADRATURE

The function `gaussxw` below calculates sample points and weights for Gaussian quadrature. The points are defined on the standard interval $[-1,1]$. To use the function you would write, for example "`x,w = gaussxw(N)`". The function returns two floating-point arrays of N elements each with `x` containing the positions of the sample points x_k and `w` containing the weights w_k such that $\sum_k w_k f(x_k)$ is the N-point Gaussian approximation to the integral $\int_{-1}^{1} f(x)\,\mathrm{d}x$. To perform integrals over any other domain $[a,b]$ both the positions and the weights must be transformed according to

$$x'_k = \tfrac{1}{2}(b-a)x_k + \tfrac{1}{2}(b+a), \qquad w'_k = \tfrac{1}{2}(b-a)w_k. \tag{E.1}$$

A second function `gaussxwab` is provided which calls the first to calculate the positions and weights then performs the transformation for you and returns arrays `x` and `w` for any interval $[a,b]$ that you specify. To use this function you would write "`x,w = gaussxwab(N,a,b)`". See Section 5.5 for further discussion and examples.

File: `gaussxw.py`

```
from numpy import ones,copy,cos,tan,pi,linspace

def gaussxw(N):

    # Initial approximation to roots of the Legendre polynomial
    a = linspace(3,4*N-1,N)/(4*N+2)
    x = cos(pi*a+1/(8*N*N*tan(a)))
```

```
    # Find roots using Newton's method
    epsilon = 1e-15
    delta = 1.0
    while delta>epsilon:
        p0 = ones(N,float)
        p1 = copy(x)
        for k in range(1,N):
            p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx = p1/dp
        x -= dx
        delta = max(abs(dx))

    # Calculate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

    return x,w

def gaussxwab(N,a,b):
    x,w = gaussxw(N)
    return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w
```

Copies of these functions can be found in the on-line resources in the file `gaussxw.py`.

E.2 SOLUTION OF TRIDIAGONAL OR BANDED SYSTEMS OF EQUATIONS

The function `banded` below calculates the solution to a system of linear simultaneous equations of the form $\mathbf{A}\mathbf{x} = \mathbf{v}$ for the vector $\mathbf{x}$ when the matrix $\mathbf{A}$ is tridiagonal, or more generally banded, as described in Section 6.1.6. To use it you say `x = banded(A,v,up,down)`, where `A` is an array containing the banded matrix, `v` is the vector on the right-hand side of the equation, and the variables `up` and `down` specify how many nonzero elements there are above and below the diagonal, respectively, in each column of the matrix. More generally, `v` can be a two-dimensional array containing several independent right-hand sides to $\mathbf{A}\mathbf{x} = \mathbf{v}$, each appearing as a separate column of the array.

To save space storing the matrix $\mathbf{A}$—given that most of its elements are zero—the array `A` is not in the usual form of a matrix, but instead contains the diagonals of the matrix along its rows. Suppose, for instance, our banded

matrix was like this:

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & & \\ a_{10} & a_{11} & a_{12} & a_{13} & \\ & a_{21} & a_{22} & a_{23} & a_{24} \\ & & a_{32} & a_{33} & a_{34} \\ & & & a_{43} & a_{44} \end{pmatrix} \tag{E.2}$$

That is, it has two nonzero elements above the diagonal, one below, and four nonzero diagonals in all. We would represent this with a four-row array `A` having elements as follows:

$$\mathtt{A} = \begin{pmatrix} - & - & a_{02} & a_{13} & a_{24} \\ - & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & - \end{pmatrix} \tag{E.3}$$

The values in the elements marked "–" do no matter—you can put anything in these elements and it will make no difference to the results.

The vector **v** is stored in the array `v` in standard form—no special arrangement is used. The function returns a single array with the same length as `v` containing the solution **x** to the equations, or, in the case of multiple right-hand side, an array with the same shape as `v` with the solution for each of the right-hand sides in the corresponding column.

File: `banded.py`

```
from numpy import copy

def banded(Aa,va,up,down):

    # Copy the inputs and determine the size of the system
    A = copy(Aa)
    v = copy(va)
    N = len(v)

    # Gaussian elimination
    for m in range(N):

        # Normalization factor
        div = A[up,m]

        # Update the vector first
        v[m] /= div
        for k in range(1,down+1):
            if m+k<N:
```

```
                v[m+k] -= A[up+k,m]*v[m]

        # Now normalize and subtract the pivot row
        for i in range(up):
            j = m + up - i
            if j<N:
                A[i,j] /= div
                for k in range(1,down+1):
                    A[i+k,j] -= A[up+k,m]*A[i,j]

    # Backsubstitution
    for m in range(N-2,-1,-1):
        for i in range(up):
            j = m + up - i
            if j<N:
                v[m] -= A[i,j]*v[j]

    return v
```

A copy of this function can be found in the file `banded.py` in the on-line resources.

E.3 DISCRETE COSINE AND SINE TRANSFORMS

Functions for performing (complex) discrete Fourier transforms (DFTs) are available in Python in the module `numpy.fft`, but not functions for performing discrete cosine and sine transforms. As discussed in Section 7.5.3, however, the cosine and sine transforms are simply DFTs of data that have a particular symmetry, either even or odd, about the midpoint of their interval. So one can calculate such transforms by first mirroring the data to create the required symmetry and then using the standard DFT functions in `numpy.fft`. The functions given below use this trick to perform both forward and inverse discrete cosine and sine transforms.

File: `dcst.py`

```
from numpy import empty,arange,exp,real,imag,pi
from numpy.fft import rfft,irfft

# 1D DCT Type-II
def dct(y):
    N = len(y)
    y2 = empty(2*N,float)
    y2[:N] = y[:]
    y2[N:] = y[::-1]
```

```
    c = rfft(y2)
    phi = exp(-1j*pi*arange(N)/(2*N))
    return real(phi*c[:N])

# 1D inverse DCT Type-II
def idct(a):
    N = len(a)
    c = empty(N+1,complex)
    phi = exp(1j*pi*arange(N)/(2*N))
    c[:N] = phi*a
    c[N] = 0.0
    return irfft(c)[:N]

# 1D DST Type-I
def dst(y):
    N = len(y)
    y2 = empty(2*(N+1),float)
    y2[0] = y2[N+1] = 0.0
    y2[1:N+1] = y[:]
    y2[N+2:] = -y[::-1]
    return -imag(rfft(y2)[1:N+1])

# 1D inverse DST Type-I
def idst(a):
    N = len(a)
    c = empty(N+2,complex)
    c[0] = c[N+1] = 0.0
    c[1:N+1] = -1j*a
    return irfft(c)[1:N+1]
```

One can then build upon these functions to perform cosine and sine transforms in higher dimensions. Here are functions to perform forward and inverse two-dimensional cosine and sine transforms.

File: `dcst.py`

```
# 2D DCT
def dct2(y):
    M = y.shape[0]
    N = y.shape[1]
    a = empty([M,N],float)
    b = empty([M,N],float)
    for i in range(M):
        a[i,:] = dct(y[i,:])
    for j in range(N):
        b[:,j] = dct(a[:,j])
```

```
    return b

# 2D inverse DCT
def idct2(b):
    M = b.shape[0]
    N = b.shape[1]
    a = empty([M,N],float)
    y = empty([M,N],float)
    for i in range(M):
        a[i,:] = idct(b[i,:])
    for j in range(N):
        y[:,j] = idct(a[:,j])
    return y

# 2D DST
def dst2(y):
    M = y.shape[0]
    N = y.shape[1]
    a = empty([M,N],float)
    b = empty([M,N],float)
    for i in range(M):
        a[i,:] = dst(y[i,:])
    for j in range(N):
        b[:,j] = dst(a[:,j])
    return b

# 2D inverse DST
def idst2(b):
    M = b.shape[0]
    N = b.shape[1]
    a = empty([M,N],float)
    y = empty([M,N],float)
    for i in range(M):
        a[i,:] = idst(b[i,:])
    for j in range(N):
        y[:,j] = idst(a[:,j])
    return y
```

Copies of these functions can be found in the on-line resources in the file `dcst.py`.

E.4 COLOR SCHEMES

As described in Section 4.3, one can change the color scheme—or *colormap* as it is technically known—used in density plots, to make particular plots clearer or more attractive. The `pylab` module defines a range of useful colormaps, some of which are listed in the table in Section 4.3. However, there are some others that are occasionally useful in physics applications that are not included in `pylab`. The following short package defines three additional color maps that I find useful. They are:

`redblue`	Goes from red to blue via black
`redwhiteblue`	Goes from red to blue via white
`inversegray`	Goes from white to black, the opposite of `gray`

Code defining these colormaps is given below and can be found in the on-line resources in the file `colormaps.py`. To use the `redblue` colormap, for example, you would say "`from colormaps import redblue`", then "`redblue()`" when you want to change the colormap. (For the technically minded, you can also import the colormap objects themselves. They are called `cp_redblue`, `cp_redwhiteblue`, and `cp_inversegray`.) The red/blue colormaps are useful for representing hot/cold distinctions and especially positive/negative distinctions in electric potentials. The `inversegray` colormap is sometimes useful when you are going to print out your density plot on paper.

File: `colormaps.py`

```
from matplotlib.colors import LinearSegmentedColormap
from matplotlib.cm import RdBu
from matplotlib.pyplot import set_cmap

cdict = {"red":   [(0.0,1.0,1.0),(0.5,0.0,0.0),(1.0,0.0,0.0)],
         "green": [(0.0,0.0,0.0),(1.0,0.0,0.0)],
         "blue":  [(0.0,0.0,0.0),(0.5,0.0,0.0),(1.0,1.0,1.0)]}
cp_redblue = LinearSegmentedColormap("redblue",cdict)

cp_redwhiteblue = RdBu

cdict = {"red":   [(0.0,1.0,1.0),(1.0,0.0,0.0)],
         "green": [(0.0,1.0,1.0),(1.0,0.0,0.0)],
         "blue":  [(0.0,1.0,1.0),(1.0,0.0,0.0)]}
cp_inversegray = LinearSegmentedColormap("inversegray",cdict)

def redblue():
    set_cmap(cp_redblue)
```

```
def redwhiteblue():
    set_cmap(cp_redwhiteblue)

def inversegray():
    set_cmap(cp_inversegray)
```