Lab 17

Key by Ryan Brunt

Problem 1

In [2]:
```
"""
lab17key.py - written by Ryan Brunt
"""
import numpy as np


def deriv(order, accuracy, type, func, vec, h=2e-5):
    """
    Return the numerical derivative of a function

    Parameters
    ----------
    order : integer
        order of derivative. must be 1 or 2

    accuracy : integer
        number of terms in approximations. must be 1,2,3 for forward,
        or 2,4,6 for centered

    type : string
        type of numerical derivative to be calculated. must be 'forward'
        or 'centered'

    func : function
        function whose derivative you wish to calculate

    vec : array_like, dtype=float
        points to evaluate the derivative at

    h : float, optional(default=2e-5)
        Step size

    Returns
    -------
    a : array_like
        vector of derivative values calculated at the points in `vec`

    """
    try:
        a = []
        q = vec.shape[0]
        if type == 'forward':
            if order == 1:
                if accuracy == 1:
                    for i in range(q):
                        p = ((-func(vec[i])+func(vec[i]+h))/h)
                        a.append(p)
                elif accuracy == 2:
                    for i in range(q):
                        p = (((-3./2)*func(vec[i])+2*func(vec[i]+h)-(1./2)*func(vec[i]+2*h))/(h))
                        a.append(p)
                elif accuracy ==3:
                    for i in range(q):
                        p = (((-11./6)*func(vec[i])+3*func(vec[i]+h)-(3./2)*func(vec[i]+2*h))+(1./3)*func(vec[i]+3*
                        a.append(p)
                else:
                    print 'invalid accuracy: must be 1, 2, or 3'
            elif order == 2:
                if accuracy == 1:
                    for i in range(q):
                        p = (func(vec[i])-2*func(vec[i]+h)+func(vec[i]+2*h))/(h**2)
                        a.append(p)
                elif accuracy ==2:
                    for i in range(q):
```

```
                    p = (2*func(vec[i])-5*func(vec[i]+h)+4*func(vec[i]+2*h)-func(vec[i]+3*h))/(h**2)
                    a.append(p)
            elif accuracy ==3:
                for i in range(q):
                    p = ((35./12)*func(vec[i])-(26./3)*func(vec[i]+h)+(19./2)*func(vec[i]+2*h)-(14./3)*func(vec
                    a.append(p)
            else:
                print 'invalid accuracy: must be 1 2 or 3'
        else:
            print 'invalid order'
    elif type == 'centered':
        if order == 1:
            if accuracy == 2:
                for i in range(q):
                    p = ((-1./2)*func(vec[i]-h)+(1./2)*func(vec[i]+h))/h
                    a.append(p)
            elif accuracy ==4:
                for i in range(q):
                    p = ((1./12)*func(vec[i]-2*h)-(2./3)*func(vec[i]-h)+(2./3)*func(vec[i]+h)-(1./12)*func(vec[
                    a.append(p)
            elif accuracy ==6:
                for i in range(q):
                    p = ((-1./60)*func(vec[i]-3*h)+(3./20)*func(vec[i]-2*h)-(3./4)*func(vec[i]-h)+(3./4)*func(v
                    a.append(p)
            else:
                print 'invalid accuracy:must be 2,4, or 6'
        elif order == 2:
            if accuracy == 2:
                for i in range(q):
                    p = (func(vec[i]-h)-2*func(vec[i])+func(vec[i]+h))/(h**2)
                    a.append(p)
            elif accuracy ==4:
                for i in range(q):
                    p = ((-1./12)*func(vec[i]-2*h)+(4./3)*func(vec[i]-h)-(5./2)*func(vec[i])+(4./3)*func(vec[i]+
                    a.append(p)
            elif accuracy ==6:
                for i in range(q):
                    p = ((1./90)*func(vec[i]-3*h)-(3./20)*func(vec[i]-2*h)+(3./2)*func(vec[i]-h)-(49./18)*func(v
                    a.append(p)
            else:
                print 'invalid accuracy: must be 2, 4, or 6'
        else:
            print 'invalid order'
    elif type == 'backward':
        if order == 1:
            if accuracy == 1:
                for i in range(q):
                    p = (func(vec[i])-func(vec[i]-h))/h
                    a.append(p)
            elif accuracy ==2:
                for i in range(q):
                    p = ((3./2)*func(vec[i])-2*func(vec[i]-h)+(1./2)*func(vec[i]-2*h))/(h)
                    a.append(p)
            elif accuracy ==3:
                for i in range(q):
                    p = (((11./6)*func(vec[i])-3*func(vec[i]-h)+(3./2)*func(vec[i]-2*h))-(1./3)*func(vec[i]-3*h
                    a.append(p)
            else:
                print 'invalid accuracy: must be 1, 2, or 3'
        elif order == 2:
            if accuracy == 1:
                for i in range(q):
                    p = (func(vec[i])-2*func(vec[i]-h)+func(vec[i]-2*h))/(h**2)
                    a.append(p)
            elif accuracy ==2:
                for i in range(q):
                    p = (2*func(vec[i])-5*func(vec[i]-h)+4*func(vec[i]-2*h)-func(vec[i]-3*h))/(h**2)
                    a.append(p)
            elif accuracy ==3:
                for i in range(q):
                    p = ((35./12)*func(vec[i])-(26./3)*func(vec[i]-h)+(19./2)*func(vec[i]-2*h)-(14./3)*func(vec
                    a.append(p)
            else:
                print 'invalid accuracy: must be 1, 2, or 3'
        else:
            print 'invalid order'
```

```
        else:
            print 'invalid type'
        return np.asarray(a)
    except AttributeError:
        print 'you n00b. you didnt enter all the parameters'
```

Problem 2

In [3]:
```
import pandas as pd
from itertools import product


def fun17_2(x):
    """
    Analytical derivative is -3 * sin(x)
    """
    return 3 * np.cos(x)


def dfunc17_2(x, order=1):
    """derivative of fun17_2"""
    if order == 1:
        return -3 * np.sin(x)
    elif order == 2:
        return -3 * np.cos(x)

test_x = np.arange(-2, 2, .025)

orders = [1, 2]
acc = [1, 2, 3]
h_vals = [1, 1e-3, 1e-5, 1e-7, 1e-9]
der_types = pd.MultiIndex.from_tuples(list(product(orders, acc, h_vals)))
results = pd.DataFrame(index=der_types, columns=['centered', 'forward'])
results.columns.names = ['errors_by_type']
results.index.names = ['Order', 'Accuracy', 'Step (h)']

for ord_i in orders:
    actual = dfunc17_2(test_x, ord_i)
    for acc_i in acc:
        for h_i in h_vals:
            cent = deriv(ord_i, 2 * acc_i, 'centered', fun17_2, test_x, h=h_i)
            forward = deriv(ord_i, acc_i, 'forward', fun17_2, test_x, h=h_i)

            results.ix[ord_i, acc_i, h_i]['centered'] = np.abs(actual - cent).max()
            results.ix[ord_i, acc_i, h_i]['forward'] = np.abs(actual - forward).max()

results
```

Out[3]:

		errors_by_type	centered	forward
Order	**Accuracy**	**Step (h)**		
1	**1**	**1**	0.4755828	1.458758
		0.001	4.999959e-07	0.0015
		1e-05	7.426371e-11	1.5e-05
		1e-07	3.758555e-09	1.525071e-07
		1e-09	3.465643e-07	6.126496e-07
	2	**1**	0.08876399	0.9332589
		0.001	6.346035e-13	9.999933e-07
		1e-05	5.761036e-11	1.885256e-10
		1e-07	4.868778e-09	1.073327e-08
		1e-09	5.516318e-07	1.320662e-06
	3	**1**	0.0176361	0.6691347
		0.001	7.482903e-13	7.512357e-10
		1e-05	7.683065e-11	2.793474e-10
		1e-07	7.228002e-09	1.948056e-08

		1e-09	6.662133e-07	2.166962e-06
2	**1**	**1**	0.2418138	2.768717
		0.001	2.503239e-07	0.002999979
		1e-05	9.967765e-06	3.802239e-05
		1e-07	0.1004527	0.106455
		1e-09	890.8112	1333.889
	2	**1**	0.03049187	2.413696
		0.001	1.609021e-09	2.750664e-06
		1e-05	1.758813e-05	3.128172e-05
		1e-07	0.1721033	0.3668233
		1e-09	2107.803	3776.379
	3	**1**	0.004586739	2.09567
		0.001	2.134336e-09	9.640946e-09
		1e-05	2.682928e-05	9.166569e-05
		1e-07	0.2289471	0.7884678
		1e-09	2158.798	8217.208

Lab 19

Key by Spencer Lyon

In [1]:
```
from itertools import product
from time import time
import numpy as np
import pandas as pd
```

Problem 1

In [2]:
```
def jacobian(f, x0, h=1e-8, how='centered', *args, **kwargs):
    """
    Numerically estimate the Jacobian matrix for the function f.
    If f operates :math:`\\mathbb{R}^n \\rightarrow \\mathbb{R}^m`, the
    return value of this function will be an (n, m) matrix of
    partial derivatives.

    Parameters
    ----------
    f : function
        The function for which you would like to calculate the jacobian.
        Must accept as an input a numpy array of the same size as
        x0 (also can accept other args passed as *args or **kwargs) and
        return a numpy array of values.

    x0 : array_like, dtype=float, shape=(n,)
        The array of values representing the point where the jacobian
        is to be evaluated.

    h : float, optional(default=1e-8)
        The step size to be used in calculating the derivative.

    how : str
        A string specifying the method of differentiation. Acceptable
        values are centered (c), forwards (f), backward (b).

    *args, **kwargs :
        Other arguments that should be passed to f for evaluation.

    Returns
    -------
    jac : array_like, dtype=float, shape=(n, m)
        The numpy array of partial derivatives representing the jacobian
```

```
    """
    # Make sure we have an array
    x0 = np.asarray(x0)

    # Get n and m
    f0 = f(x0, *args, **kwargs)
    nx = x0.size
    mx = f0.size

    jac = np.zeros((nx, mx))

    ident = np.eye(nx)

    if how.startswith('f'):
        for xi in xrange(nx):
            ei = ident[:, xi]
            jac[xi, :] = (f(x0 + h*ei, *args, **kwargs) - f0) / h

    elif how.startswith('c'):
        for xi in xrange(nx):
            ei = ident[:, xi]
            jac[xi, :] = (f(x0 + h*ei, *args, **kwargs) -
                          f(x0 - h*ei, *args, **kwargs)) / (2 * h)

    elif how.startswith('b'):
        for xi in xrange(nx):
            ei = ident[:, xi]
            jac[xi, :] = (f0 - f(x0 - h*ei, *args, **kwargs)) / h

    return jac.T
```

In [3]:

```
def test_fun(x):
    x1, x2 = x
    return np.array([np.exp(x1)*np.sin(x2) + x2 ** 3,
                     3*x2 - np.cos(x1)])

def real_jac(x):
    x1, x2 = x
    return np.array([[np.exp(x1) * np.sin(x2), np.exp(x1)*np.cos(x2) + 3*x2**2],
                     [np.sin(x1), 3]])

x_grid = np.linspace(-1, 1, 100)
y_grid = np.linspace(-1, 1, 100)

max_err = 0.

methods = 'cbf'
h_vals = [1., 1e-4, 1e-5, 1e-6, 1e-7, 1e-10]

ind = pd.MultiIndex.from_tuples(list(product(list(methods), h_vals)),
                                names=['method', 'h'])
jac_results = pd.DataFrame(index=ind, columns=['max_error', 'time_delta'])

print 'Computing max error in my Jacobian for various how and h values'
for der_type in product(methods, h_vals):
    how, h =  der_type
    max_err = 0.
    for point in product(x_grid, y_grid):

        numerical = jacobian(test_fun, point, how=how, h=h)
        analytical = real_jac(point)
        this_err = np.abs(numerical - analytical).max()
        max_err = this_err if this_err > max_err else max_err

    # Time methods. Should be same for all points so just use 1
    t1 = time()
    _ = jacobian(test_fun, point, how=how, h=h)
    num_time = time() - t1

    t1 = time()
    _ = real_jac(point)
    ana_time = time() - t1

    jac_results.ix[how, h]['max_error'] = max_err
    jac_results.ix[how, h]['time_delta'] = num_time - ana_time
```

```
jac_results
```

```
Computing max error in my Jacobian for various how and h values
```

Out[3]:

		max_error	time_delta
method	**h**		
c	**1.000000e+00**	0.9684898	0.000177145
	1.000000e-04	9.667494e-09	0.0001790524
	1.000000e-05	1.175886e-10	0.0001740456
	1.000000e-06	5.293845e-10	0.0001809597
	1.000000e-07	6.391792e-09	0.0001749992
	1.000000e-10	4.380994e-06	0.0001809597
b	**1.000000e+00**	3.826186	0.0001118183
	1.000000e-04	0.0002845317	0.0001239777
	1.000000e-05	2.845231e-05	0.0001139641
	1.000000e-06	2.845191e-06	0.0001130104
	1.000000e-07	2.870152e-07	0.0001199245
	1.000000e-10	9.935029e-06	0.0001139641
f	**1.000000e+00**	3.826186	0.000109911
	1.000000e-04	0.0002845317	0.0001111031
	1.000000e-05	2.845231e-05	0.0001130104
	1.000000e-06	2.845191e-06	0.0001108646
	1.000000e-07	2.870152e-07	0.0001130104
	1.000000e-10	8.019226e-06	0.0001130104

Problem 2

In [4]:
```
def hessian(f, x0, h=1e-4, *args, **kwargs):
    """
    Numerically estimate the Hessian matrix for the function f.
    It is assumed that f goes from :math:`\mathbb{R}^n
    \rightarrow \mathbb{R}`.

    Parameters
    ----------
    f : function
        The function for which you would like to calculate the jacobian.
        Must accept as an input a numpy array of the same size as
        x0 (also can accept other args passed as *args or **kwargs) and
        return a numpy array of values.

    x0 : array_like, dtype=float, shape=(n,)
        The array of values representing the point where the jacobian
        is to be evaluated.

    h : float, optional(default=1e-4)
        The step size to be used in calculating the derivative.

    *args, **kwargs :
        Other arguments that should be passed to f for evaluation.

    Returns
    -------
    hes : array_like, dtype=float, shape=(n, n)
        The numpy array of mixed second ordre partial derivatives
        at the point x0

    """
    # Make sure we have an array
    x0 = np.asarray(x0)
```

```
    # Get n
    nx = x0.size

    hes = np.zeros((nx, nx))

    ident = np.eye(nx)

    for xi in xrange(nx):
        ei =  ident[:, xi]
        for xj in xrange(nx):
            ej = ident[:, xj]
            hes[xi, xj] = (f(x0 + h*(ei + ej), *args, **kwargs) -
                           f(x0 + h*(ei - ej), *args, **kwargs) -
                           f(x0 + h*(ej - ei), *args, **kwargs) +
                           f(x0 - h*(ei + ej), *args, **kwargs)) / (4*h**2)

    return hes.T
```

In [5]:

```
def test_hes(vec):
    """Rosenbrock banana function"""
    x, y = vec
    return (1 - x)**2 + 100*(y - x**2)**2


def ana_ban(vec):
    x, y = vec
    return np.array([[1200*x**2 - 400*y + 2, -400*x],
                     [-400*x,    200]])

h_vals = [1., 1e-4, 1e-5, 1e-6, 1e-7, 1e-10]
hes_results = pd.DataFrame(index=h_vals, columns=['max_error', 'time_delta'])

hes_x = np.linspace(-2, 2, 100)
hes_y = np.linspace(0, 2, 100)

print 'Computing max error in my Hessian for various how and h values'
for h in h_vals:
    max_err = 0.
    for point in product(hes_x, hes_y):

        numerical = hessian(test_hes, point, h=h)
        analytical = ana_ban(point)
        this_err = np.abs(numerical - analytical).max()
        max_err = this_err if this_err > max_err else max_err

    # Time methods. Should be same for all points so just use 1
    t1 = time()
    _ = hessian(test_hes, point, h=h)
    num_time = time() - t1

    t1 = time()
    _ = ana_ban(point)
    ana_time = time() - t1

    hes_results.ix[h]['max_error'] = max_err
    hes_results.ix[h]['time_delta'] = num_time - ana_time

hes_results
```

```
Computing max error in my Hessian for various how and h values
```

Out[5]:

	max_error	time_delta
1.000000e+00	800	0.0003399849
1.000000e-04	3.043648e-05	0.0003130436
1.000000e-05	0.0029908	0.0003321171
1.000000e-06	0.2581446	0.0003361702
1.000000e-07	27.96591	0.0003869534
1.000000e-10	2.274182e+07	0.0003211498

sympy lab

Key by Spencer Lyon

In [6]:
```
import sympy as sym
import matplotlib.pyplot as plt
%load_ext sympyprinting
```

Problem 1

In [7]:
```
print('7 appears %i times in the first 10000 digits of pi' %
      str(sym.pi.n(10000)).count('7'))
```

```
7 appears 970 times in the first 10000 digits of pi
```

Problem 2

In [8]:
```
a, b, c, x, x1, x2, x3, h, y1, y2, y3 = sym.symbols('a, b, c, x, x1, x2, x3, \
                                                    h, y1, y2, y3')
x2 = x1 + h
x3 = x1 + 2 * h
eqns = (sym.Eq(y1, a + b * x1 + c * x1 ** 2),
        sym.Eq(y2, a + b * x2 + c * x2 ** 2),
        sym.Eq(y3, a + b * x3 + c * x3 ** 2))
soln = sym.solve(eqns, a, b, c)
print('(a, b, c) = ')
soln[a], soln[b], soln[c]
```

```
(a, b, c) =
```

Out[8]:
$$\left(\frac{2h^2y_1 + 3hx_1y_1 - 4hx_1y_2 + hx_1y_3 + x_1^2y_1 - 2x_1^2y_2 + x_1^2y_3}{2h^2}, \quad \frac{-3hy_1 + 4hy_2 - hy_3 - 2x_1y_1 + 4x_1y_2 - 2x_1y_3}{2h^2}, \quad \frac{y_1 - 2y_2 + y_3}{2h^2} \right)$$

Problem 3

In [10]:
```
y = a + b * x + c * x ** 2

soln[x] = (x1 + x2) / 2
y_est1 = sym.simplify(y.subs(soln))
print 'y(x1 + x2) / 2 ->\n'
y_est1
```

```
y(x1 + x2) / 2 ->
```

Out[10]:
$$\frac{3}{8}y_1 + \frac{3}{4}y_2 - \frac{1}{8}y_3$$

In [11]:
```
soln.pop(x)

soln[x] = (x2 + x3) / 2
y_est2 = sym.simplify(y.subs(soln))
print '\ny(x1 + x2) / 2 ->\n'
y_est2
```

```
y(x1 + x2) / 2 ->
```

Out[11]:
$$-\frac{1}{8}y_1 + \frac{3}{4}y_2 + \frac{3}{8}y_3$$

Problem 4

In [12]:
```
t, c, b, g, h = sym.symbols('t, c, b, g, h')

P = sym.Function('P')
m = sym.symbols('m', real=True)
sol = sym.dsolve(sym.Derivative(P(t), t) - m * (c + b * P(t) - g - h*P(t)), P(t))

sol
```

Out[12]:
$$P(t) = \frac{-c + g + e^{C_1 + bmt - hmt}}{b - h}$$

Problem 5

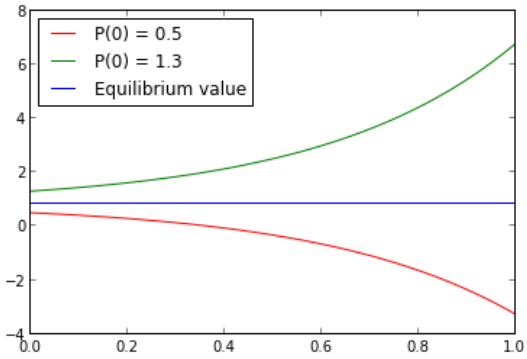
In [13]:
```
b, c, g, h, m= (-1.2, 2, 0.1, 1.1, 1.1)


def p_t(t, p0=1.):
    """
    I used the sympy solution to write this by hand.
    I could have done it with sym.solve and soln.subs,
    but this was easier.
    """
    # Constant of integration
    cc = c - g - (h - b) * p0

    term1 = (c - g) / (h - b)
    term2 = cc * np.exp(m * (h - b) * t) / (h - b)
    return term1 - term2


times = np.linspace(0, 1, 100)
pbar = np.ones_like(times) * (c - g) / (h - b)
plt.plot(times, p_t(times, 0.5), 'r', label=r'P(0) = 0.5')
plt.plot(times, p_t(times, 1.3), 'g', label=r'P(0) = 1.3')
plt.plot(times, pbar, 'b', label='Equilibrium value')
plt.legend(loc=0)
```

Out[13]: Legend



In [ ]: