Code_Samples

September 23, 2016

1 Python for Scientists, 2nd Edition: Code Samples

In [ ]:

The book contains many valid Python code snippets. Many are trivial, but most of those whose length is at least five lines are available here.

1.1 Chapter 2

1.1.1 §2.2

In [ ]:
```
a = 3
b = 2.7
c = 12 + 5j
s = "Hello World!"
L = [a, b, c, s, 77.77]
```

1.1.2 §2.5

In [ ]:
```
#%%writefile fib.py
# File: fib.py Fibonacci numbers

"""Module fib: Fibonacci numbers
   Contains one function fib(n).
"""

def fib(n):
    """ Returns n'th Fibonacci number. """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a

############################################################

if __name__ == "__main__":
    for i in range(1001):
        print "fib(", i, ") = ", fib(i)
```

1.1.3 §2.5.2

```
In [ ]: #%%writefile gcd.py
        #  File gcd.py  Implementing the GCD Euclidean algorithm.

        """ Module gcd: contains two implementations of the Euclid
            GCD algorithm, gcdr and gcd.
        """

        def gcdr(a, b):
            """ Euclidean algorithm, recursive vers., returns GCD. """
            if b == 0:
                return a
            else:
                return gcdr(b, a%b)

        def gcd(a, b):
            """ Euclidean algorithm, non-recursive vers., returns GCD. """
            while b:
                a, b = b, a%b
            return a

        ##############################################################
        if __name__ == "__main__":
            import fib

            for i in range(963):
                print i, '  ', gcdr(fib.fib(i), fib.fib(i+1))
```

1.2 Chapter 3

1.2.1 §3.1

```
In [ ]: a = 4; b = 5.5; c = 1.5+2j; d = 'a'
        e = 6.0*a - b*b + \
                c**(a+b+c)
        f = 6.0*a - b*b + c**(
                 a+b+c)
        a, b, c, d, e, f
```

1.2.2 §3.2

```
In [ ]: p =3.14
        p
        q = p
        p = 'pi'
        p
        q
```

1.2.3 §3.5.1

```
In [ ]: [1, 4.0, 'a']
        u = [1, 4.0, 'a']
        v = [3.14, 2.78, u, 42]
        v
        len(v)
        len?                    # or help(len)
        v*2
        v + u
        v.append('foo')
        v
```

1.2.4 §3.5.3

```
In [ ]: u = [0, 1, 2, 3, 4, 5, 6, 7]
        su = u[2 : 4]
        su
        su[0] = 17
        su
        u
        u[2 : 4] = [3.14,' a']
        u
```

1.2.5 §3.5.4

```
In [ ]: a = 4
        b = a
        b = 'foo'
        a
        b
        u = [0, 1, 4, 9, 16]
        v = u
        v[2] = 'foo'
        v
        u
```

```
In [ ]: u = [0, 1, 4, 9, 16]
        v = u[ : ]
        v[2] = 'foo'
        v
        u
```

1.2.6 §3.5.7

```
In [ ]: empty={}
        parms = {'alpha':1.3, 'beta':2.74}
        #parms = dict(alpha=1.3,beta=2.74)
```

```
parms['gamma']=0.999
parms
```

1.2.7 §3.7.3

```
In [ ]: y = 107
        for x in range(2, y):
            if y%x == 0:
                print y, " has a factor ",  x
                break
        else:
                print y, "is prime."
```

1.2.8 §3.8.1

```
In [ ]: def add_one(x):
            """ Takes x and returns x + 1. """
            x = x + 1
            return x

        x=23
        #add_one?                    # or help(add_one)
        add_one(0.456)
        x
```

```
In [ ]: def add_x_and_y(x, y):
            """ Add x and y  and return them and their sum. """
            z = x + y
            return x, y, z

        a, b, c = add_x_and_y(1,0.456)
        a, b, c
```

```
In [ ]: L = [0, 1, 2]
        print id(L)
        def add_with_side_effects(M):
            """ Increment first element of list. """
            M[0] += 1

        add_with_side_effects(L)
        print L
        id(L)
```

```
In [ ]: L = [0, 1, 2]
        id(L)
        def add_without_side_effects(M):
            """ Increment first element of list. """
            MC = M[ : ]
            MC[0] +=1
```

```
        return MC

    L = add_without_side_effects(L)
    L
    id(L)
```

1.2.9 §3.8.4

In [ ]:
```
def average(*args):
    """ Return mean of a non-empty tuple of numbers. """
    print args
    sum = 0.0
    for x in args:
        sum += x
    return sum/len(args)

print average(1,2,3,4)
print average(1,2,3,4,5)
```

1.2.10 §3.9

In [ ]:
```
#%%writefile frac.py
# File: frac.py

import gcd

class Frac:
    """ Fractional class.  A Frac is a pair of integers num, den
        (with den!=0) whose GCD is 1.
    """

    def __init__(self, n, d):
        """ Construct a Frac from integers n and d.
            Needs error message if d = 0!
        """
        hcf=gcd.gcd(n, d)
        self.num, self.den = n/hcf, d/hcf

    def __str__(self):
        """ Generate a string representation of a Frac. """
        return "%d/%d" % (self.num,self.den)

    def __mul__(self, another):
        """ Multiply two Fracs to produce a Frac. """
        return Frac(self.num*another.num, self.den*another.den)

    def __add__(self,another):
        """ Add two Fracs to produce a Frac. """
```

```
            return Frac(self.num*another.den + self.den*another.num,
                        self.den*another.den)

        def to_real(self):
            """ Return floating point value of Frac. """
            return float(self.num)/float(self.den)

    if __name__ == "__main__":
        a=Frac(3,7)
        b=Frac(24,56)
        print "a.num= ", a.num, ", b.den= ", b.den
        print a
        print b
        print "floating point value of a is ", a.to_real()
        print "product= ",a*b,", sum= ",a + b
```

1.2.11 §3.11

```
In [ ]: def sieve_v1(n):
            """
               Use Sieve of Eratosthenes to compute list of primes <= n.
               Version 1
            """
            primes = range(2, n+1)
            for p in primes:
                if p*p > n:
                    break
                product = 2*p
                while product <= n:
                    if product in primes:
                        primes.remove(product)
                    product += p
            return len(primes), primes
```

```
In [ ]: def sieve_v2(n):
            """
                Sieve of Eratosthenes to compute list of primes <= n.
                Version 2.
            """
            sieve = [True]*(n+1)
            for i in xrange(3, n+1, 2):
                if i*i > n:
                    break
                if sieve[i]:
                    sieve[i*i : : 2*i] = [False]*((n - i*i) // (2*i) + 1)
            answer = [2] + [i for i in xrange(3, n+1, 2) if sieve[i]]
            return len(answer), answer
```

1.3 Chapter 4

1.3.1 §4.1.2

```
In [ ]: import numpy as np
        xl = np.linspace(0, 0.5, 5, endpoint=False)
        xm = np.linspace(0.5, 0.6, 10,endpoint=False)
        xr = np.linspace(0.6, 1.0, 5)
        xs = np.hstack((xl, xm, xr))
        xs
```

1.3.2 §4.1.5

```
In [ ]: def m1(x):
            return np.exp(2*x)
        def m2(x):
            return 1.0
        def m3(x):
            return np.exp(1.0-x)

        x = np.linspace(-10, 10, 21)
        conditions=[ x >= 0,  x >= 1, x < 0 ]
        functions=[ m2, m3, m1 ]
        m = np.piecewise(x, conditions, functions)
        m
```

1.3.3 §4.2.1

```
In [ ]: x=np.array([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
        r=np.array([2, 3, 4, 5])
        c=np.array([[5], [6], [7]])
        print "2*x = \n", 2*x
        print "r*x = \n", r*x
        print "x*r = \n", x*r
        print "c*x = \n", c*x
```

1.3.4 §4.2.2

```
In [ ]: xv=np.linspace(-1, 1, 5)
        yv=np.linspace(0, 1, 3)
        [xa, ya] = np.meshgrid(xv, yv)
        print "xa = \n", xa
        print "ya = \n", ya
        print "xa*ya = \n", xa*ya
```

```
In [ ]: [xo, yo]=np.ogrid[-1: 1: 5j, 0: 1: 3j]
        print "xo = \n", xo
        print "yo = \n", yo
        print "xo.shape = ", xo.shape, " yo.shape = ", yo.shape
        print "xo*yo = \n",  xo*yo
```

1.3.5 §4.4.1

```
In [ ]: import numpy as np
        quarter = np.array([1, 2, 3, 4],dtype=int)
        results=np.array([37.4, 47.3, 73.4, 99])
        outfile=open("q4.txt","w")
        outfile.write("The results for the first four quarters\n\n")
        for q,r in zip(quarter, results):
            outfile.write("For quarter %d the result is %5.1f\n" % (q,r))
        outfile.close()

In [ ]: infile=open("q4.txt", "r")
        lquarter = []
        lresult = []
        temp = infile.readline()
        temp = infile.readline()
        for line in infile:
            words = line.split()
            lquarter.append(int(words[2]))
            lresult.append(float(words[6]))
        infile.close()
        import numpy as np
        aquarter = np.array(lquarter, dtype=int)
        aresult = np.array(lresult)
        aquarter, aresult
```

1.3.6 §4.4.2

```
In [ ]: len = 21
        x = np.linspace(0, 2*np.pi, len)
        c = np.cos(x)
        s = np.sin(x)
        t = np.tan(x)
        arr = np.empty((4, len), dtype=float)
        arr[0, : ] = x
        arr[1, : ] = c
        arr[2, : ] = s
        arr[3, : ] = t
        np.savetxt('x.txt', x)
        np.savetxt('xcst.txt', (x, c, s, t))
        np.savetxt('xarr.txt', arr)
```

1.3.7 §4.4.3

```
In [ ]: np.savez('test.npz',x = x, c = c, s = s, t = t)

In [ ]: temp=np.load('test.npz')
        temp.files
        xc = temp['x']
```

```
cc = temp['c']
sc = temp['s']
tc = temp['t']
```

1.3.8 §4.7.2

```
In [ ]: import numpy as np

        roots = [0, 1, 1, 2]
        coeffs = np.poly(roots)
        print "coeffs = \n", coeffs
        print "np.roots(coeffs) = \n", np.roots(coeffs)
        x = np.linspace(0,0.5*np.pi,7)
        y = np.sin(x)
        c = np.polyfit(x, y, 3)
        print "c = \n", c
        y1 = np.polyval(c, x)
        print "y = \n", y, "\n y1 = \n", y1, "\n y1-y = \n", y1-y
```

1.3.9 §4.8.2

```
In [ ]: import numpy as np
        import numpy.linalg as npl

        a=np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]])
        print "a = ", a
        print "det(a) = ", npl.det(a)
        b=npl.inv(a)
        print "b = ", b, "\n b*a = ", np.dot(b,a)
```

```
In [ ]: import numpy as np
        import numpy.linalg as npl

        a = np.array([[-2, -4, 2], [-2, 1, 2], [4, 2, 5]])
        evals, evecs = npl.eig(a)
        eval1 = evals[0]
        evec1 = evecs[:,0]
        eval1, evec1
```

```
In [ ]: npl.norm(evec1), np.dot(evec1, evec1)
```

```
In [ ]: np.dot(a, evec1) - eval1*evec1
```

1.3.10 §4.8.3

```
In [ ]: import numpy as np
        import numpy.linalg as npl
        a=np.array([[3,2,1],[5,5,5],[1,4,6]])
        b=np.array([[5,1],[5,0],[-3,-7.0/2]])
```

```
        x=npl.solve(a,b)
        x
```

```
In [ ]: np.dot(a,x)-b
```

1.3.11 §4.9.1

```
In [ ]: import numpy as np
        import scipy.optimize as sco
        def fun(x):
            return np.cosh(x)/np.sinh(x)-x

        roots = sco.fsolve(fun, 1.0)
        root = roots[0]
        print "root is %15.12f and value is %e" % (root, fun(root))
```

1.4 Chapter 5

1.4.1 §5.2.3

If you are comparing the two notebook options, you should reset the kernel before switching.

```
In [ ]: #%%writefile foo.py
        import numpy as np
        import matplotlib.pyplot as plt

        #  Use the next command in Ipython terminal mode
        #plt.ion()
        #  Use the next command in Ipython non-inline notebook mode
        #matplotlib
        #  Use the next command in Ipython inline notebook mode
        %matplotlib notebook

        x = np.linspace(-np.pi, np.pi, 101)
        y = np.sin(x) + np.sin(3*x)/3.0

        plt.plot(x, y)

        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('A simple plot')

        #plt.show()
        plt.savefig('foo.pdf')
```

1.4.2 §5.3

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
```

```
%matplotlib notebook

x = np.linspace(-np.pi, np.pi, 101)
y = np.sin(x) + np.sin(3*x)/3.0

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('A simple plot')
plt.plot(x, y)

fig.savefig('foo.pdf')
```

1.4.3 §5.4

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook

        x=np.linspace(-np.pi,np.pi,101)
        f=np.ones_like(x)
        f[x<0] = -1
        y1 = (4/np.pi)*(np.sin(x) + np.sin(3*x)/3.0)
        y2 = y1 + (4/np.pi)*(np.sin(5*x)/5.0 + np.sin(7*x)/7.0)
        y3 = y2 + (4/np.pi)*(np.sin(9*x)/9.0 + np.sin(11*x)/11.0)

        plt.ion()
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(x,f,'b-',lw=3,label='f(x)')
        ax.plot(x,y1,'c--',lw=2,label='two terms')
        ax.plot(x,y2,'r-.',lw=2,label='four terms')
        ax.plot(x, y3,'b:',lw=2,label='six terms')
        ax.legend(loc='best')
        ax.set_xlabel('x',style='italic')
        ax.set_ylabel('partial sums',style='italic')
        fig.suptitle('Partial sums for Fourier series of f(x)',
                     size=16,weight='bold')
```

1.4.4 §5.5

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook

        theta = np.linspace(0,2*np.pi,201)
```

```
r1 = np.abs(np.cos(5.0*theta) - 1.5*np.sin(3.0*theta))
r2 = theta/np.pi
r3 = 2.25*np.ones_like(theta)

fig = plt.figure()
ax = fig.add_subplot(111,projection='polar')
ax.plot(theta, r1,label='trig')
ax.plot(5*theta, r2,label='spiral')
ax.plot(theta, r3,label='circle')
ax.legend(loc='best')
```

1.4.5 §5.6

```
In [ ]: import numpy as np
        import numpy.random as npr
        x = np.linspace(0,4,21)
        y = np.exp(-x)
        xe = 0.08*npr.randn(len(x))
        ye = 0.1*npr.randn(len(y))

        import matplotlib.pyplot as plt
        %matplotlib notebook

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.errorbar(x, y, fmt='bo', lw=2, xerr=xe, yerr=ye, ecolor='r', elinewidth=
```

1.4.6 §5.7

```
In [ ]: import numpy as np
        x = np.linspace(0,2,101)
        y = (x-1)**3+1

        import matplotlib.pyplot as plt
        %matplotlib notebook

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(x,y)
        ax.annotate('point of inflection at x=1', xy=(1,1), xytext=(0.8,0.5),
                    arrowprops=dict(facecolor='black',width=1, shrink=0.05))
```

1.4.7 §5.9

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
[X,Y] = np.mgrid[-2.5:2.5:51j, -3:3:61j]
Z=X**2-Y**2
curves = ax.contour(X, Y, Z, 12,colors='k')
ax.clabel(curves)
fig.suptitle(r'The level contours of $z=x^2-y^2$', fontsize=20)
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

fig = plt.figure()
ax = fig.add_subplot(111)
[X,Y] = np.mgrid[-2.5:2.5:51j, -3:3:61j]
Z=X**2-Y**2
im = ax.contourf(X, Y, Z, 12)
fig.colorbar(im, orientation='vertical')

ax.set_title(r'${\rm The\ level\ contours\ of\:} z=x^2-y^2$', fontsize=20)
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

from matplotlib import rc
rc('font',family='serif')
rc('text',usetex = True)

fig = plt.figure()
ax = fig.add_subplot(111)
[X,Y] = np.mgrid[-2.5:2.5:51j, -3:3:61j]
Z=X**2-Y**2
im = ax.imshow(Z)
fig.colorbar(im, orientation='vertical')

ax.set_title(r'The level contours of $z=x^2-y^2$', fontsize=20)
```

1.4.8 §5.10.1

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

x = np.linspace(0, 2*np.pi, 301)
y = np.cos(x)
z = np.sin(x)
```

```
plt.ion()
fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
ax1.plot(x,y) # First figure
fig2 = plt.figure()
ax2 = fig2.add_subplot(111)
ax2.plot(x,z) # Second figure
```

1.4.9 §5.10.2

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook

        x = np.linspace(0, 5, 101)
        y1 = 1.0/(x + 1.0)
        y2 = np.exp(-x)
        y3 = np.exp(-0.1*x**2)
        y4 = np.exp(-5*x**2)

        fig = plt.figure()
        ax1 = fig.add_subplot(2, 2, 1)
        ax1.plot(x, y1)
        ax1.set_xlabel('x')
        ax1.set_ylabel('y1')
        ax2 = fig.add_subplot(222)
        ax2.plot(x, y2)
        ax2.set_xlabel('x')
        ax2.set_ylabel('y2')
        ax3 = fig.add_subplot(223)
        ax3.plot(x, y3)
        ax3.set_xlabel('x')
        ax3.set_ylabel('y3')
        ax4 = fig.add_subplot(224)
        ax4.plot(x, y4)
        ax4.set_xlabel('x')
        ax4.set_ylabel('y4')
        fig.suptitle('Various decay functions')
```

1.4.10 §5.11

```
In [ ]: import numpy as np
        from time import time

        # Set the parameters
        max_iter = 256              # maximum number of iterations
        nx, ny = 1024, 1024         # x- and y-image resolutions
        x_lo, x_hi = -2.0, 1.0      # x bounds in complex plane
```

```
y_lo, y_hi = -1.5, 1.5    # y bounds in complex plane

start_time = time()

# Construct the two dimensional arrays
ix, iy = np.mgrid[0:nx, 0:ny]
x, y = np.mgrid[x_lo:x_hi:1j*nx, y_lo:y_hi:1j*ny]
c = x + 1j*y
esc_parm = np.zeros((ny, nx, 3),dtype='uint8') # holds pixel rgb data

# Flattened arrays
nxny = nx*ny
ix_f = np.reshape(ix,nxny)
iy_f = np.reshape(iy,nxny)
c_f = np.reshape(c,nxny)
z_f=c_f.copy()                          # the iterated variable

for iter in xrange(max_iter):           # do the iterations
    if not len(z_f):                    # all points have escaped
        break
    # rgb values for this choice of iter
    n = iter + 1
    r, g, b = n % 4 * 64, n % 8 * 32, n % 16 * 16

    # Mandelbrot evolution
    z_f *= z_f
    z_f += c_f
    escape=np.abs(z_f) > 2.0          # points which are escaping
    # Set the rgb pixel value for the escaping points
    esc_parm[iy_f[escape], ix_f[escape], :] = r, g, b
    escape = -escape                   # points not escaping
    # Remove batch of newly escaped points from flattened arrays
    ix_f = ix_f[escape]
    iy_f = iy_f[escape]
    c_f = c_f[escape]
    z_f = z_f[escape]

print "Time taken = ", time() - start_time

from PIL import Image

picture = Image.fromarray(esc_parm)
picture.show()
picture.save("mandelbrot.jpg")
```

1.5 Chapter 6

1.5.1 6.5.1

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
from matplotlib.widgets import Slider, Button, RadioButtons

def solwave(x, t, c):
    """ Solitary wave solution of the K deV equation."""
    return c/(2*np.cosh(np.sqrt(c)*(x-c*t)/2)**2)

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.15, bottom=0.30)
plt.xlabel("x")
plt.ylabel("u")
x = np.linspace(-5.0, 20.0, 1001)
t0 = 5.0
c0 = 1.0
line, = plt.plot(x, solwave(x, t0, c0), lw=2, color='blue')
plt.axis([-5, 20, 0, 2])

axcolor = 'lightgoldenrodyellow'
axtime = plt.axes([0.20, 0.15, 0.65, 0.03], axisbg=axcolor)
axvely = plt.axes([0.20, 0.1, 0.65, 0.03], axisbg=axcolor)

stime = Slider(axtime, 'Time', 0.0, 20.0, valinit=t0)
svely = Slider(axvely, 'Vely', 0.1, 3.0, valinit=c0)

def update(val):
    time = stime.val
    vely = svely.val
    line.set_ydata(solwave(x, time, vely))
    fig.canvas.draw_idle()

svely.on_changed(update)
stime.on_changed(update)

resetax = plt.axes([0.75, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor,
                hovercolor='0.975')

def reset(event):
    svely.reset()
    stime.reset()

button.on_clicked(reset)
```

```
plt.show()
```

1.5.2 §6.5.2

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import animation
from JSAnimation import IPython_display

def solwave(t, x, c=1):
    """ Solitary wave solution of the K deV equation."""
    return c/(2*np.cosh(np.sqrt(c)*(x-c*t)/2)**2)
```

In [ ]:
```
# Initialization
fig = plt.figure()
ax = plt.axes(xlim=(-5, 20), ylim=(0, 0.6))
line, = ax.plot([], [], lw=2)

t=np.linspace(-10,25,91)
x = np.linspace(-5, 20.0, 101)

def init():
    line.set_data([], [])
    return line,

def animate(i):
    y = solwave(t[i], x)
    line.set_data(x, y)
    return line,

animation.FuncAnimation(fig, animate, init_func=init,
                                frames=90, interval=30, blit=True)
```

1.5.3 §6.5.3

In [ ]:
```
import numpy as np

# Fix speed
c = 1.0

def solwave(t, x):
    """ Solitary wave solution of the K deV equation."""
    return c/(2*np.cosh(np.sqrt(c)*(x-c*t)/2)**2)

import matplotlib.pyplot as plt
```

```
def plot_solwave(t, x):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(x, solwave(t, x))
    ax.set_ylim(0, 0.6*c)
    ax.text(-4, 0.55*c, "t = " + str(t))
    ax.text(-4, 0.50*c, "c = " + str(c))

x = np.linspace(-5, 20.0, 101)
t = np.linspace(-10, 25, 701)

for i in range(len(t)):
    file_name='_temp%05d.png' % i
    plot_solwave(t[i], x)
    plt.savefig(file_name)
    plt.clf()

import os
os.system("rm _movie.mpg")
os.system("/opt/local/bin/ffmpeg -r 25 " +
          " -i _temp%05d.png  -b:v 1800  _movie.mpg")
os.system("rm _temp*.png")
```

1.5.4 §6.7.1

In [ ]:
```
import numpy as np

theta = np.linspace(0, 2*np.pi, 401)
a = 0.3    # specific but arbitrary choice of the parameters
m = 11
n = 9
x = (1 + a*np.cos(n*theta))*np.cos(m*theta)
y = (1 + a*np.cos(n*theta))*np.sin(m*theta)
z = a*np.sin(n*theta)

import matplotlib.pyplot as plt
%matplotlib notebook
from mpl_toolkits.mplot3d import Axes3D

plt.ion()
fig = plt.figure()
ax = Axes3D(fig)
ax.plot(x, y, z, 'g', linewidth=4)
ax.set_zlim3d(-1.0, 1.0)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('A spiral as a parametric curve', weight='bold', size=16)
```

```
#ax.elev, ax.azim = 60, -120
fig.savefig('torus.pdf')
```

1.5.5 §6.7.2

This snippet should be saved as a file. You then need to invoke the Mayavi application.

In [ ]:
```
%%writefile torus.py

import numpy as np

theta = np.linspace(0, 2*np.pi, 401)
a = 0.3 # specific but arbitrary choice of the parameters
m = 11
n = 9
x = (1 + a*np.cos(n*theta))*np.cos(m*theta)
y = (1 + a*np.cos(n*theta))*np.sin(m*theta)
z = a*np.sin(n*theta)

from mayavi import mlab
mlab.plot3d(x, y, z, np.sin(n*theta),
            tube_radius=0.025, colormap='spectral')
mlab.axes(line_width=2, nb_labels=5)
mlab.title('A spiral wrapped around a torus',size=1.2)
```

1.5.6 §6.8.1

In [ ]:
```
import numpy as np

xx, yy = np.mgrid[-2:2:81j, -3:3:91j]
zz = np.exp(-2*xx**2 - yy**2)*np.cos(2*xx)*np.cos(3*yy)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

plt.ion()

fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(xx, yy, zz, rstride=4, cstride=3, color='c',alpha=0.9)
ax.contour(xx, yy, zz, zdir='x',offset=-3.0, colors='black')
ax.contour(xx, yy, zz, zdir='y', offset=4.0, colors='blue')
ax.contour(xx, yy, zz, zdir='z', offset=-2.0)
ax.set_xlim3d(-3.0, 2.0)
ax.set_ylim3d(-3.0, 4.0)
ax.set_zlim3d(-2.0, 1.0)
ax.set_xlabel('x')
```

```
ax.set_ylabel('y')
ax.set_zlabel('z')

fig.savefig('surf1.pdf')
```

1.5.7 §6.8.2

```
In [ ]: %%writefile surf2.py
        import numpy as np

        xx, yy = np.mgrid[-2:2:81j, -3:3:91j]
        zz = np.exp(-2*xx**2 - yy**2)*np.cos(2*xx)*np.cos(3*yy)

        from mayavi import mlab
        fig = mlab.figure()
        s = mlab.surf(xx, yy, zz, representation='surface')
        ax = mlab.axes(line_width=2, nb_labels=5)
        #mlab.title('Simple surface plot',size=0.4)
```

1.5.8 §6.9.1

```
In [ ]: import numpy as np

        [u, v] = np.mgrid[-2:2:51j, -2:2:61j]
        x, y, z = u*(1 - u**2/3 + v**2), v*(1 - v**2/3 + u**2), u**2 - v**2

        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        plt.ion()

        fig = plt.figure()
        ax = Axes3D(fig)
        ax.plot_surface(x.T, y.T, z.T, rstride=2, cstride=2, color='r',
                        alpha=0.2, linewidth=0.5)
        ax.elev, ax.azim = 50, -80
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')
        #ax.set_title('A parametric surface plot', # weight='bold',size=18)
        fig.savefig('surf3.pdf')
```

1.5.9 6.9.2

```
In [ ]: %%writefile surf4.py

        import numpy as np
        [u, v] = np.mgrid[-2:2:51j, -2:2:61j]
        x, y, z = u*(1 - u**2/3 + v**2), v*(1 - v**2/3 + u**2), u**2 - v**2
```

```
from mayavi import mlab
fig = mlab.figure()
s = mlab.mesh(x, y, z, representation='surface',
              line_width=0.5, opacity=0.5)
ax = mlab.axes(line_width=2, nb_labels=5)
```

1.5.10 §6.10

In [ ]:
```
%%writefile Julia.py

import numpy as np

# Set up initial grid
x, y = np.ogrid[-1.5:0.5:1000j, -1.0:1.0:1000j]
z= x + 1j*y
julia = np.zeros(z.shape)
c = -0.7 - 0.4j

# Build the Julia set
for it in range(1,101):
    z = z**2 + c
    escape = z*z.conj()>4
    julia += (1/float(it))*escape

from mayavi import mlab
mlab.figure(size = (800,600))
mlab.surf(julia, colormap='gist_ncar', warp_scale='auto', vmax=1.5)
mlab.view(15,30,500,[-0.5,-0.5,2.0])
mlab.show()
```

1.6 Chapter 7

1.6.1 §7.2

In [ ]:
```
import sympy as sy
sy.init_printing()
x, y = sy.symbols("x y")
D = (x + y)*sy.exp(x)*sy.cos(y)
D
```

In [ ]:
```
f, g = sy.symbols("f g", cls=sy.Function)
f(x), f(x, y), g(x)
i,  j = sy.symbols("i j", integer=True)
u, v = sy.symbols("u v", real=True)
i.is_integer, j*j,  (j*j).is_integer, u.is_real
```

1.6.2 §7.3

```
In [ ]: D_s = sy.sympify('(x+y)*exp(x)*cos(y)')
        cosdiff = sy.sympify("cos(x)*cos(y) + sin(x)*sin(y)")
        D_s, cosdiff
```

```
In [ ]: import numpy as np
        func = sy.lambdify((x, y), cosdiff, "numpy")
        xn = np.linspace(0, np.pi, 13)
        func(xn, 0.0) # should return np.cos(xn)
```

1.6.3 §7.4

```
In [ ]: M = sy.Matrix([[1,x],[y,1]])
        V = sy.Matrix([[u],[v]])
        M, V, M*V
```

```
In [ ]: M[0,1]= u
        M
```

1.6.4 §7.8

```
In [ ]: from sympy.solvers import dsolve
        dsolve?
```

```
In [ ]: ode1 = f(x).diff(x, 2) + 4*f(x)
        sol1 = dsolve(ode1, f(x))
        ode1, sol1
```

```
In [ ]: C1, C2 = sy.symbols("C1, C2")
        fun = sol1.rhs
        fund = fun.diff(x)
        fun.subs(x, 0), fund.subs(x, 0)
```

```
In [ ]: psol = sol1.subs([(C2, 2), (C1, 0)])
        psol
```

```
In [ ]:  ode2 = sy.sin(f(x)) + (x*sy.cos(f(x)) + f(x))*f(x).diff(x)
         ode2, dsolve(ode2, f(x))
```

```
In [ ]: ode3 = x*f(x).diff(x) + f(x) - sy.log(x)*f(x)**2
        ode3, dsolve(ode3)
```

```
In [ ]:  ode5 = f(x).diff(x, 2) + (f(x).diff(x))**2/f(x) + f(x).diff(x)/x
         ode5, dsolve(ode5)
```

```
In [ ]:  ode6 = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
         ode6, dsolve(ode6)
```

```
In [ ]: ode7 = [f(x).diff(x) - 2*f(x) - g(x), g(x).diff(x) -f(x) - 2*g(x)]
        ode7, dsolve(ode7)
```

```
In [ ]: ode8 = f(x).diff(x)  - (x + f(x))**2
        ode8, dsolve(ode8)

In [ ]: ode9 = g(x).diff(x) -1 - (g(x))**2
        dsolve(ode9)
```

1.6.5 §7.9

```
In [ ]: %matplotlib notebook
        import sympy.plotting as syp

In [ ]: s1 = syp.plot(sy.sin(x), x, x-x**3/6, x-x**3/6 + x**5/120, (x, -4, 4),
                      title='sin(x) and its first three Taylor approximants')

In [ ]: xc = sy.cos(u) + sy.cos(7*u)/2 + sy.sin(17*u)/3
        yc = sy.sin(u) + sy.sin(7*u)/2 + sy.cos(17*u)/3
        fig = syp.plot_parametric(xc, yc, (u, 0, 2*sy.pi))
        fig.save("sympy2.pdf")

In [ ]: x, y = sy.symbols("x y", real=True)
        z = x + sy.I*y
        w = sy.cos(z**2).expand(complex=True)
        wa = sy.Abs(w).expand(complex=True)
        syp.plot_implicit(wa**2- 1)

In [ ]: fig = syp.plot3d_parametric_surface((3 + sy.sin(u) + sy.cos(v))*sy.cos(2*u)
                                           (3 + sy.sin(u) + sy.cos(v))*sy.sin(2*u)
                                           2*sy.cos(u) + sy.sin(v),
                                           (u, 0, 2*sy.pi), (v, 0, 2*sy.pi))
        fig.save("foo.pdf")
```

1.7 Chapter 8

1.7.1 §8.3.2

```
In [ ]:  import numpy as np
         import matplotlib.pyplot as plt
         from scipy.integrate import odeint
         %matplotlib notebook

         def rhs(Y, t, omega):
              y, ydot = Y
              return ydot, -omega**2*y

         t_arr = np.linspace(0, 2*np.pi, 101)
         y_init = [1, 0]
         omega = 2.0
         y_arr=odeint(rhs, y_init, t_arr, args=(omega,))
         y, ydot = y_arr[:, 0], y_arr[:, 1]
```

```
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.plot(t_arr, y, t_arr, ydot)
ax1.set_xlabel('t')
ax1.set_ylabel('y and ydot')
ax2 = fig.add_subplot(122)
ax2.plot(y, ydot)
ax2.set_xlabel('y')
ax2.set_ylabel('ydot')
plt.suptitle("Solution curve when omega = %5g" % omega)
fig.tight_layout()
fig.subplots_adjust(top=0.90)
```

The next snippet must be run in terminal mode!

In [ ]:
```
%%writefile traj.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(Y, t, omega):
    y,ydot=Y
    return ydot, -omega**2*y

t_arr = np.linspace(0, 2*np.pi, 101)
y_init = [1, 0]
omega = 2.0

fig = plt.figure()
y, ydot = np.mgrid[-3:3:21j, -6:6:21j]
u, v = rhs(np.array([y, ydot]), 0.0, omega)
mag = np.hypot(u, v)
mag[mag==0] = 1.0
plt.quiver(y, ydot, u/mag, v/mag, color='red')

# Enable drawing of arbitrary number of trajectories
print "\n\n\nUse mouse to select each starting point"
print "Timeout after 30 seconds"
choice = [(0, 0)]
while len(choice) > 0:
    y01 = np.array([choice[0][0], choice[0][1]])
    y = odeint(rhs, y01, t_arr, args=(omega,))
    plt.plot(y[:, 0], y[:, 1], lw=2)
    choice = plt.ginput()
print "Timed out!"
```

1.7.2 §8.3.2

In [ ]:
```
import numpy as np
from scipy.integrate import odeint

def rhs(y, t, mu):
    return [ y[1], mu*(1-y[0]**2)*y[1] - y[0]]

def jac(y, t, mu):
    return [ [0, 1], [-2*mu*y[0]*y[1]-1, mu*(1-y[0]**2)] ]

mu = 1.0
t_final = 15.0 if mu < 10 else 4.0*mu
n_points = 1001 if mu < 10 else 1001*mu
t = np.linspace(0,t_final,n_points)
y0 = np.array([2.0, 0.0])
y, info = odeint(rhs, y0, t, args=(mu,), Dfun=jac, full_output=True)

print " mu = %g, number of Jacobian calls is %d" % \
        (mu, info['nje'][-1])

import matplotlib.pyplot as plt
%matplotlib notebook
plt.plot(y[:,0], y[:,1])
```

1.7.3 §8.3.4

In [ ]:
```
import numpy as np
from scipy.integrate import odeint

def rhs(u, t, beta, rho, sigma):
    x, y, z = u
    return [sigma*(y-x), rho*x-y-x*z, x*y-beta*z]

sigma = 10.0
beta = 8.0/3.0
rho1 = 29.0
rho2 = 28.8

u01 = [1.0, 1.0, 1.0]
u02 = [1.0, 1.0, 1.0]

t = np.linspace(0.0, 50.0, 10001)
u1 = odeint(rhs, u01, t, args=(beta,rho1,sigma))
u2 = odeint(rhs, u02, t, args=(beta,rho2,sigma))

x1, y1, z1 = u1[:, 0], u1[:, 1], u1[:, 2]
x2, y2, z2 = u2[:, 0], u2[:, 1], u2[:, 2]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

plt.ion()
fig = plt.figure()
ax=Axes3D(fig)
ax.plot(x1, y1, z1, 'b-')
ax.plot(x2, y2, z2, 'r:')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Lorenz equations with rho = %g, %g' % (rho1,rho2))
```

1.7.4 §8.4.3

In [ ]:
```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees = [2]
boundary_points = np.array([0,np.pi])
tol = 1.0e-8*np.ones_like(boundary_points)

def fsub(x, Z):
    """The equations"""
    u, du = Z
    return np.array([-u])

def gsub(Z):
    """The boundary conditions"""
    u, du = Z
    return np.array([u[0], du[1]-1.0])

solution = colnew.solve(boundary_points, degrees, fsub, gsub,
                        is_linear=True, tolerances=tol,
                        vectorized=True, maximum_mesh_size=300)
x = solution.mesh
u_exact = -np.sin(x)

import matplotlib.pyplot as plt
%matplotlib notebook
plt.ion()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,solution(x)[:,0],'b.')
ax.plot(x,u_exact,'g-')
```

1.7.5 §8.4.4

In [ ]:
```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees = [2, 1, 1]
boundary_points = np.array([0.0, 0.0, 1.0, 1.0])
tol = 1.0e-5*np.ones_like(boundary_points)

def fsub(x, Z):
    """The equations"""
    u, du, v, w = Z
    ddu = -u*v
    dv = np.zeros_like(x)
    dw = u*u
    return np.array([ddu, dv, dw])

def gsub(Z):
    """The boundary conditions"""
    u, du, v, w = Z
    return np.array([u[0], w[1], u[2], w[3]-1.0])


guess_lambda = 100.0
def guess(x):
    u = x*(1.0/3.0 - x)*(2.0/3.0 - x)*(1.0 - x)
    du = 2.0*(1.0 - 2.0*x)*(1.0 - 9.0*x + 9.0*x*x)/9.0
    v = guess_lambda*np.ones_like(x)
    w = u*u
    Z_guess = np.array([u, du, v, w])
    f_guess = fsub(x, Z_guess)
    return Z_guess, f_guess


solution=colnew.solve(boundary_points, degrees,fsub,gsub,
                      is_linear=False, initial_guess=guess,
                       tolerances=tol, vectorized=True,
                       maximum_mesh_size=300)
x= solution.mesh
u_exact = np.sqrt(2)*np.sin(3.0*np.pi*x)


# plot solution

import matplotlib.pyplot as plt
%matplotlib notebook

plt.ion()
```

```
plt.plot(x, solution(x)[:, 0], 'b.', x, u_exact, 'g-')
print "Third eigenvalue is %16.10e ." % solution(x)[0, 2]
```

1.7.6 §8.4.5

In [ ]:
```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees = [2, 1, 1]
boundary_points = np.array([0.0, 0.0, 1.0, 1.0])
tol = 1.0e-8*np.ones_like(boundary_points)

def fsub(x, Z):
    """The equations"""
    u, du, v, w = Z
    ddu = -v*np.exp(u)
    dv = np.zeros_like(x)
    dw = u*u
    return np.array([ddu, dv, dw])


def gsub(Z):
    """The boundary conditions"""
    u, du, v, w = Z
    return np.array([u[0], w[1], u[2], w[3]-gamma])

def guess(x):
    u = 0.5*x*(1.0 - x)
    du = 0.5*(1.0 - 2.0*x)
    v = np.zeros_like(x)
    w = u*u
    Z_guess = np.array([u, du, v, w])
    f_guess = fsub(x, Z_guess)
    return Z_guess, f_guess

solution = guess
gaml = []
laml = []

for gamma in np.linspace(0.01, 5.01, 1001):
    solution = colnew.solve(boundary_points, degrees, fsub, gsub,
                            is_linear=False, initial_guess=solution,
                            tolerances=tol, vectorized=True,
                            maximum_mesh_size=300)
    x = solution.mesh
    lam = solution(x)[:, 2]
    gaml.append(gamma)
    laml.append(np.max(lam))
```

```
# plot solution
import matplotlib.pyplot as plt
%matplotlib notebook

plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(gaml, laml)
ax.set_xlabel(r'$\gamma$', size=20)
ax.set_ylabel(r'$\lambda$', size=20)
ax.grid(b=True)
```

1.7.7 §8.5.3

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
from pydelay import dde23

t_final = 50
delay = 2.0
x_initial = 0.1

equations={'x' : 'x*(1.0-x(t-tau))'}
parameters={'tau' : delay}
dde=dde23(eqns=equations, params=parameters)
dde.set_sim_params(tfinal=t_final, dtmax=1.0,
                   AbsTol=1.0e-6, RelTol=1.0e-3)
histfunc={'x': lambda t: x_initial}
dde.hist_from_funcs(histfunc, 101)
dde.run()

t_vis = 0.1*t_final
sol = dde.sample(tstart=t_vis+delay, tfinal=t_final, dt=0.1)
t = sol['t']
x = sol['x']
sold = dde.sample(tstart=t_vis, tfinal=t_final-delay, dt=0.1)
xd = sold['x']

plt.ion()
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.plot(t, x)
ax1.set_xlabel('t')
ax1.set_ylabel('x(t)')

ax2 = fig.add_subplot(122)
```

```
ax2.plot(x, xd)
ax2.set_xlabel('tx')
ax2.set_ylabel('x(t-tau)')
```

1.7.8 §8.5.4

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook
        from pydelay import dde23

        t_final = 500
        a = 2.0
        b = 1.0
        m = 7.0
        delay = 2.0
        x_initial = 0.5

        equations={'x' : 'a*x(t-tau)/(1.0+pow(x(t-tau), m))- b*x'}
        parameters={'a': a, 'b' : b, 'm': m, 'tau' : delay}
        dde=dde23(eqns=equations, params=parameters)
        dde.set_sim_params(tfinal=t_final, dtmax=1.0,
                           AbsTol=1.0e-6, RelTol=1.0e-3)
        histfunc={'x': lambda t: x_initial}
        dde.hist_from_funcs(histfunc, 101)
        dde.run()

        t_vis = 0.95*t_final
        sol = dde.sample(tstart=t_vis+delay, tfinal=t_final, dt=0.1)
        t = sol['t']
        x = sol['x']
        sold = dde.sample(tstart=t_vis, tfinal=t_final-delay, dt=0.1)
        xd = sold['x']

        plt.ion()
        fig = plt.figure()
        ax1 = fig.add_subplot(121)
        ax1.plot(t, x)
        ax1.set_xlabel('t')
        ax1.set_ylabel('x(t)')

        ax2 = fig.add_subplot(122)
        ax2.plot(x, xd)
        ax2.set_xlabel('tx')
        ax2.set_ylabel('x(t-tau)')
```

1.7.9 §8.6.1

```
In [ ]: import numpy as np
        import numpy.random as npr

        T = 1
        N = 500
        t, dt = np.linspace(0, T, N+1, retstep=True)
        dW = npr.normal(0.0, np.sqrt(dt), N+1)
        dW[0] = 0.0
        W = np.cumsum(dW)

        import matplotlib.pyplot as plt
        %matplotlib notebook
        plt.ion()

        plt.plot(t,W)
        plt.xlabel('t')
        plt.ylabel('W(t)')
        #plt.title('Sample Wiener Process',weight='bold',size=16)
```

1.7.10 §8.6.2

```
In [ ]: import numpy as np
        import numpy.random as npr

        T = 1
        N = 1000
        M = 5000
        t, dt = np.linspace(0, T, N+1, retstep=True)
        dW = npr.normal(0.0, np.sqrt(dt), (M,N+1))
        dW[ : ,0] = 0.0
        W = np.cumsum(dW, axis=1)
        U = np.exp(- 0.5*W)
        Umean = np.mean(U, axis=0)
        Uexact = np.exp(t/8)

        import matplotlib.pyplot as plt
        %matplotlib notebook
        plt.ion()

        fig = plt.figure()
        ax = fig.add_subplot(111)

        ax.plot(t, Umean, 'b-', label="mean of %d paths" % M)
        ax.plot(t, Uexact, 'r-', label="exact " + r'$\langle U\rangle$')
        for i in range(5):
            ax.plot(t,U[i, : ], '--')
```

```
ax.set_xlabel('t')
ax.set_ylabel('U')
ax.legend(loc='best')

maxerr = np.max(np.abs(Umean-Uexact))
print "With %d paths and %d intervals the max error is %g" %  \
        (M, N, maxerr)
```

1.7.11 8.6.4

```
In [ ]: import numpy as np
        import numpy.random as npr

        # Set up grid
        T=1.0
        N=1000
        t, dt = np.linspace(0, T, N+1, retstep=True)

        # Get Brownian motion
        dW = npr.normal(0.0,  np.sqrt(dt), N+1)
        dW[0] = 0.0
        W = np.cumsum(dW)

        # Equation parameters and functions
        lamda = 2.0
        mu = 1.0
        Xzero = 1.0
        def a(X): return lamda*X
        def b(X): return mu*X
        def bd(X): return mu*np.ones_like(X)

        # Analytic solution
        Xanal = Xzero*np.exp((lamda-0.5*mu*mu)*t+mu*W)

        # Milstein solution
        Xmil = np.empty_like(t)
        Xmil[0] = Xzero
        for n in range(N):
            Xmil[n+1] = Xmil[n] + dt*a(Xmil[n]) + dW[n+1]*b(Xmil[n]) + 0.5*(
                b(Xmil[n])*bd(Xmil[n])*(dW[n+1]**2-dt))

        # Plot solution
        import matplotlib.pyplot as plt
        %matplotlib notebook
        plt.ion()

        fig = plt.figure()
```

```
ax = fig.add_subplot(111)

ax.plot(t, Xanal, 'b-', label='analytic')
ax.plot(t, Xmil, 'g-.', label='Milstein')
ax.legend(loc='best')
ax.set_xlabel('t')
ax.set_ylabel('X(t)')
```

In [ ]:
```
import numpy as np
import numpy.random as npr

# Problem definition
M = 1000                    # Number of paths sampled
P = 6                       # Number of discretizations
T = 1                       # Endpoint of time interval
N = 2**12                   # Finest grid size
dt = 1.0*T/N

# Problem parameters
lamda = 2.0
mu = 1.0
Xzero = 1.0

def a(X): return lamda*X
def b(X): return mu*X
def bd(X): return mu*np.ones_like(X)

# Build the Brownian paths.
dW = npr.normal(0.0, np.sqrt(dt), (M,N+1))
dW[ : , 0] = 0.0
W = np.cumsum(dW, axis=1)

# Build the exact solutions at the ends of the paths
ones = np.ones(M)
Xexact = Xzero*np.exp((lamda-0.5*mu*mu)*ones+mu*W[ : , -1])
Xemerr = np.empty((M,P))
Xmilerr = np.empty((M,P))

# Loop over refinements
for p in range(P):
    R = 2**p
    L = N/R                         # must be an integer!
    Dt = R*dt
    Xem = Xzero*ones
    Xmil = Xzero*ones
    Wc = W[ : , : :R]
    for j in range(L):              # integration
        deltaW = Wc[ : , j+1]-Wc[ : , j]
```

```
            Xem += Dt*a(Xem) + deltaW*b(Xem)
            Xmil += Dt*a(Xmil) + deltaW*b(Xmil)+ \
                    0.5*b(Xmil)*bd(Xmil)*(deltaW**2-Dt)
        Xemerr[ : ,p] = np.abs(Xem-Xexact)
        Xmilerr[ : ,p] = np.abs(Xmil-Xexact)

    Dtvals = dt*np.array([2**p for p in range(P)])
    lDtvals = np.log10(Dtvals)
    Xemerrmean = np.mean(Xemerr,axis=0)

    # Do some plotting
    import matplotlib.pyplot as plt
    %matplotlib notebook
    plt.ion()

    fig = plt.figure()
    ax = fig.add_subplot(111
                         )
    ax.plot(lDtvals, np.log10(Xemerrmean), 'bo')
    ax.plot(lDtvals, np.log10(Xemerrmean), 'b:', label='EM actual')
    ax.plot(lDtvals, 0.5*np.log10(Dtvals), 'b-.', label='EM theoretical')
    Xmilerrmean = np.mean(Xmilerr, axis=0)
    ax.plot(lDtvals, np.log10(Xmilerrmean), 'bo')
    ax.plot(lDtvals, np.log10(Xmilerrmean),' b--', label='Mil actual')
    ax.plot(lDtvals, np.log10(Dtvals), 'b-', label='Mil theoretical')
    ax.legend(loc='best')
    ax.set_xlabel(r'$\log_{10}\Delta t$', size=16)
    ax.set_ylabel(r'$\log_{10}\left(\langle|X_n-X(\tau)|\rangle\right)$', size=

    emslope = ((np.log10(Xemerrmean[-1])-np.log10(Xemerrmean[0])) /
               (lDtvals[-1]-lDtvals[0]))
    print 'Empirical EM slope is %g' % emslope
    milslope = ((np.log10(Xmilerrmean[-1])- np.log10(Xmilerrmean[0])) /
               (lDtvals[-1]-lDtvals[0]))
    print 'Empirical MIL slope is %g' % milslope
```

1.8 Chapter 9

1.8.1 §9.4

In [ ]:
```
    import numpy as np
    from scipy.fftpack import diff

    def fd(u):
        """ Return 2*dx*finite-difference x-derivative of u. """
        ud = np.empty_like(u)
        ud[1:-1]=u[2: ]-u[ :-2]
        ud[0]=u[1]-u[-1]
```

```
    ud[-1]=u[0]-u[-2]
    return ud

for N in [4,8,16,32,64,128,256]:
    dx=2.0*np.pi/N
    x=np.linspace(0,2.0*np.pi,N,endpoint=False)
    u=np.exp(np.sin(x))
    du_ex=np.cos(x)*u
    du_sp=diff(u)
    du_fd=fd(u)/(2.0*dx)
    err_sp=np.max(np.abs(du_sp-du_ex))
    err_fd=np.max(np.abs(du_fd-du_ex))
    print "N=%3d, err_sp=%.1e err_fd=%.1e" % (N, err_sp,err_fd)
```

1.8.2 §9.5

In [ ]:
```
import numpy as np
from scipy.fftpack import diff
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

def u_exact(t,x):
    """ Exact solution. """
    return np.exp(np.sin(x-2.0*np.pi*t))

def rhs(u,t):
    """ Return rhs. """
    return -2.0*np.pi*diff(u)

N=32
x=np.linspace(0,2.0*np.pi,N,endpoint=False)
u0 =u_exact(0.0,x)
t_initial=0.0
t_final=64.0*np.pi
t=np.linspace(t_initial,t_final,101)
sol=odeint(rhs,u0,t,mxstep=5000)

plt.ion()
fig=plt.figure()
ax=Axes3D(fig)
t_gr,x_gr=np.meshgrid(x,t)
ax.plot_surface(t_gr,x_gr,sol,cstride=2,rstride=8,alpha=0.1)
ax.elev,ax.azim=47,-137
ax.set_xlabel('x')
ax.set_ylabel('t')
```

```
        ax.set_zlabel('u')

        u_ex=u_exact(t[-1],x)
        err=np.abs(np.max(sol[-1,: ]-u_ex))
        print "With %d Fourier modes the final error = %g" % (N,err)
```

1.8.3 §9.6

```
In [ ]: import numpy as np

        def u(x): return 1.0/(1.0+25.0*x**2)

        N=20
        x_grid=np.linspace(-1.0,1.0,N+1)
        u_grid=u(x_grid)
        z=np.polyfit(x_grid,u_grid,N)
        p=np.poly1d(z)
        x_fine=np.linspace(-1.0,1.0,5*N+1)
        u_fine=p(x_fine)
        u_fine
```

1.9 Introduction to f2py

Run the following line in the notebook, after changing the address to that of your copy of *f2py*, so that you can access *f2py* from your notebook in a compact manner.

```
In [ ]: %alias f2py '/Users/john/Library/Enthought/Canopy_64bit/User/bin/f2py'
```

1.9.1 §9.7.1

```
In [ ]: %%writefile norm3.f90
        ! File:norm3.f90  A simple subroutine in f90

        subroutine norm(u,v,w,s)
        real(8), intent(in) :: u,v,w
        real(8), intent(out) :: s
        s=sqrt(u*u+v*v+w*w)
        end subroutine norm

In [ ]: !gfortran norm3.f90

In [ ]: f2py -c norm3.f90 --f90flags=-O3 -m normv3

In [ ]: import normv3
        normv3?

In [ ]: normv3.norm(3, 4, 5)
```

```
In [ ]: %%writefile norm3.f
        C     FILE NORM3.F A SIMPLE SUBROUTINE IN F77

              SUBROUTINE NORM(U,V,W,S)
              REAL*8 U,V,W,S
        Cf2py intent(in) U,V,W
        Cf2py intent(out) S
              S=SQRT(U*U+V*V+W*W)
              END

In [ ]: !rm normv3.so

In [ ]: f2py -c norm3.f -m normv3

In [ ]: import normv3
        normv3?

In [ ]: normv3.norm(3,4,5)

In [ ]: rm  normv3.so
```

1.9.2 §9.7.2

```
In [ ]: %%writefile normn.f
        C     FILE NORMN.F  EXAMPLE OF N-DIMENSIONAL NORM

              SUBROUTINE NORM(U,S,N)
              INTEGER N
              REAL*8 U(N)
              REAL*8 S
        Cf2py intent(in) N
        Cf2py intent(in) U
        Cf2py depend(U) N
        Cf2py intent(out) S
              REAL*8 SUM
              INTEGER I
              SUM=0
              DO 100 I=1,N
         100  SUM=SUM + U(I)*U(I)
              S=SQRT(SUM)
              END

In [ ]: f2py -c normn.f --f77flags=-O3 -m normn

In [ ]: import normn
        normn.norm([3,4,5,6,7])
```

1.9.3 §9.7.3

```
In [ ]: %%writefile modmat.f
        C     FILE MODMAT.F  MODIFYING A MULTIDIMENSIONAL ARRAY

              SUBROUTINE MMAT(A,B,C,R,M,N)
              INTEGER M,N
              REAL*8 A(M,N),B(M),C(N),R(M,N)
        Cf2py intent(in) M,N
        Cf2py intent(in) A,B,C
        Cf2py depend(A) M,N
        Cf2py intent(out) R
              INTEGER I,J
              DO 10 I=1,M
                 DO 10 J=1,N
         10         R(I,J)=A(I,J)
              DO 20 I=1,M
         20      R(I,1)=R(I,1)+B(I)
              DO 30 J=1,N
         30      R(1,J)=R(1,J)+C(J)
              END


In [ ]: f2py -c modmat.f --f77flags=-O3 -m modmat

In [ ]: import numpy as np
        import modmat

        a=np.array([[1,2,3],[4,5,6]],dtype='float',order='F')
        b=np.array([10,20],dtype='float')
        c=np.array([7,11,13],dtype='float')
        r=modmat.mmat(a,b,c)
        r
```

1.9.4 §9.8

The code from Appendix B

```
In [ ]: %%writefile cheb.f

              SUBROUTINE CHEBPTS(X, N)
        C     CREATE N+1 CHEBYSHEV DATA POINTS IN X
              IMPLICIT REAL*8 (A-H,O-Z)
              DIMENSION X(0:N)
        Cf2py intent(in) N
        Cf2py intent(out) X
        Cf2py depend(N) X
              PI = 4.D0*ATAN(1.D0)
              DO 10 I=0,N
```

```
 10      X(I) = -COS(PI*I/N)
      RETURN
      END

      SUBROUTINE FFT (A,B,IS,N,ID)
C-- +--------------------------------------------------------------
C-- | A CALL TO FFT REPLACES THE COMPLEX DATA VALUES A(J) + i B(J),
C-- | J=0,1,... ,N-1 WITH THEIR TRANSFORM
C-- |
C-- |                           2 i ID PI K J / N
C-- | SUM      (A(K) + iB(K))e                  , J=0,1,.. .,N-1
C-- | K=0..N-1
C-- |
C-- | INPUT AND OUTPUT PARAMETERS
C-- |      A   ARRAY A (0: *), REAL PART OF DATA/TRANSFORM
C-- |      B   ARRAY B (0: *), IMAGINARY PART OF DATA/TRANSFORM
C-- | INPUT PARAMETERS
C-- |    IS    SPACING BETWEEN CONSECUTIVE ELEMENTS IN A AND B
C-- |          (USE IS=+1 FOR ELEMENTS STORED CONSECUTIVELY)
C-- |    N     NUMBER OF DATA VALUES, MUST BE A POWER OF TWO
C-- |    ID    USE +1 OR -1 TO SPECIFY DIRECTION OF TRANSFORM
C-- +--------------------------------------------------------------
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:*),B(0:*)
Cf2py intent(in, out) A, B
Cf2py intent(in) IS, ID
Cf2py integer intent(hide), depend(A) :: N
      J=0
C---   APPLY PERMUTATION MATRIX ----
      DO 20 I=0,(N-2)*IS,IS
         IF (I.LT.J) THEN
            TR = A(J)
            A(J) = A(I)
            A(I) = TR
            TI   = B(J)
            B(J) = B(I)
            B(I) = TI
         ENDIF
         K = IS*N/2
 10      IF (K.LE.J) THEN
            J = J-K
            K = K/2
            GOTO 10
         ENDIF
 20      J = J+K
C---   PERFORM THE LOG2 N MATRIX-VECTOR MULTIPLICATIONS ---
         S = 0.0D0
         C = -1.0D0
```

```
         L = IS
 30      LH=L
         L = L+L
         UR = 1.0D0
         UI = 0.0D0
         DO 50 J=0,LH-IS,IS
            DO 40 I=J,(N-1)*IS,L
               IP = I+LH
               TR = A(IP)*UR-B(IP)*UI
               TI = A(IP)*UI+B(IP)*UR
               A(IP) = A(I)-TR
               B(IP) = B(I)-TI
               A(I) = A(I)+TR
 40            B(I) = B(I)+TI
            TI = UR*S+UI*C
            UR = UR*C-UI*S
 50         UI=TI
         S = SQRT (0.5D0*(1.0D0-C))*ID
         C = SQRT (0.5D0*(1.0D0+C))
         IF (L.LT.N*IS) GOTO 30
       RETURN
       END

      SUBROUTINE FCT (A,X,N,B)
C-- +--------------------------------------------------------------
C-- | A CALL TO FCT PLACES IN B(0:N) THE COSINE TRANSFORM OF THE
C-- | VALUES IN A(0:N)
C-- |
C-- | B(J) = SUM    C(K)*A(K)*COS(PI*K*J/N) , J=0,1,...,N, K=0..N
C-- |
C-- | WHERE C(K) = 1.0 FOR K=0,N, C(K) =2.0 FOR K=1,2,...,N-1
C__ |
C-- | INPUT PARAMETERS:
C-- | A   A(0:N) ARRAY WITH INPUT DATA
C-- | X   X(0:N) ARRAY WITH CHEBYSHEV GRID POINT LOCATIONS
C-- |     X(J) = -COS(PI*J/N) , J=0,1,...,N
C-- | N   SIZE OF TRANSFORM - MUST BE A POWER OF TWO
C-- | OUTPUT PARAMETER
C-- | B   B(0:N) ARRAY RECEIVING TRANSFORM COEFFICIENTS
C-- |     (MAY BE IDENTICAL WITH THE ARRAY A)
C-- |
C-- +--------------------------------------------------------------
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:*),X(0:*),B(0:*)
Cf2py intent(in) A, X
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) :: N
      N2 = N/2
```

```
      A0 = A(N2-1)+A(N2+1)
      A9 = A(1)
      DO 10 I=2,N2-2,2
         A0 = A0+A9+A(N+1-I)
         A1 = A(  I+1)-A9
         A2 = A(N+1-I)-A(N-1-I)
         A3 = A(I)+A(N-I)
         A4 = A(I)-A(N-I)
         A5 = A1-A2
         A6 = A1+A2
         A1 = X(N2-I)
         A2 = X(I)
         A7 = A1*A4+A2*A6
         A8 = A1*A6-A2*A4
         A9 = A(I+1)
         B(I  ) = A3+A7
         B(N-I) = A3-A7
         B(I+1  ) = A8+A5
 10      B(N+1-I) = A8-A5
      B(1) = A(0)-A(N)
      B(0) = A(0)+A(N)
      B(N2 ) = 2.D0*A(N2)
      B(N2+1) = 2.D0*(A9-A(N2+1))
      CALL FFT(B(0),B(1),2,N2,1)
      A0 = 2.D0*A0
      B(N) = B(0)-A0
      B(0) = B(0)+A0
      DO 20 I=1,N2-1
         A1 = 0.5 D0         *(B(I)+B(N-I))
         A2 = 0.25D0/X(N2+I)*(B(I)-B(N-I))
         B(I  ) = A1+A2
 20      B(N-I) = A1-A2
      RETURN
      END


      SUBROUTINE FROMCHEB (A,X,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),X(0:N),B(0:N)
Cf2py intent(in) A, X
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) :: N
      B(0) = A(0)
      A1 = 0.5D0
      DO 10 I=1,N-1
         A1 = -A1
 10      B(I) = A1*A(I)
      B(N) = A(N)
```

```
      CALL FCT(B,X,N,B)
      RETURN
      END


      SUBROUTINE TOCHEB (A,X,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),X(0:N),B(0:N)
Cf2py intent(in) A, X
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) :: N
      CALL FCT(A,X,N,B)
      B1 = 0.5D0/N
      B(0) = B(0)*B1
      B(N) = B(N)*B1
      B1 = 2.D0*B1
      DO 10 I=1,N-1
         B1 = -B1
 10      B(I) = B(I)*B1
      RETURN
      END


      SUBROUTINE DIFFCHEB (A,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),B(0:N)
Cf2py intent(in) A
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) :: N
      A1 = A(N)
      A2 = A(N-1)
      B(N) = 0.D0
      B(N-1) = 2.D0*N*A1
      A1 = A2
      A2 = A(N-2)
      B(N-2) = 2.D0*(N-1)*A1
      DO 10 I=N-2,2,-1
         A1 = A2
         A2 = A(I-1)
 10      B(I-1) = B(I+1)+2.D0*I*A1
      B(0) = 0.5D0*B(2)+A2
      RETURN
      END
```

In [ ]:
```
f2py -c cheb.f --f77flags=-O3 -m cheb
```

In [ ]:
```
import numpy as np
import cheb
```

```
print cheb.__doc__

for N in [8,16,32,64]:
    x=cheb.chebpts(N)
    ex=np.exp(x)
    u=np.sin(ex)
    dudx=ex*np.cos(ex)

    uc=cheb.tocheb(u,x)
    duc=cheb.diffcheb(uc)
    du=cheb.fromcheb(duc,x)
    err=np.max(np.abs(du-dudx))
    print "With N = %d error is %e" % (N,err)
```

1.9.5 §9.9.2

In [ ]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import cheb
%matplotlib notebook

c=1.0
mu=0.1

N=64
x=cheb.chebpts(N)
tau1=N**2
tau2=N**2
t_initial=-2.0
t_final=2.0

def u_exact(t,x):
    """ Exact kink solution of Burgers' equation. """
    return c*(1.0+np.tanh(c*(c*t-x)/(2*mu)))

def mu_ux_exact(t,x):
    """ Return mu*du/dx for exact solution. """
    arg=np.tanh(c*(c*t-x)/(2*mu))
    return 0.5*c*c*(arg*arg-1.0)

def f(x):
    """ Return initial data. """
    return u_exact(t_initial,x)

def g1(t):
    """ Return function needed at left boundary. """
```

```
        return (u_exact(t,-1.0)**2-mu_ux_exact(t,-1.0))

    def g2(t):
        """ Return function needed at right boundary. """
        return mu_ux_exact(t,1.0)

    def rhs(u,t):
        """ Return du/dt. """
        u_cheb=cheb.tocheb(u,x)
        ux_cheb=cheb.diffcheb(u_cheb)
        uxx_cheb=cheb.diffcheb(ux_cheb)
        ux=cheb.fromcheb(ux_cheb,x)
        uxx=cheb.fromcheb(uxx_cheb,x)
        dudt=-u*ux+mu*uxx
        dudt[0]-=tau1*(u[0]**2-mu*ux[0]-g1(t))
        dudt[-1]-=tau2*(mu*ux[-1]-g2(t))
        return dudt

    t=np.linspace(t_initial,t_final,81)
    u_initial=f(x)
    sol=odeint(rhs,u_initial,t,rtol=1.0e-12,atol=1.0e-12,mxstep=5000)
    xg,tg=np.meshgrid(x,t)
    ueg=u_exact(tg,xg)
    err=sol-ueg
    print "With %d points error is %e" % (N,np.max(np.abs(err)))

    plt.ion()
    fig=plt.figure()
    ax=Axes3D(fig)
    ax.plot_surface(xg,tg,sol,rstride=1,cstride=2,alpha=0.1)
    ax.set_xlabel('x',style='italic')
    ax.set_ylabel('t',style='italic')
    ax.set_zlabel('u',style='italic')
```

1.10 Chapter 10

1.10.1 §10.2.1

In [ ]:
```
    import numpy as np
    import matplotlib.pyplot as plt
    %matplotlib notebook

    N=16
    omega=2.0/3.0
    max_its=8

    x=np.linspace(0,1,N+1)
    s1=np.sin(np.pi*x)
```

```
s13=np.sin(13*np.pi*x)
e_old=s1+s13/3.0
e=np.zeros_like(e_old)

plt.ion()
plt.plot(e_old)
for it in range(max_its):
    e[1:-1]=(1.0-omega)*e_old[1:-1]+\
            0.5*omega*(e_old[0:-2]+e_old[2: ])
    plt.plot(e)
    e_old=e.copy()
plt.xlabel('The 17 grid points')
plt.ylabel('The first %d iterates of the error' % N)
```

1.10.2 §10.4.1

In [ ]:
```
%%writefile util.py
# File: util.py:  useful 2D utilities

import numpy as np

def l2_norm(a,h):
    """ Returns L2-norm of a-values with equal spacing h. """
    return h*np.sqrt(np.sum(a**2))

def prolong_lin(a):
    """ Return linear prolongation of 2-dimensional array a. """
    pshape=(2*np.shape(a)[0]-1, 2*np.shape(a)[1] -1)
    p=np.empty(pshape,float)
    p[0: :2,0: :2]=a[0: ,0: ]
    p[1:-1:2,0: :2]=0.5*(a[0:-1,0: ]+a[1: ,0: ])
    p[0: :2,1:-1:2]=0.5*(a[0: ,0:-1]+a[0: ,1: ])
    p[1:-1:2,1:-1:2]=0.25*(a[0:-1,0:-1]+a[1: ,0:-1]+
                           a[0:-1,1: ]+a[1: ,1:])
    return p

def restrict_hw(a):
    """ Return half-weighted restriction of 2-dimensional array a."""
    rshape=(np.shape(a)[0]/2+1, np.shape(a)[1]/2+1)
    r=np.empty(rshape,float)
    r[1:-1,1:-1]=0.5*a[2:-1:2,2:-1:2]+ \
                 0.125*(a[2:-1:2,1:-2:2]+a[2:-1:2,3: :2]+
                        a[1:-2:2,2:-1:2]+a[3: :2,2:-1:2])
    r[0,0: ]=a[0,0: :2]
    r[-1,0: ]=a[-1,0: :2]
    r[0: ,0]=a[0: :2,0]
    r[0: ,-1]=a[0: :2,-1]
    return r
```

```
#-------------------------------------------------------------
if __name__=='__main__':
    a=np.linspace(1,81,81)
    b=a.reshape(9,9)
    c=restrict_hw(b)
    d=prolong_lin(c)
    print " original grid:\n",b
    print "\n with spacing 1 its norm is %g" % l2_norm(b,1)
    print "\n restricted grid:\n",c
    print "\n prolonged restricted grid:\n", d
```

1.10.3 §10.4.2

In [ ]:
```
%%writefile smooth.py
# File: smooth.py: problem-dependent 2D utilities.

import numpy as np

def get_lhs(u,h2):
    """ Return discretized operator L(u).  h2=h**2 for spacing h. """
    w=np.zeros_like(u)
    w[1:-1,1:-1]=(u[0:-2,1:-1]+u[2: ,1:-1]+
                  u[1:-1,0:-2]+u[1:-1,2: ]-
                  4*u[1:-1,1:-1])/h2+u[1:-1,1:-1]**2

    return w


def gs_rb_step(v,f,h2):
    """ Carry out single Gauss-Seidel iteration step on v.
        f is source term, h2 is square of grid spacing.
    """
    u=v.copy()
    res=np.empty_like(v)

    res[1:-1:2,1:-1:2]=(u[0:-2:2,1:-1:2]+u[2: :2,1:-1:2]+
                        u[1:-1:2,0:-2:2]+u[1:-1:2,2: :2]-
                        4*u[1:-1:2,1:-1:2])/h2 +\
                        u[1:-1:2,1:-1:2]**2-f[1:-1:2,1:-1:2]
    u[1:-1:2,1:-1:2]-=res[1:-1:2,1:-1:2]/(
                      -4.0/h2+2*u[1:-1:2,1:-1:2])

    res[2:-2:2,2:-2:2]=(u[1:-3:2,2:-2:2]+u[3:-1:2,2:-2:2]+
                        u[2:-2:2,1:-3:2]+u[2:-2:2,3:-1:2]-
                        4*u[2:-2:2,2:-2:2])/h2 +\
                        u[2:-2:2,2:-2:2]**2-f[2:-2:2,2:-2:2]
```

```
        u[2:-2:2,2:-2:2]-=res[2:-2:2,2:-2:2]/(
                          -4.0/h2+2*u[2:-2:2,2:-2:2])


        res[2:-2:2,1:-1:2]=(u[1:-3:2,1:-1:2]+u[3:-1:2,1:-1:2]+
                            u[2:-2:2,0:-2:2]+u[2:-2:2,2: :2]-
                            4*u[2:-2:2,1:-1:2])/h2 +\
                            u[2:-2:2,1:-1:2]**2-f[2:-2:2,1:-1:2]
        u[2:-2:2,1:-1:2]-=res[2:-2:2,1:-1:2]/(
                          -4.0/h2+2*u[2:-2:2,1:-1:2])


        res[1:-1:2,2:-2:2]=(u[0:-2:2,2:-2:2]+u[2: :2,2:-2:2]+
                            u[1:-1:2,1:-3:2]+u[1:-1:2,3:-1:2]-
                            4*u[1:-1:2,2:-2:2])/h2 +\
                            u[1:-1:2,2:-2:2]**2-f[1:-1:2,2:-2:2]
        u[1:-1:2,2:-2:2]-=res[1:-1:2,2:-2:2]/(
                          -4.0/h2+2*u[1:-1:2,2:-2:2])

        return u

    def solve(rhs):
        """ Return exact solution on the coarsest 3x3 grid."""
        h=0.5
        u=np.zeros_like(rhs)
        fac=2.0/h**2
        dis=np.sqrt(fac**2+rhs[1,1])
        u[1,1]=-rhs[1,1]/(fac+dis)
        return u
```

1.10.4 §10.4.3

In [ ]:
```
%%writefile grid.py
# File: grid.py: linked grid structures and associated algorithms.

import numpy as np
from util import l2_norm, restrict_hw, prolong_lin
from smooth import gs_rb_step, get_lhs, solve

class Grid:
    """
       A Grid contains the structures and algorithms for a
       given level, together with a pointer to a coarser grid.
    """
    def __init__(self,name,steplength,u,f,coarser=None):
        self.name=name
        self.co=coarser         # pointer to a coarser grid
        self.h=steplength       # step length h
```

```
        self.h2=steplength**2   # h**2
        self.u=u                # improved variable array
        self.f=f                # right hand side array

    def __str__(self):
        """ Generate an information string about this level. """
        sme='Grid at %s with steplength = %0.4g\n' % (
                              self.name,self.h)
        if self.co:
            sco='Coarser grid with name %s\n' % self.co.name
        else:
            sco='No coarser grid\n'
        return sme+sco

    def smooth(self,nu):
        """
           Carry out Newton-Raphson/Gauss-Seidel red-black
           iteration u --> u, nu times.
        """
        print 'Relax in %s for %d times' % (self.name,nu)
        v=self.u.copy()
        for i in range(nu):
            v=gs_rb_step(v,self.f,self.h2)
        self.u=v

    def fas_v_cycle(self,nu1,nu2):
        """ Recursive implementation of (nu1,nu2) FAS V-cycle. """
        print 'FAS-V-cycle called for grid at %s\n' % self.name
        # Initial smoothing
        self.smooth(nu1)
        if self.co:
            # There is a coarser grid
            self.co.u=restrict_hw(self.u)
            # Get residual
            res=self.f-get_lhs(self.u,self.h2)
            # Get coarser f
            self.co.f=restrict_hw(res)+get_lhs(self.co.u,self.co.h2)
            oldc=self.co.u
            # Get new coarse solution
            newc=self.co.fas_v_cycle(nu1,nu2)
            # Correct current u
            self.u+=prolong_lin(newc-oldc)
        self.smooth(nu2)
        return self.u

    def fmg_fas_v_cycle(self,nu0,nu1,nu2):
        """ Recursive implementation of FMG-FAS_V-cycle. """
        print 'FMG-FAS-V-cycle called for grid at %s\n' % self.name
```

```
        if not self.co:
            # Coarsest grid
            self.u=solve(self.f)
        else:
            # Restrict f
            self.co.f=restrict_hw(self.f)
            # Use recursion to get coarser u
            self.co.u=self.co.fmg_fas_v_cycle(nu0,nu1,nu2)
            # Prolong to current u
            self.u=prolong_lin(self.co.u)
        for it in range(nu0):
            self.u=self.fas_v_cycle(nu1,nu2)
        return self.u
```

In [ ]:
```
# File rungrid.py: runs the multigrid programme.

import numpy as np
from grid import Grid
from smooth import get_lhs

n_grids=5
size_init=2**n_grids+1
size=size_init-1
h=1.0/size
foo=[]                    # Container list for grids

for k in range(n_grids):
    # Set up list of grids
    u=np.zeros((size+1,size+1),float)
    f=np.zeros_like(u)
    name='Level '+str(n_grids-1-k)
    temp=Grid(name,h,u,f)
    foo.append(temp)
    size/=2
    h*=2

for k in range(1,n_grids):
    # Set up coarser Grid links
    foo[k-1].co=foo[k]

# Check that the initial construction works
for k in range(n_grids):
    print foo[k]

# Set up data for the Numerical Recipes problem
```

```
u_init=np.zeros((size_init,size_init))
f_init=np.zeros_like(u_init)
f_init[size_init/2,size_init/2]=2.0
foo[0].u=u_init
foo[0].f=f_init

foo[0].fmg_fas_v_cycle(1,1,1)

# As a check, get lhs for the final grid
lhs=get_lhs(foo[0].u,foo[0].h2)
print "max abs lhs = ", np.max(np.abs(lhs))

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

plt.ion()
fig1=plt.figure()
ax1=Axes3D(fig1)

xx,yy=np.mgrid[0:1:1j*size_init,0:1:1j*size_init]
ax1.plot_surface(xx,yy,1000*foo[0].u,rstride=1,cstride=1,alpha=0.2)
ax1.set_xlabel('x',style='italic')
ax1.set_ylabel('y',style='italic')
ax1.set_zlabel('1000*u',style='italic')

fig2=plt.figure()
ax2=Axes3D(fig2)
ax2.plot_surface(xx,yy,lhs,rstride=1,cstride=1,alpha=0.2)
ax1.set_xlabel('x',style='italic')
ax2.set_ylabel('y',style='italic')
ax2.set_zlabel('lhs',style='italic')
```

In [ ]: