Question 1

(a) (10 points) Using the second order, centered difference approximation to the first derivative find the Δx that gives the smallest absolute error graphically for the function

$$f(x) = x\sin(x) + 3\cos(x) - x$$

at $x = 2$. Repeat this for the first order forward difference. You do not have to be precise but be within an order of magnitude.

```
In [ ]: ### INSERT CODE HERE
        raise NotImplementedError("Replace this statement with your solution.")
```

(b) (3 points) Why is there an optimal Δx? Why might it be different for each approach? What method would you choose?

Type *Markdown* and LaTeX: α^2

Question 2

(a) (5 points) Analytically compute the interpolating polynomial given by the following data

x_j	0	1	2	3
f_j	0	0	6	24

Type *Markdown* and LaTeX: α^2

(b) (3 points) Show that the interpolating polynomial you found is indeed an interpolant of the data.

Type *Markdown* and LaTeX: α^2

(c) (2 points) Determine the first and second derivative of p and use the result to describe the shape of the function that data may have come from.

(c) (2 points) Determine the first and second derivative of p and use the result to describe the shape of the function that data may have come from.

Type *Markdown* and LaTeX: α^2

(d) (3 points) Plot the interpolating polynomial with the data points included to verify all that you have found. Make sure to properly label your plot. Also mark the maxima and minima you found.

```
### INSERT CODE HERE
raise NotImplementedError("Replace this statement with your solution.")
```

Question 3

Suppose we have a system of non-linear equations we want to solve. Define

$$\vec{x} = [x_0, x_1, \ldots, x_n]$$

and

$$\vec{f}(\vec{x}) = [f_0(\vec{x}), f_1(\vec{x}), \ldots, f_n(\vec{x})] = \vec{0}.$$

We can derive a multi-dimensional version of Newton's method by considering the multi-dimensional Taylor series around the point $\vec{x}$ of each function f_i,

$$f_i(\vec{x} + \vec{\delta}) = f_i(\vec{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}\delta_j + \mathcal{O}(\Delta x^2).$$

where

$$\vec{\delta} = [\delta_0, \delta_1, \ldots, \delta_n]$$

and

$$\Delta x = \max_{i=0,n} \delta_i.$$

$$\Delta x = \max_{i=0,n} \delta_i.$$

As we did before for Newton's method we drop the higher order terms and set $\vec{f}(\vec{x}+\vec{\delta}) = 0$ we can rewrite the above equation as

$$\mathbf{J}(\vec{x})\vec{\delta} = -\vec{f}(\vec{x})$$

where $\mathbf{J}$ is the Jacobian matrix defined by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

This is now a linear system that needs to be solved for the vector $\vec{\delta}$ which can then be used in the update formula

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\delta}.$$

For the rest of the question please consider the system (note that $\log$ is the natural log)

$$\begin{aligned} \sin x + y^2 + \log z - 7 &= 0 \\ 3x + 2^y - z^3 + 1 &= 0 \\ x + y + z - 5.0 &= 0 \end{aligned}$$

(a) (3 points) Analytically find the Jacobian of the system of equations.

Type *Markdown* and LaTeX: α^2

(b) (3 points) Write a function that computes the Jacobian matrix given a vector $\vec{x} = [x, y, z]$.

In [ ]:
```
def jacobian(x):
    ### INSERT CODE HERE
    raise NotImplementedError("Replace this statement with your solution.")

    return J
```

In [ ]:
```
x = (1.0, 1.0, 1.0)
answer = jacobian(x)
true = numpy.array([[ 0.5403023058681398, 2.0, 1.0],
                    [ 3.0, 1.3862943611198906, -3.0],
                    [1.0, 1.0, 1.0]])
numpy.testing.assert_allclose(jacobian((1.0, 1.0, 1.0)), true)
print "Success!"
```

(c) (10 points) Using the function from (b) write a new function that finds the root given an initial guess $\vec{x}_0$ using Newton's method for the system. Use the `numpy.linalg.solve` command to solve the linear system at each step. As a stopping criteria check that the distance of $\vec{f}(\vec{x})$ from $\vec{0}$ is below the given tolerance.

In [ ]:
```
def newton(x, tolerance=1e-10):
    ### INSERT CODE HERE
    raise NotImplementedError("Replace this statement with your solution.")

    return x
```

In [ ]:
```
numpy.testing.assert_allclose(newton(x), numpy.array([ 0.5990537566405669, 2.3959314023778169, 2.005014840981616 ]))
print "Success!"
```

Question 4

Consider the Lennard-Jones potential between two molecules

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]$$

where ϵ and σ are constants, and r is the distance between the molecules.

(a) (10 points) Analytically find the minimum σ/r that minimizes the potential. Plot the potential and the minimum you found as a check.

Type *Markdown* and LaTeX: α^2

(b) (10 points) Using Golden section search compute the minimum of the potential numerically down to a bracket size of 10^{-6}. Plot the convergence to the true solution you found (the location) above at each step of the algorithm. Also plot the potential and the minimum you found. You can choose any value of ϵ you want.

In [ ]:
```
### INSERT CODE HERE
raise NotImplementedError("Replace this statement with your solution.")
```

Question 5

The equation for an ellipse is $x^2/a^2 + y^2/b^2 = 1$. The general equation for arc-length and therfore the circumference of an ellipse can be computed using

$$S = 2\int_{-a}^{a} \sqrt{1 + (\mathrm{dy}/\mathrm{dx})^2}\,dx.$$

(a) (3 points) We can approximate the circumference by following a paper by Srinivasa Ramanujan in 1914 [1] where

$$S \approx \pi(a+b)\left(1 + \frac{3h}{10 + \sqrt{4 - 3h}}\right)$$

which is h^5 order accurate where

$$h = \frac{(a-b)^2}{(a+b)^2}.$$

Implement a function that computes the approximation using Ramanujan's result.

1. Ramanujan, Srinivasa, (1914). "Modular Equations and Approximations to π". Quart. J. Pure App. Math. 45: 350-372.

In [ ]:
```
def S_ramanujan(a, b):
    ### INSERT CODE HERE
    raise NotImplementedError("Replace this statement with your solution.")

    return S
```

In [ ]:
```
x = numpy.random.random((2)) * 2.0 + 1.0
x.sort()
import scipy.special
S_exact = lambda a, b: 4.0 * a * scipy.special.ellipe(1.0 - b**2 / a**2)
computed = S_ramanujan(x[1], x[0])
true = S_exact(x[1], x[0])
print "Computed = %s" % computed
print "True = %s" % true
print "Error = %s" % (numpy.abs(computed - true))
numpy.testing.assert_allclose(computed, true)
print "Success!"
```

(b) (10 points) Another way to compute the circumference is to use a series due to Ivory and Bessel (you can find the original papers online which may be worth a look). One way to write this series is

$$S = \pi(a+b) \sum_{n=0}^{\infty} \binom{0.5}{n}^2 h^n \quad \text{with} \quad h = \frac{(a-b)^2}{(a+b)^2}$$

where $\binom{0.5}{n}$ is the binomial coefficient which can be computed via a `scipy` function. Write a function to compute this series to a point where the difference between the partial sums is less than $\epsilon_{\text{machine}}$.

```
In [ ]: def S_ivory(a, b):
            ### INSERT CODE HERE
            raise NotImplementedError("Replace this statement with your solution.")

            return S
```

```
In [ ]: x = numpy.random.random((2)) * 2.0 + 1.0
        x.sort()
        import scipy.special
        S_exact = lambda a, b: 4.0 * a * scipy.special.ellipe(1.0 - b**2 / a**2)
        computed = S_ivory(x[1], x[0])
        true = S_exact(x[1], x[0])
        print "Computed = %s" % computed
        print "True = %s" % true
        print "Error = %s" % (numpy.abs(computed - true))
        numpy.testing.assert_allclose(computed, true)
        print "Success!"
```

(c) (5 points) Derive an expression for the integrand using implicit differentiation.

Type *Markdown* and LaTeX: α^2

(d) (10 points) Write a function that computes the integral to a given tolerance. Note that this may converge very slowly so use a high enough order quadrature rule so that the computation takes less than a minute to do (this will result in an error otherwise). Suggested largest N is 1000. If the integration does not succeed raise a `ValueError`. Use the `S_exact` function used in the tests above to evaluate the tolerance.

In [ ]:
```
def S_direct(a, b, tolerance):
    ### INSERT CODE HERE
    raise NotImplementedError("Replace this statement with your solution.")

    return S
```

In [ ]:
```
x = numpy.random.random((2)) * 2.0 + 1.0
x.sort()
import scipy.special
S_exact = lambda a, b: 4.0 * a * scipy.special.ellipe(1.0 - b**2 / a**2)
tolerance = 1e-1
computed = S_direct(x[1], x[0], tolerance)
true = S_exact(x[1], x[0])
print "Computed = %s" % computed
print "True = %s" % true
print "Error = %s" % (numpy.abs(computed - true))
numpy.testing.assert_allclose(computed, true, atol=tolerance)
print "Success!"
```

(e) (10 points) Note that we have been using a special function from SciPy called `ellipe`. This function computes the elliptic integral of the second kind

$$E(e) = \int_0^{\pi/2} \sqrt{1 - e^2 \sin^2 \theta} d\theta$$

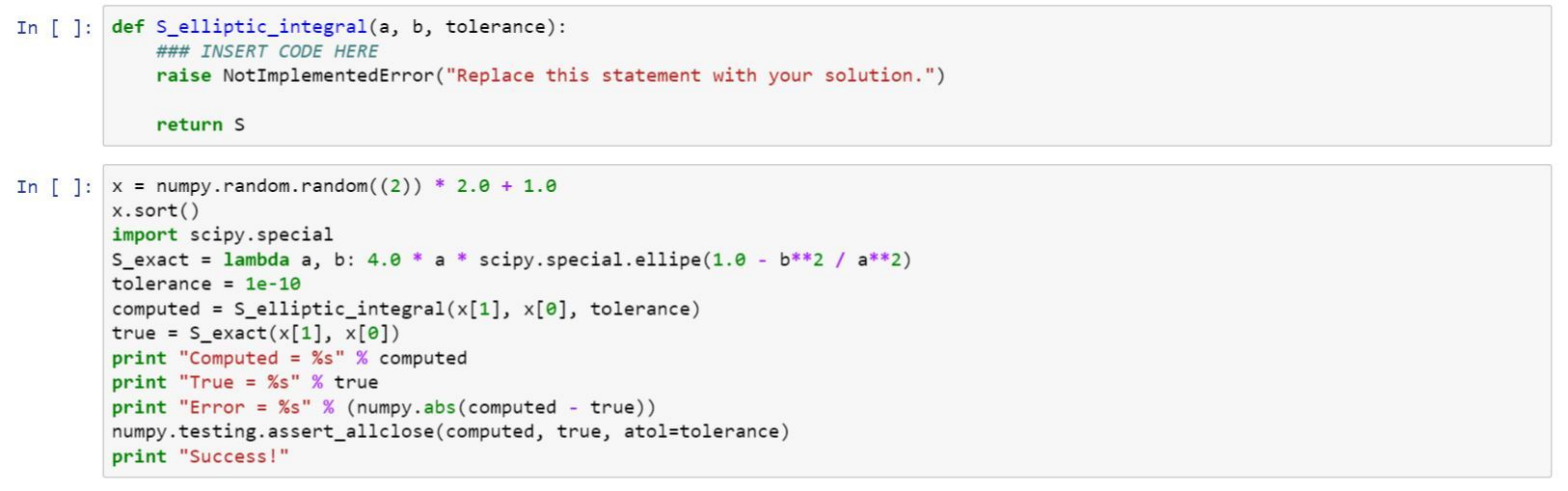
which is the arc-length over one quadrant of the ellipse. The value e is the eccentricity defined as

$$e = \sqrt{1 - b^2 / a^2}$$

where here we need to define a as the major-axis and b the minor-axis of the ellipse (this definition is slightly different than the one in SciPy which we have been accommodating). Elliptic integrals are a broad class of special functions that all arose from attempts at computing things related to ellipses.

Again write a function that computes the circumference of an ellipse to a provided tolerance but using this formulation of the problem. If the integration does not succeed raise a `ValueError`. Remember that computing the function above only gives you one quarter of the circumference and to find the total circumference you want to use

$$S = 4aE(e).$$



```
In [ ]: def S_elliptic_integral(a, b, tolerance):
            ### INSERT CODE HERE
            raise NotImplementedError("Replace this statement with your solution.")

            return S
```

```
In [ ]: x = numpy.random.random((2)) * 2.0 + 1.0
        x.sort()
        import scipy.special
        S_exact = lambda a, b: 4.0 * a * scipy.special.ellipe(1.0 - b**2 / a**2)
        tolerance = 1e-10
        computed = S_elliptic_integral(x[1], x[0], tolerance)
        true = S_exact(x[1], x[0])
        print "Computed = %s" % computed
        print "True = %s" % true
        print "Error = %s" % (numpy.abs(computed - true))
        numpy.testing.assert_allclose(computed, true, atol=tolerance)
        print "Success!"
```