Assesment

October 9, 2013

```
In [1]: import numpy as np
        import math as math
        from matplotlib import pyplot as plt
        from IPython.core.display import Image
        %matplotlib inline
```

1 Scientific Computing: Assesment

1.1 Integration

1.1.1 Implement a trapezium integration routine on the function $\cos(x)$ from 0 to 10.

```
In [2]: def trap_rule_single_step(f, t, dt):
            return 0.5*dt*(f(t) + f(t+dt))
```

```
In [3]: def trapezium_rule(f, a, b, dt):

            x = np.arange(a, b, dt)

            integral = 0
            for i in range(len(x)):
                integral += trap_rule_single_step(f, x[i], dt)

            return integral
```

```
In [4]: trapezium_rule(np.cos, 0, 10, 0.001)
```

```
Out[4]: -0.54402106555427943
```

1.1.2 Write a bisection routine to find the roots of the function $x^2 - 200x + 10^4 - 10 - 6$. Try to use while loop and else loop. How many iterations have you used if you initially bracketed the zero between 100 and 2000? could you have guessed such a number even before running the routine?

```
In [5]: def bisection1(f, a, b, tol_y = 1e-15, tol_x = 1e-10, max_iterations = 100000):

            iterations = 0
            aprox = 0.5*(a + b)
            while iterations < max_iterations:

                if (abs(f(aprox)) < tol_y) or ((b -a)/2.0 < tol_x):
                    #print "Iterations: ", iterations
                    #print "Tolerance: ", tol
                    return aprox
```

```
            iterations += 1

            if np.sign(f(aprox)) == np.sign(f(a)):
                a = aprox
            elif np.sign(f(aprox)) == np.sign(f(b)):
                b = aprox

            aprox = 0.5*(a + b)

        return aprox
```

```
In [6]: def f(x):
            return x**2.0 - 200.0*x +1e4 - 1e-10
```

```
In [7]: bisection1(f, 100, 2000)
```

```
Out[7]: 100.00000995669609
```

With starting points of 100 and 200, it took 27 iterations to converge. The tolerance was set to 10^{-10}. Since the size of the interval $\sim \delta/2^n$ we expect the to reach a value colse to the root relativley fast, however, once we are close to the root, the process slows down considerably. If we now take the upper bracket to be 200, it only takes 5 iterations less.

```
In [8]: bisection1(f, 100, 200)
```

```
Out[8]: 100.00000995669325
```

```
In [9]: (np.log(1900) + 10*np.log(10) ) / np.log(2)
```

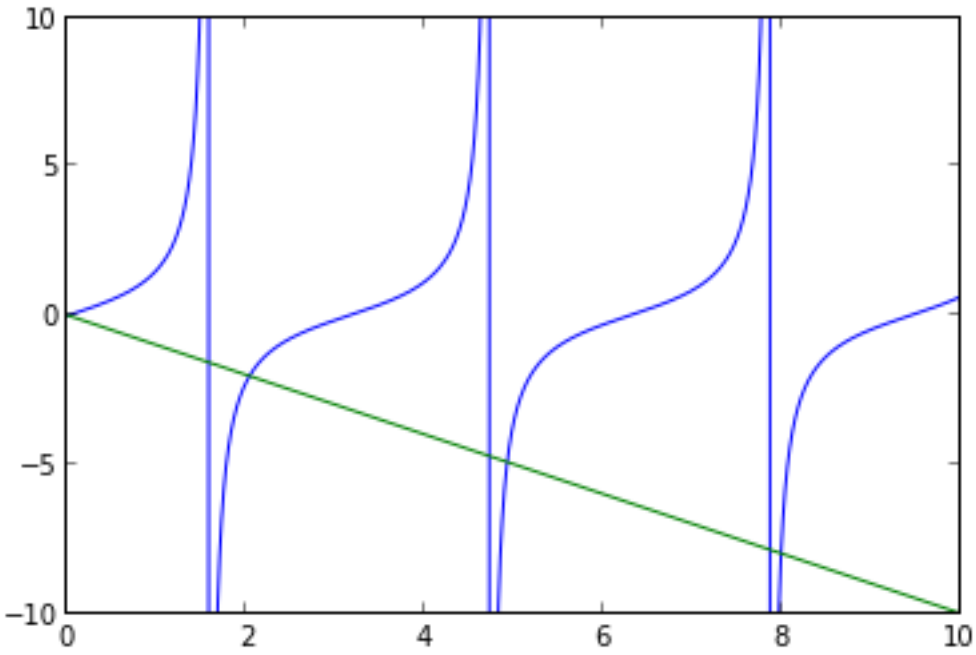
```
Out[9]: 44.11106465209194
```

1.1.3 Write a bisection method to find the first positive 100 roots of the equation $-y = \tan(y)$ with a 10^{-10} precision. Be aware of the tangent function, try to write a code that allows you to have some control over how steep the tangent function becomes as the order of the roots increases.

The roots we are trying to find are the intersections of the following cuves:

```
In [10]: x = np.arange(-3, 13, 0.0001)
         plt.plot(x, np.tan(x), x, -x)
         plt.ylim([-10, 10])
         plt.xlim([0, 10])
```

```
Out[10]: (0, 10)
```

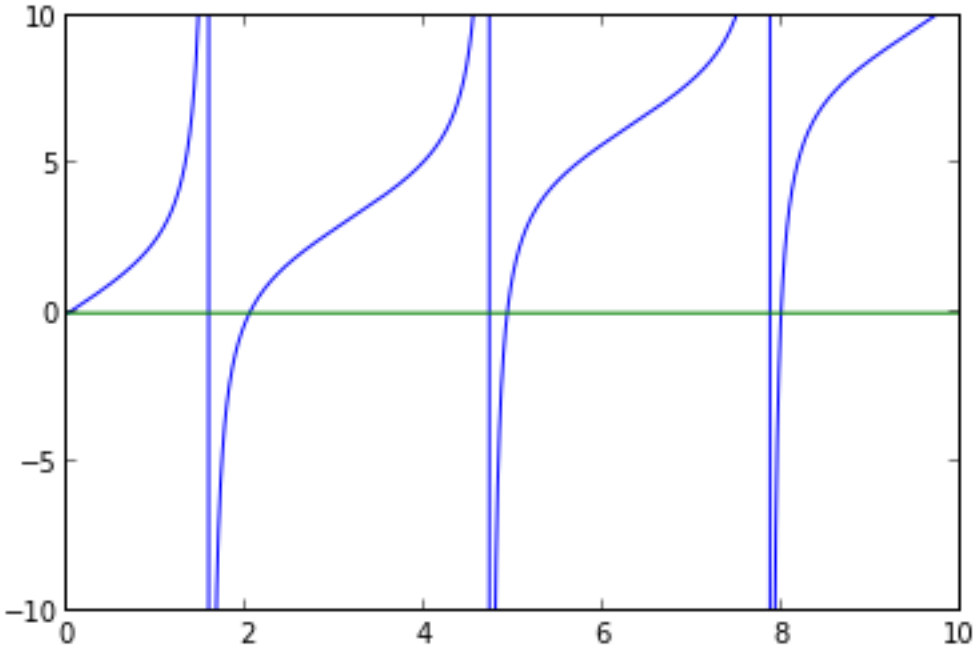



```
In [11]: def f1(y):
             return np.tan(y) + y
```

Which is equivalent of findin the roots of the function $f(x) = \tan(x) + x$:

```
In [12]: plt.plot(x, f1(x), x, np.zeros(len(x)))
         plt.ylim([-10, 10])
         plt.xlim([0, 10])
```

```
Out[12]: (0, 10)
```



Since the tangent function gets steeper at further intersections, we can tell that the intersections will happen colse to $x = (k+1)\pi/2$. So to find the first 100 roots, we can take brackets close to those points.

To keep track of the steepness of the tangent function we can calculate the derivative at te point, since we know that $\frac{d}{dx}\tan(x) = \sec^2(x)$.

In [13]:
```
def bisection2(f, a, b, tol_y = 1e-15, max_iterations = 500000):

    iterations = 0
    aprox = 0.5*(a + b)
    while iterations < max_iterations:

        if abs(f(aprox)) < tol_y:
            #print "Iterations: ", iterations
            #print "Tolerance: ", tol
            return aprox

        iterations += 1

        if np.sign(f(aprox)) == np.sign(f(a)):
            a = aprox
        elif np.sign(f(aprox)) == np.sign(f(b)):
            b = aprox

        aprox = 0.5*(a + b)

    #print "While loop finished. Iterations = ", iterations
    return aprox
```

In [14]:
```
%time
tolerance = 1e-10
delta = 0.01
roots = []
b = 0.5*np.pi + np.pi/4.0

for k in range(201):
    #print "finding {}th root".format(k)

    dist_to_discont = abs( (k+1)*np.pi*0.5 - b)

    a =  (k+1)*np.pi*0.5 + 0.001*dist_to_discont
    #print "starting while"

    #counter = 0
    #while np.sign(f1(a)) == np.sign(f1(b)):

     ##   dist_to_discont = abs( (k+1)*np.pi*0.5 - a)
     #   a -=dist_to_discont*0.5
      #  dist_to_discont = abs( (k+1)*np.pi*0.5 - a)
       #
        #counter += 1

    #print "found appropriate lower bound\n"
```

```
            roots.append(bisection2(f1, a, b, tol_y = tolerance))
            #gradient = (f(b + tolerance) - f(roots[-1]))/tolerance
            #gradient = 1/np.cos(b)**2.0 + 1

            b = roots[-1] + np.pi*0.5
```

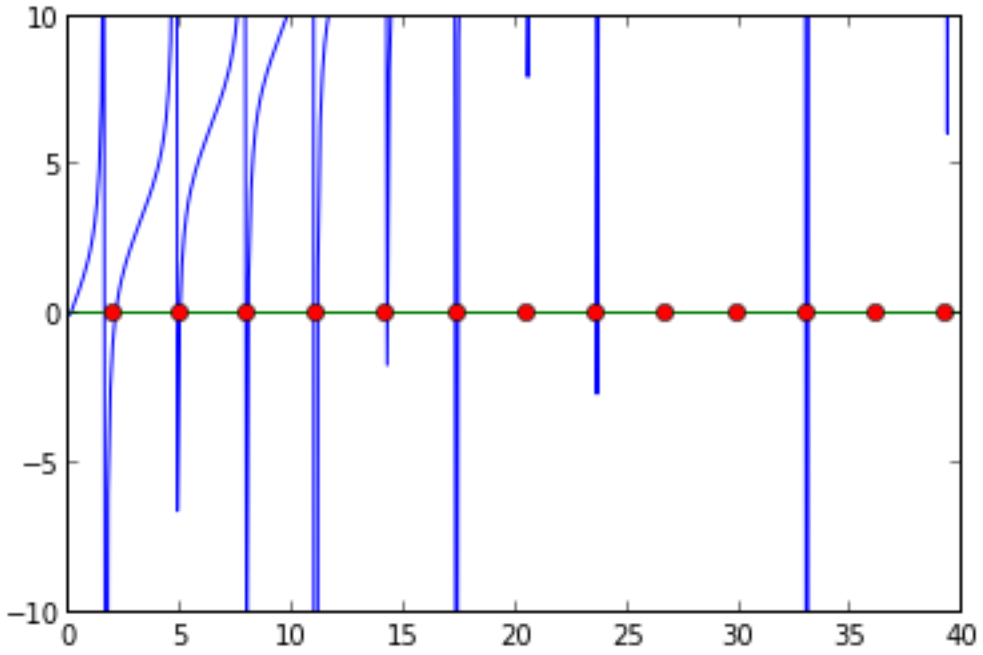
```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 8.82 µs
```

```
In [114]: roots2 = [roots[i] for i in range(len(roots)) if i%2 == 0 ]
          print roots2
```

```
[2.0287578381109945, 4.913180439433632, 7.97866571241299, 11.085538406497381, 14.207436725191254, 17.33
```

```
In [113]: plt.plot(x, f1(x), x, np.zeros(len(x)), roots2, np.zeros(len(roots2)), "o")
          plt.ylim([-10, 10])
          plt.xlim([0, 40])
```

```
Out[113]: (0, 40)
```




```
In [17]: f1(roots[2])
```

```
Out[17]: -3.2711611197555612e-11
```

1.1.4 How would you go about creating a stable routine to calculate $\mathcal{G}_n$? In such case, what would be the value of $\mathcal{G}_{500}$

```
In [18]: def g(n):
             if n == 0:
```

```
            return (np.e -1.0)/np.e
        else:
            return 1.0 - n*g(n-1)
```

```
In [19]: values = [g(x) for x in range(25)]
```

```
In [20]: for i in range(len(values)):
             if values[i] > 0.0:
                 continue
             else:
                 print "The value becomes negative when n = ", i
                 break
```

```
The value becomes negative when n =  18
```

What would the value of $\mathcal{G}_{500}$ be? We can rewrite the recursion formula in terms of $\mathcal{G}_n$ for the $(n-1)$th term: $\mathcal{G}_{n-1} = \frac{1}{n}(1 - \mathcal{G}_n)$

```
In [125]: def g2(n):
              if n == 1000:
                  return 0
              else:
                  return (1 - g2(n+1))/float(n)
```

```
In [126]: g2(500)
```

```
Out[126]: 0.0019960159204766978
```

1.1.5 Evaluate the sum, $\sum_{n=1}^{n=\infty} \frac{1}{n+x}$, with errors less than 1 in the 10th decimal digit: For values $0 \leq x \leq 1$ and 0.1 as a step size.

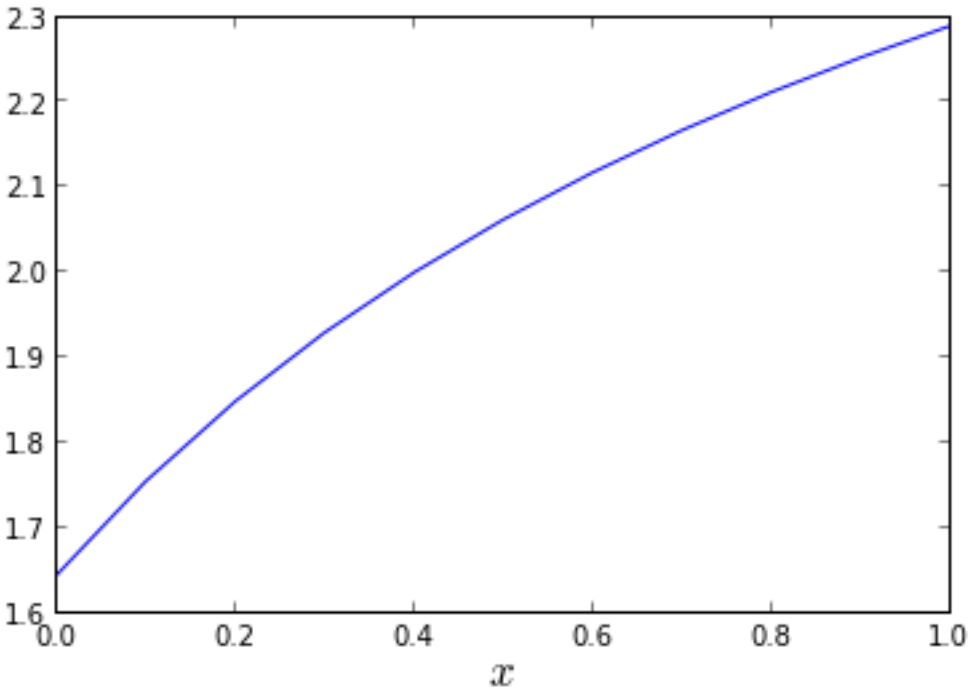
```
In [22]: def sumation(x, tolerance = 1e-10):

             error = 1
             suma = (np.pi**2)/6
             n = 1

             while error > tolerance:

                 new_suma = suma + float(x)/(n**3 + x*n**2)
                 error = abs(suma - new_suma)

                 suma = new_suma
                 n += 1
             #print "n = ", n
             return suma
```

```
In [23]: sumation(0.4)
```

```
Out[23]: 2.0002902536584526
```

```
In [24]: x = np.arange(0,1.1,0.1)
         vals = [sumation(i) for i in x]

         plt.plot(x, vals)
         #plt.ylabel("Series")
         plt.xlabel("$x$", fontsize=17)
```

Out[24]: <matplotlib.text.Text at 0x7f72fc1c80d0>



1.1.6 Ordinary differential equations

In [25]:
```
def euler(f, x, dt):

            x += dt*f(x)
            return x
```

In [26]:
```
def rk4(f, x, dt):

            k1 = f(x)
            k2 = f(x + 0.5*dt*k1)
            k3 = f(x + dt*0.5*k2)
            k4 = f(x + dt*k3)

            x += (dt/6.)*(k1 +2*k2 + 2*k3 + k4)

            return x
```

In [27]:
```
def integrate(f, x_0, final_time, dt, method):
            if type(x_0) == type(np.array([])):
                x = x_0.copy()
            else:
                x = x_0

            t = 0
            T = []
```

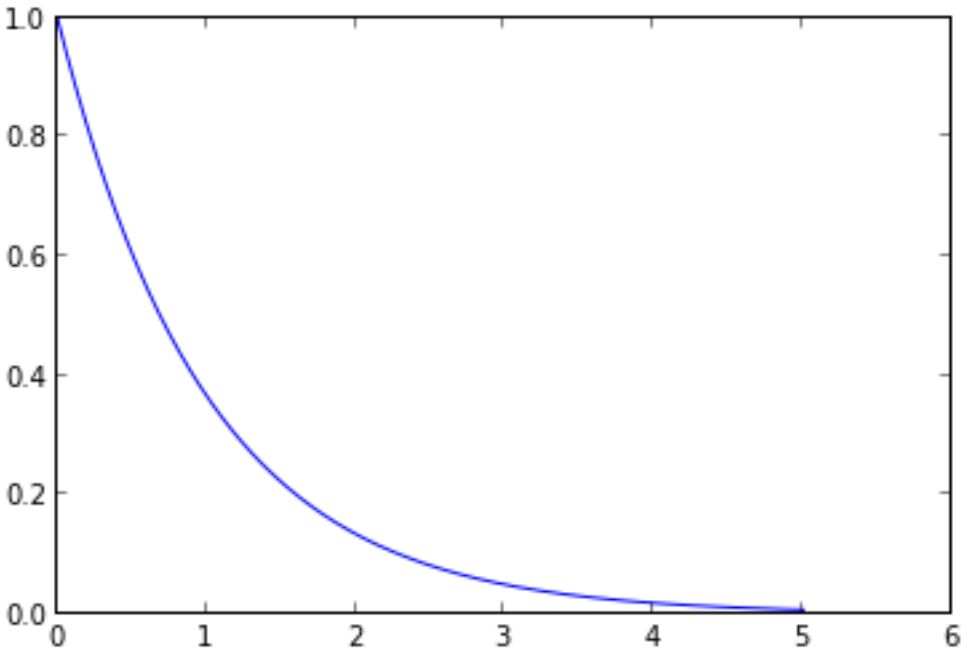
```
            T.append(x)

            while abs(t) < abs(final_time):
                x = method(f,x,dt)
                T.append(x)
                t += dt

            return np.array(T)
```

In [28]:
```
def f2(x):
    return -x
```

In [29]:
```
x = np.arange(0, 5.02, 0.01)
a = integrate(f2, 1.0, 5, 0.01, euler)
```

In [30]:
```
plt.plot(x, a)
```

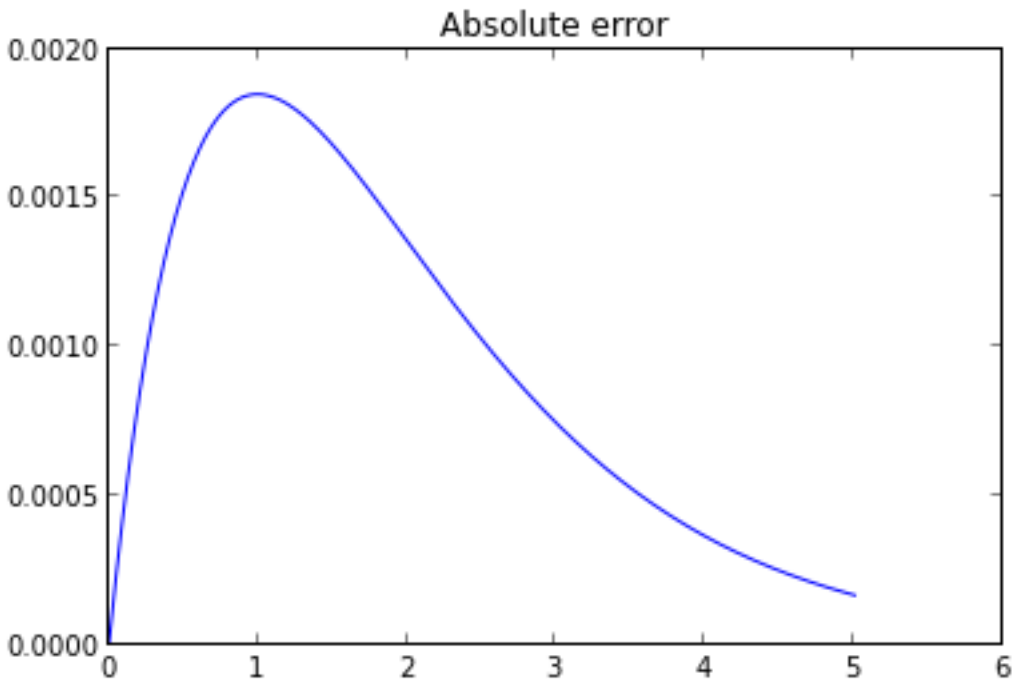
Out[30]: [<matplotlib.lines.Line2D at 0x7f72fc4f4e90>]



In [31]:
```
abs_error = np.abs( np.exp(-x)- a)
rel_error = np.abs( (np.exp(-x) - a)/np.exp(-x) )
```

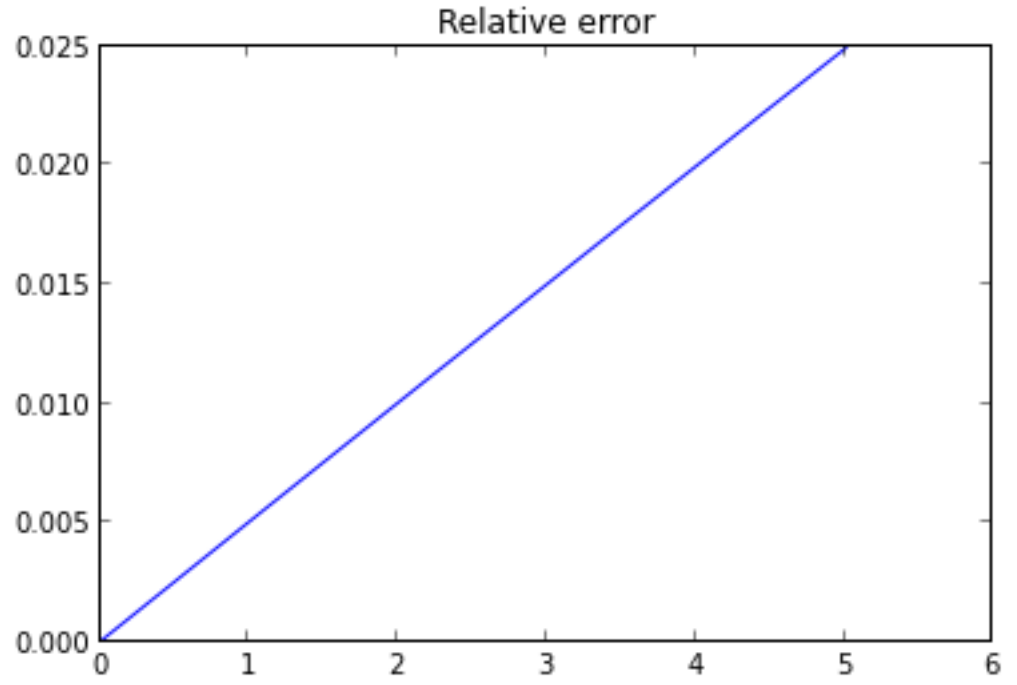
In [32]:
```
plt.plot(x, abs_error)
plt.title("Absolute error")
```

Out[32]: <matplotlib.text.Text at 0x7f72fc4fea90>



In [33]:
```
plt.plot(x, rel_error)
plt.title("Relative error")
```

Out[33]: <matplotlib.text.Text at 0x7f72fce47950>



For the differential equation $\frac{dP(t)}{dt} = P^2(t) + 1$, we first define the function:

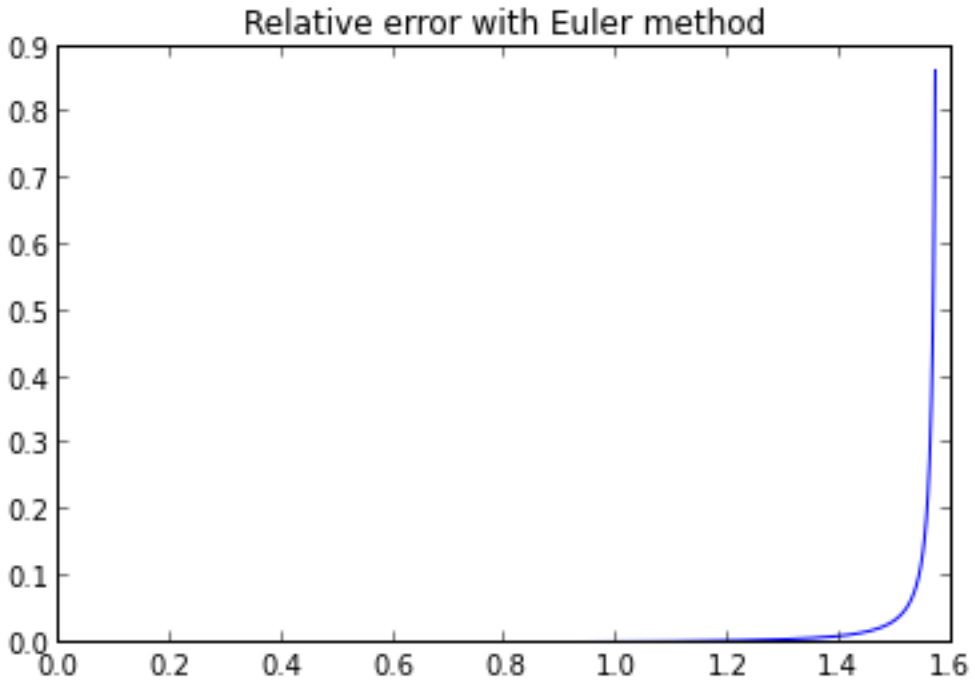
```
In [34]: def f3(x):
             return x**2 + 1
```

We now integrate and plot the relative error:

```
In [35]: x = np.arange(0, np.pi*9999/20000, 0.001 )

         vals = integrate(f3, 0.0, np.pi*9999/20000 - 0.001, 0.001, euler)

         rel_error_euler = np.abs((np.tan(x) - vals)/np.tan(x))
         plt.plot(x, rel_error_euler)
         plt.title("Relative error with Euler method")
```

```
-c:5: RuntimeWarning: invalid value encountered in divide
```

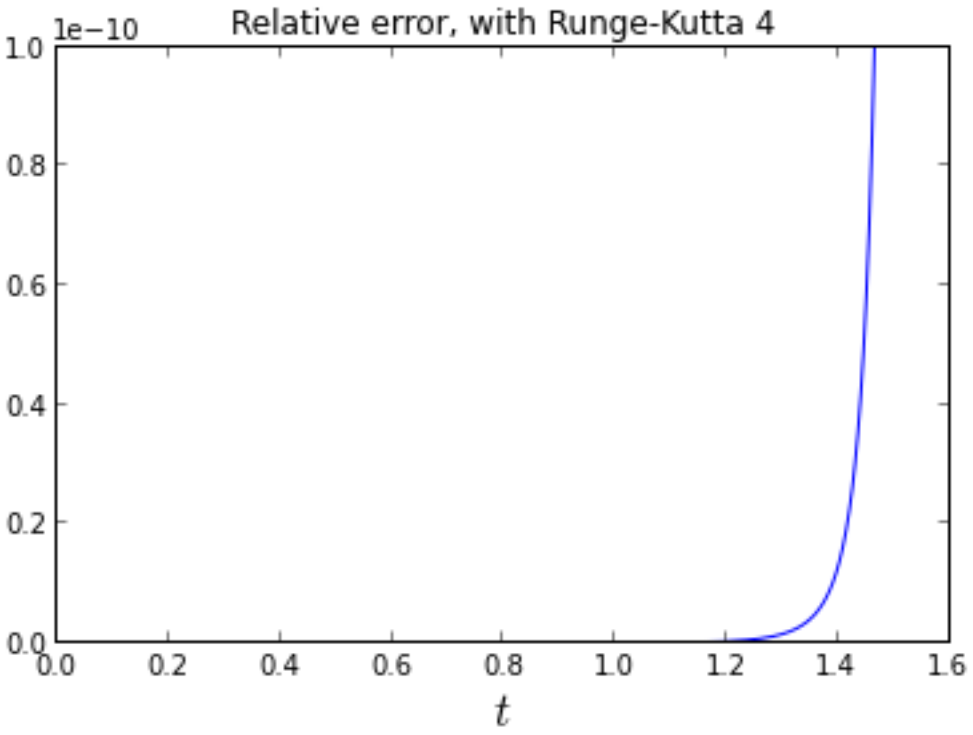
```
Out[35]: <matplotlib.text.Text at 0x7f72fc3b66d0>
```



If we now repeat the procedure above using a higher order integration method, in this case the Runge-Kutta method of order 4 accuracy, we obtain:

```
In [36]: x = np.arange(0, np.pi*9999/20000, 0.001 )

         vals = integrate(f3, 0.0, np.pi*9999/20000 - 0.001, 0.001, rk4)

         rel_error_rk = np.abs((np.tan(x) - vals)/np.tan(x))
         plt.plot(x, rel_error_rk)
         plt.ylim([0,1e-10])
         plt.title("Relative error, with Runge-Kutta 4")
         plt.xlabel("$t$", fontsize = 17)
```

Out[36]: <matplotlib.text.Text at 0x7f72fce50110>



Notice that the scale on this graph is $\times 10^{-10}$.

1.1.7 The linear gain-loss equation

We first construct the matrix for the gain-loss equation

We now define the derivative:

```
In [37]: def gain_loss(x):
             return np.dot(A,x)

In [38]: def integrate(f, x_0, final_time, dt, method):

             x = x_0.copy()


             t = 0
             T = []
             T.append(x)

             while abs(t) < abs(final_time):
                 x = method(f,x,dt)
                 T.append(x)
                 t += dt

             return np.array(T)
```

```
In [39]: length = 11
         A = -2*np.eye(length)

             #boundary conditions:
         A[0,0] = -1
         A[0,1] = 1

         A[-1,-1] = -1
         A[-1, -2] = 1

         for i in range(1,np.shape(A)[0]-1):
             A[i,i-1] = 1
             A[i,i+1] = 1

         A = 0.1*A
         print A
```

```
[[-0.1  0.1 -0.  -0.  -0.  -0.  -0.  -0.  -0.  -0.  -0. ]
 [ 0.1 -0.2  0.1 -0.  -0.  -0.  -0.  -0.  -0.  -0.  -0. ]
 [-0.   0.1 -0.2  0.1 -0.  -0.  -0.  -0.  -0.  -0.  -0. ]
 [-0.  -0.   0.1 -0.2  0.1 -0.  -0.  -0.  -0.  -0.  -0. ]
 [-0.  -0.  -0.   0.1 -0.2  0.1 -0.  -0.  -0.  -0.  -0. ]
 [-0.  -0.  -0.  -0.   0.1 -0.2  0.1 -0.  -0.  -0.  -0. ]
 [-0.  -0.  -0.  -0.  -0.   0.1 -0.2  0.1 -0.  -0.  -0. ]
 [-0.  -0.  -0.  -0.  -0.  -0.   0.1 -0.2  0.1 -0.  -0. ]
 [-0.  -0.  -0.  -0.  -0.  -0.  -0.   0.1 -0.2  0.1 -0. ]
 [-0.  -0.  -0.  -0.  -0.  -0.  -0.  -0.   0.1 -0.2  0.1]
 [-0.  -0.  -0.  -0.  -0.  -0.  -0.  -0.  -0.   0.1 -0.1]]
```

We now construct the initial condition:

```
In [40]: p_0 = np.zeros(11)
         p_0[5] = 1
         p_0
```

```
Out[40]: array([ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.])
```

We now integrate with Runge-Kutta, however, the code is slightly changed to be able to handle vectors:

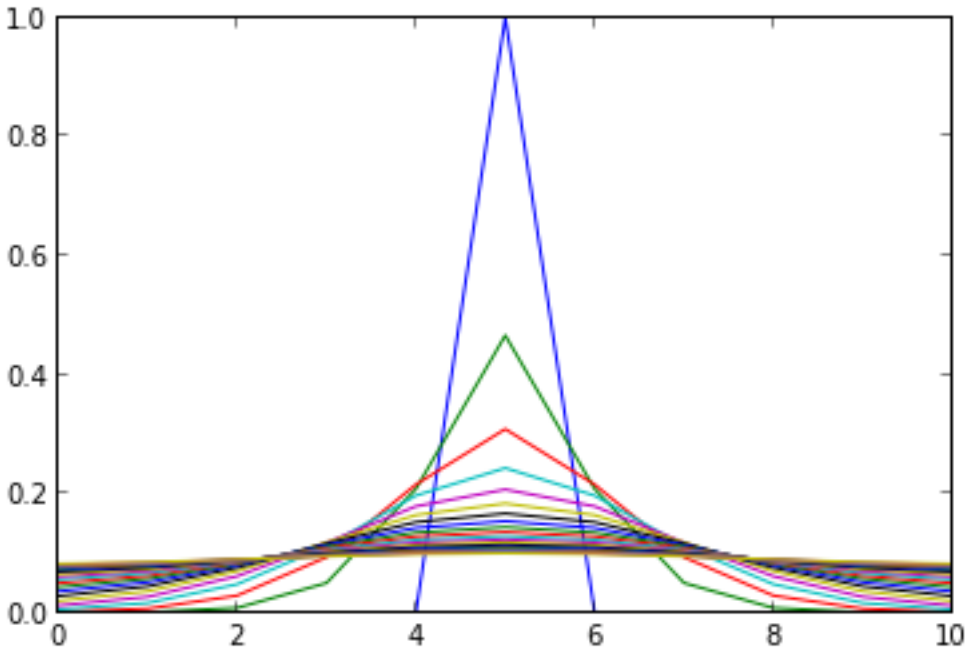
```
In [41]: def rk4(f, x_0, dt):
             """Does 1 step of the integration using Ruge-Kutta method. Works for vectors """
             x = x_0.copy()

             k1 = f(x)
             k2 = f(x + 0.5*dt*k1)
             k3 = f(x + dt*0.5*k2)
             k4 = f(x + dt*k3)

             x += (dt/6.)*(k1 +2*k2 + 2*k3 + k4)

             return x
```

```
In [42]: p = integrate(gain_loss, p_0, 99, 1, rk4)
```

```
In [43]: p[3]
```

```
Out[43]: array([  1.18815034e-05,   1.89061935e-04,   2.52199737e-03,
                  2.54579892e-02,   1.72141154e-01,   5.99355832e-01,
                  1.72141154e-01,   2.54579892e-02,   2.52199737e-03,
                  1.89061935e-04,   1.18815034e-05])
```

```
In [44]: M = range(11)

         for i in range(len(p)):

             if i%5 == 0:
                 plt.plot(M, p[i])
```



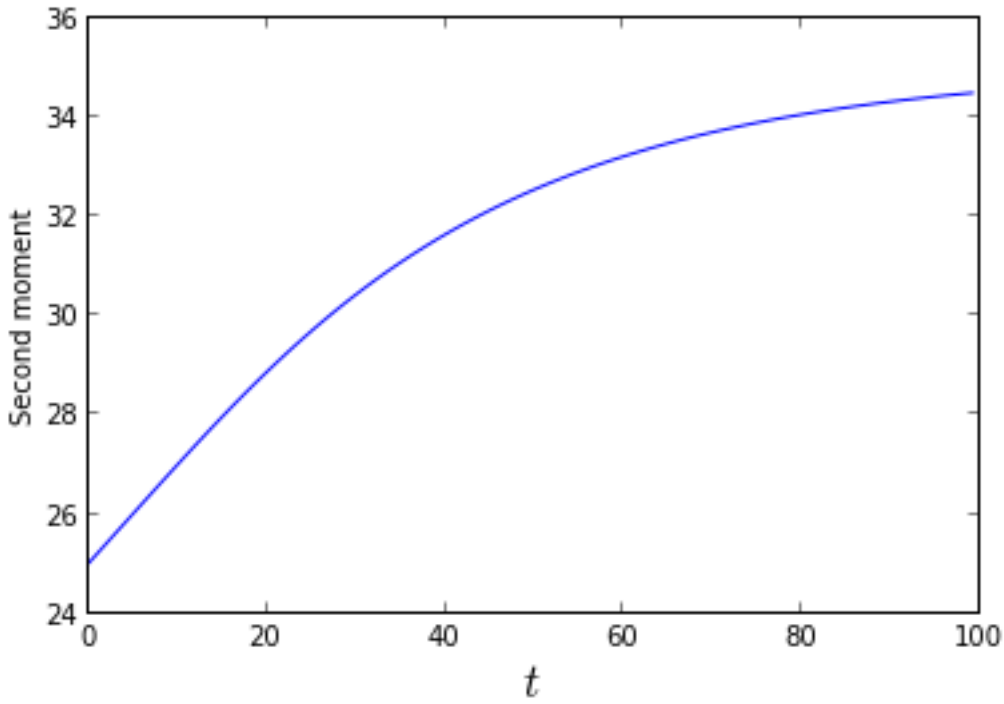
To plot the second moment we need to calculate $\sum_{m=1}^{11} P_{m(t)}$:

```
In [45]: for p_i in p:
             p_i = np.array(p_i)

         M = np.array(M)
```

```
In [46]: sigma = []
         for p_i in p:
             sigma.append(sum(M*M*p_i))
```

```
In [47]: plt.plot(range(100), sigma)
         plt.xlabel("$t$", fontsize = 17)
         plt.ylabel("Second moment")
```

```
Out[47]: <matplotlib.text.Text at 0x7f72fc50aa90>
```



1.1.8 Random Walks

```
In [86]: import numpy.random as rand
```

We construct a matrix of random numbers with dimensions 50000×200 to have our walks.

```
In [87]: walks = rand.rand(50000, 200);
```

```
In [88]: def choose(x):
             if x <= 0.5:
                 return -1
             else:
                 return 1
```

```
In [89]: walks = [map(choose, w) for w in walks]
```

```
In [90]: walks = np.array(walks)
```

```
In [91]: def position(w, t):
             return sum(w[:t])
```

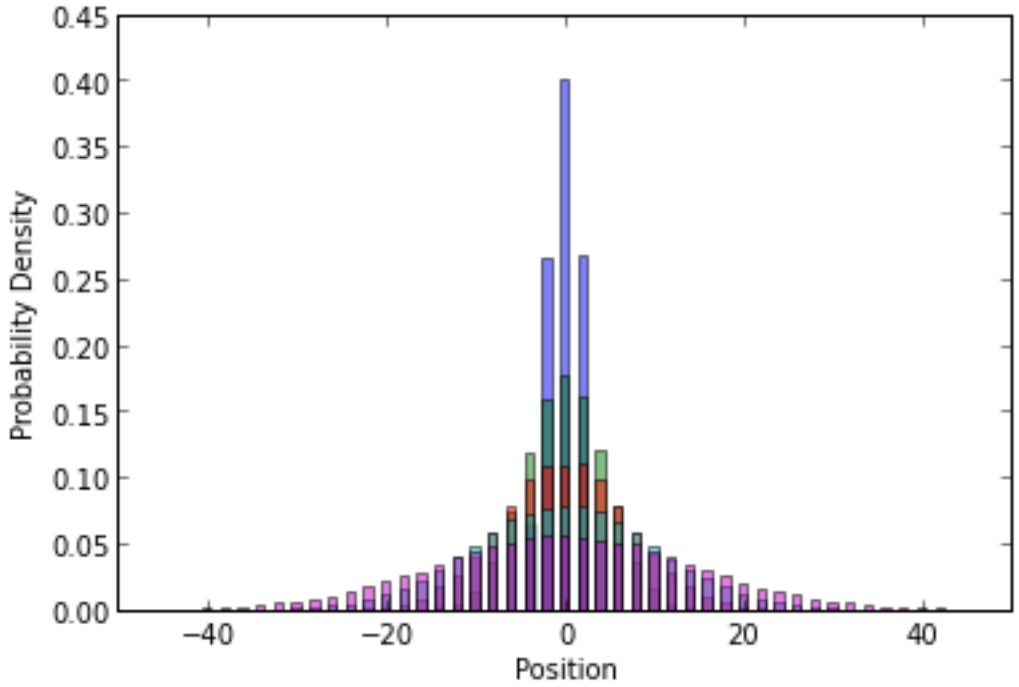
```
In [92]: times = [4, 20, 50, 100, 200]

         positions = [[position(w, t) for t in times] for w in walks]
```

```
In [93]: positions = np.array(positions)
         np.shape(positions)
```

```
Out[93]: (50000, 5)
```

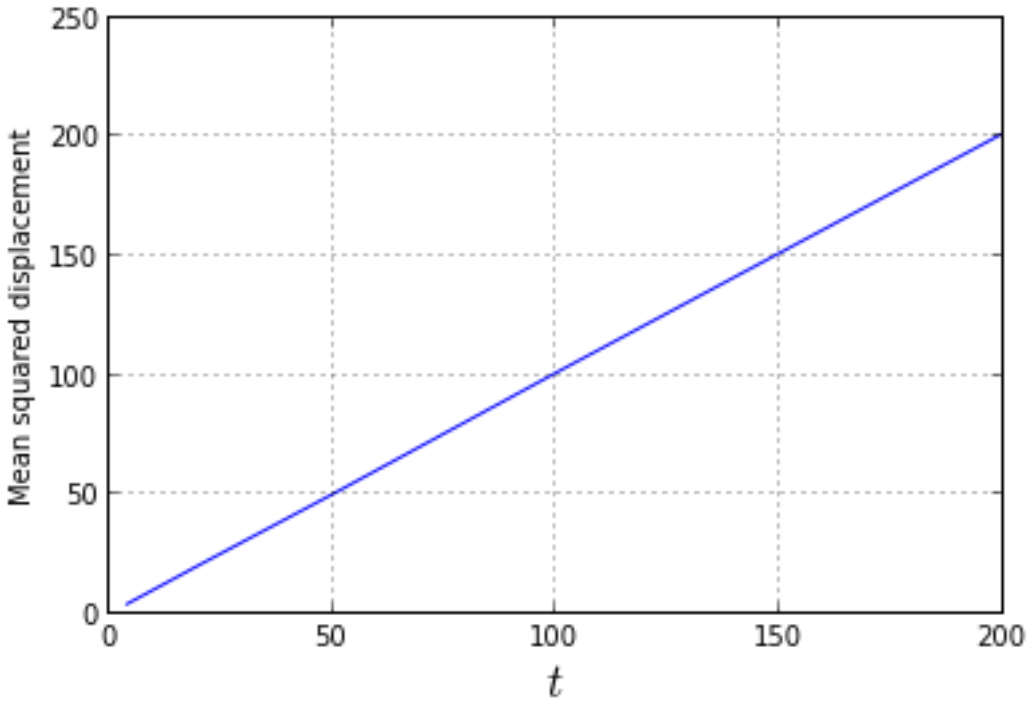
In [96]:
```
for i in range(len(positions[0])):
    plt.hist(positions[:,i], bins = np.arange(-max(np.abs(positions[:,i]))-0.5,max(np.abs(posi
plt.xlim([-50, 50])
plt.xlabel("Position")
plt.ylabel("Probability Density")
```

Out[96]: <matplotlib.text.Text at 0x7f72d5923710>



In [57]:
```
second_moment = np.array([sum(positions[:,t]**2.0 / 50000.0) for t in range(len(times))])
```

In [58]:
```
plt.plot(times, second_moment)
plt.xlabel("$t$", fontsize=17)
plt.ylabel("Mean squared displacement")
plt.grid(True)
```



As we can see, the second moment varies linearly with time and therefore the process is diffusive.

Radom walk in bounded domain In order to bound the domain, we need to change the function `position` to take into account that when the walker reaches the boundary, it cannot go on. We will do this by changing the way the simulation is done. Instead of calculating a matrix of random numbers we will use loops.

```
In [59]: def walk(x_0, lower_boundary, upper_boundary, t):

             positions = [x_0]
             pos = x_0

             for i in range(t):
                 choose = rand.rand()

                 if (pos > lower_boundary) and (pos < upper_boundary):
                     if choose <= 0.5:
                         pos -= 1
                     else:
                         pos += 1

                 elif pos == lower_boundary:
                     if choose > 0.5:
                         pos += 1
                 elif pos == upper_boundary:
                     if choose <= 0.5:
                         pos -=1
```

```
            positions.append(pos)

        return np.array(positions)
```

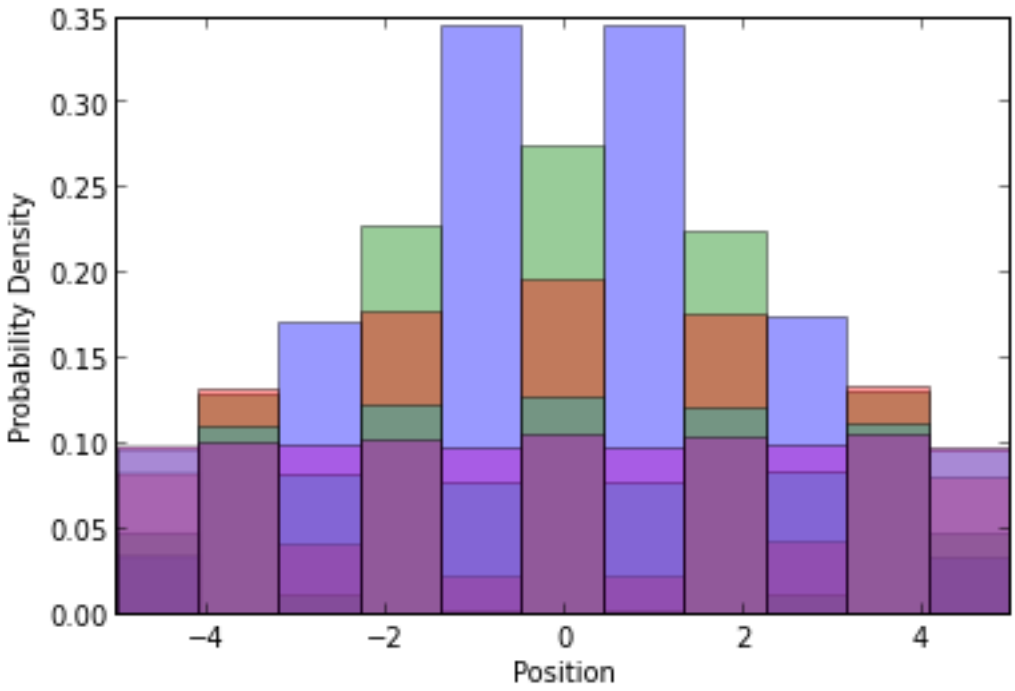
In [60]:
```
pos = walk(0, -5, 5, 20)
pos
```

Out[60]:
```
array([ 0, -1,  0,  1,  0, -1, -2, -1,  0,  1,  2,  3,  4,  5,  5,  5,  5,
        5,  4,  5,  4])
```

In [61]:
```
positions = np.array([walk(0,-5,5,1000) for i in range(50000)])
```

In [62]:
```
for i in [5, 10, 20, 50, 100]:
    plt.hist(positions[:,i], bins=11, normed=True, alpha=0.4)

plt.xlim([-5, 5])
plt.xlabel("Position")
plt.ylabel("Probability Density")
```

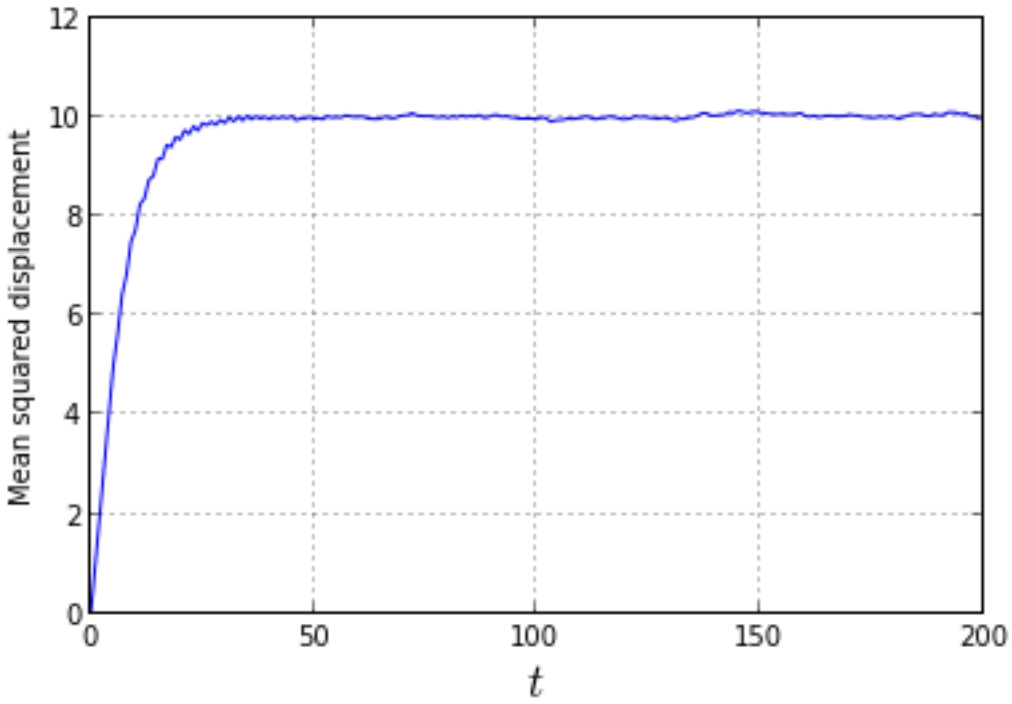
Out[62]:
```
<matplotlib.text.Text at 0x7f72fd7d5dd0>
```



In [63]:
```
times = range(200)
```

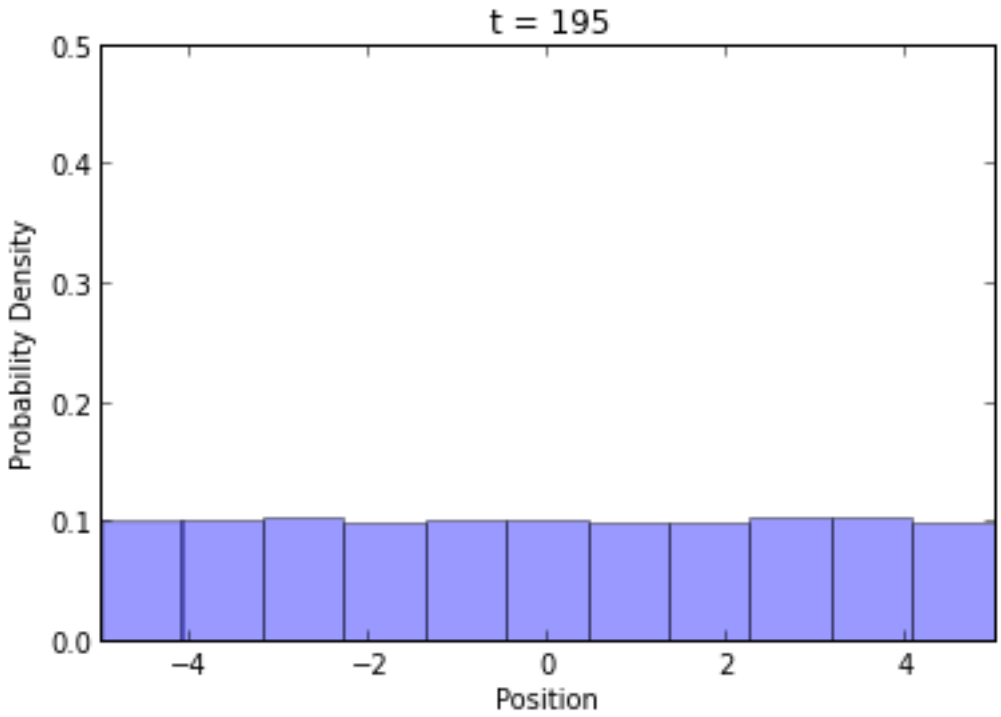
In [64]:
```
second_moment = np.array([sum(positions[:,t]**2.0 / 50000.0) for t in times])
```

In [65]:
```
plt.plot(times, second_moment)
plt.xlabel("$t$", fontsize=17)
plt.ylabel("Mean squared displacement")
plt.grid(True)
```



We can see that there is diffusion during the initial steps as the curve appears to be linear for small times, but as time goes on, the mean squared displacement stops growing. This happens since the domain is bounded and therefore as the distribution reaches a stationary state (uniform), the mean squred displacement becomes a constant.

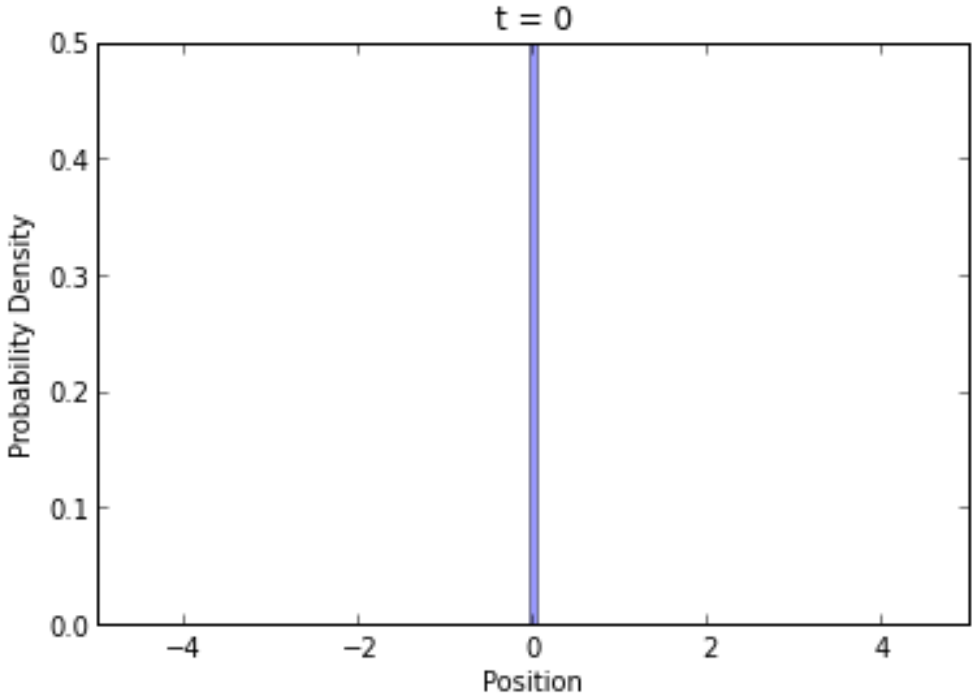
```
In [66]: plt.xlim([-5, 5])
         plt.xlabel("Position")
         plt.ylabel("Probability Density")

         for i in range(0, 200, 5):
             plt.clf()
             plt.hist(positions[:,i], bins=11, normed=True, alpha=0.4)

             plt.xlim([-5, 5])
             plt.ylim([0, 0.5])
             plt.xlabel("Position")
             plt.ylabel("Probability Density")
             plt.title("t = {}".format(i))

             plt.savefig("hist_walker_{:=03}.png".format(i))
```




```
In [67]: def simple_anim():
             for i in range(0, 200, 5):
                 image_name = "hist_walker_{:=03}.png".format(i)
                 print image_name
                 yield image_name

         it = iter(simple_anim())

In [68]: try:
             image_name = it.next()
         except StopIteration:
             it = iter(simple_anim())
             image_name = it.next()

         Image(image_name )
```

```
hist_walker_000.png
```

Out[68]:



1.2 The diffusion equation

The discretized version of the diffusion equation is:

$$\frac{P(x,t+\Delta t)-P(x,t)}{\Delta t}=\frac{D}{(\Delta x)^2}\left\{P(x+\Delta x,t)-2P(x,t)+P(x-\Delta x,t)\right\}$$

.

Therefore, we have that at time $t+\Delta t$ the solution is:

$$P(x,t)=\frac{D\Delta t}{(\Delta x)^2}\left\{P(x+\Delta x,t)-2P(x,t)+P(x-\Delta x,t)\right\}+P(x,t)$$

.

We can express this equation in matrix form:

$$P(x,t+\Delta t)=\left(\frac{D\Delta t}{(\Delta x)^2}\mathbb{A}+\mathbb{I}\right)\cdot P(x,t)$$

, where $\mathbb{A}$ is the same matrix similar to the one for the gain-loss equation and $\mathbb{I}$ is the identity. Here we are viewing P as a vector whose elements are the values of the the distribution $P(x,t)$ at the discretized points representing the domain of x.

We can also think about it as:

$$\mathbf{P}(t+\Delta t)=\mathbf{P}(t)+\frac{D\Delta t}{(\Delta x)^2}\mathbb{A}\cdot\mathbf{P}(t)$$

,

where we have removed the x from the notation and made the $\mathbf{P}$ bold to emphasize that it is a "vector". To satisfy the Courant-Freidrichs-Lewy condition, we must choose time and spatial steps, Δt and Δx, respectively such that:

$$\frac{2D\Delta t}{(\Delta x)^2}<1$$

.

For instance, we can take $\Delta t = 0.001$, $\Delta x = 0.1$ for D = 1.

```
In [69]: dt = 0.001
         dx = 0.1
         D = 1

         2*D*dt/dx**2.0
         D*dt/dx**2.0
```

```
Out[69]: 0.09999999999999998
```

Which is less than 1, so we can guarantee the convergence of the solution.

```
In [70]: x = np.arange(-100, 100, dx)
         t = np.arange(0,100, dt)

         def matrix(dx, dt, x_min, x_max):

             length = len(np.arange(x_min, x_max, dx))

             A = -2*np.eye(length)

             #boundary conditions:
             A[0,0] = -1
             A[0,1] = 1

             A[-1,-1] = -1
             A[-1, -2] = 1

             for i in range(1,np.shape(A)[0]-1):
                 A[i,i-1] = 1
                 A[i,i+1] = 1

             A = (D*dt/dx**2.0)*A

             return A

         A = matrix(dx, dt, -100, 100)
         A
```

```
Out[70]: array([[-0.1,  0.1, -0. , ..., -0. , -0. , -0. ],
                [ 0.1, -0.2,  0.1, ..., -0. , -0. , -0. ],
                [-0. ,  0.1, -0.2, ..., -0. , -0. , -0. ],
                ...,
                [-0. , -0. , -0. , ..., -0.2,  0.1, -0. ],
                [-0. , -0. , -0. , ...,  0.1, -0.2,  0.1],
                [-0. , -0. , -0. , ..., -0. ,  0.1, -0.1]])
```

```
In [71]: t = np.arange(0,100, dt)
```

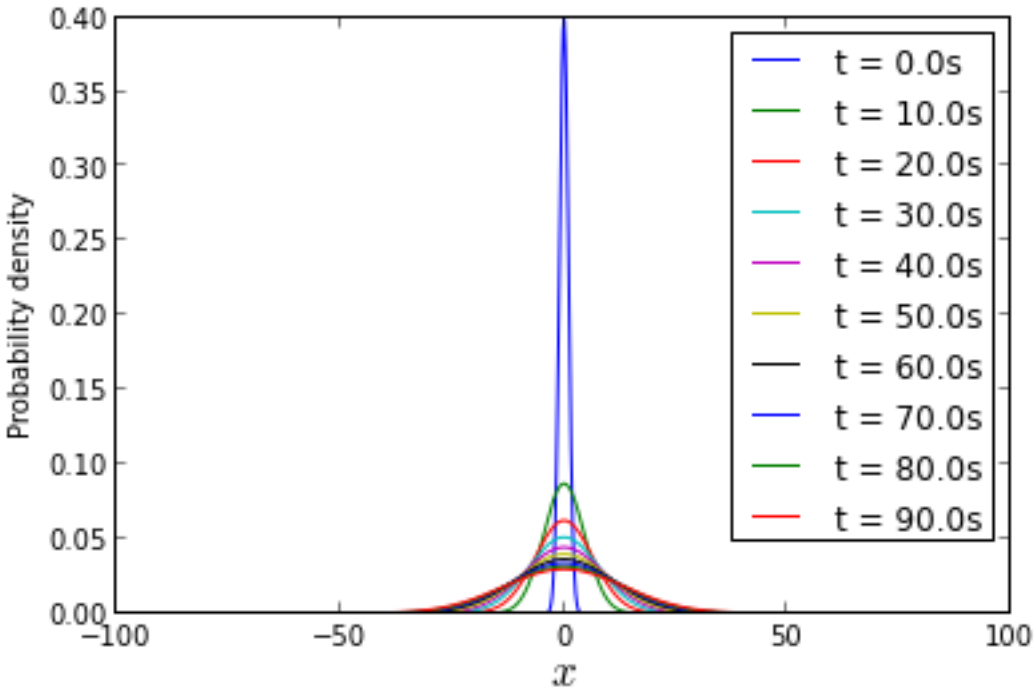
```
In [73]: def integrate_with_matrix(p_0, dx, dt, x_min = -100, x_max = 100, t_ini = 0, t_max = 100):

             x = np.arange(x_min, x_max, dx)
             t = np.arange(t_ini, t_max, dt)
```

```
            A = matrix(dx, dt, x_min, x_max)

            P = []

            p_old = p_0.copy()

            for i in range(len(t)):

                p_new = p_old + np.dot(A,p_old)
                p_old = p_new.copy()

                P.append(p_old)

            return x, t, np.array(P)


        p_0 = np.exp(-x**2/2.0)/np.sqrt(2*np.pi)

        %time x, t, P = integrate_with_matrix(p_0, dx, dt)
```

```
CPU times: user 7min 27s, sys: 1.94 s, total: 7min 29s
Wall time: 7min 29s
```

```
In [74]: for i in range(len(t)):
             if i%10000 == 0:

                 plt.plot(x, P[i],  label = "t = {}s".format(i*dt))


         plt.xlabel("$x$", fontsize=17)
         plt.ylabel("Probability density")
         plt.legend()
```

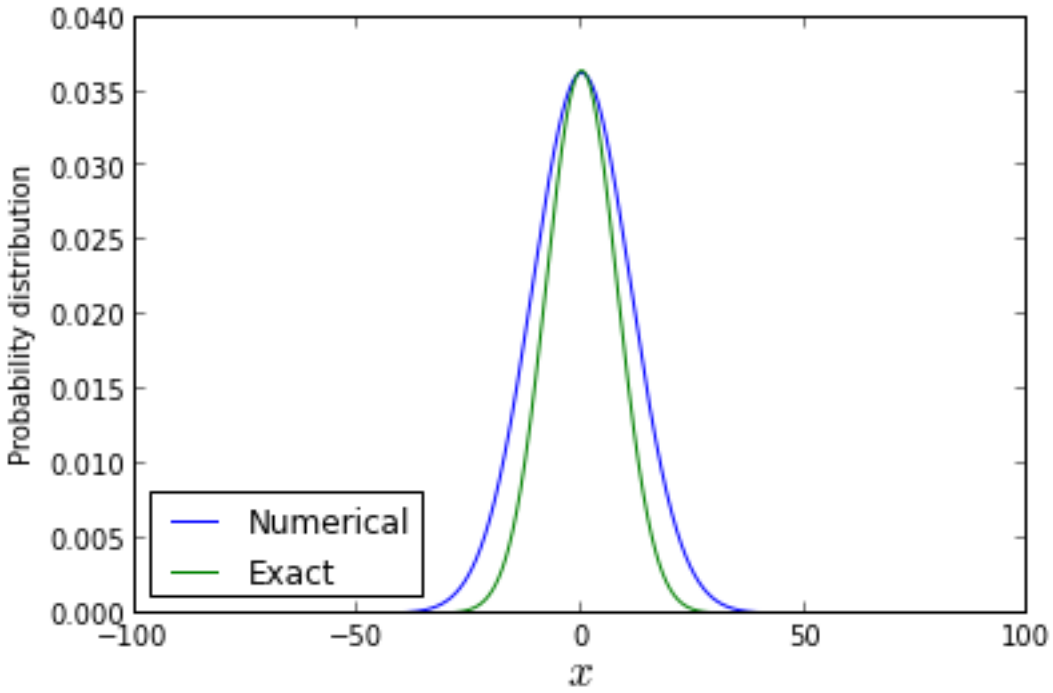
```
Out[74]: <matplotlib.legend.Legend at 0x7f72fe199810>
```




```
In [75]: def p_exact(x, t):

             return np.exp(-x**2/(2*np.pi + 2*D*t)) / (np.sqrt(4*np.pi*D*t))
```

We are going to compare the numerical solution to the exact solution at $t = 60s$.

```
In [76]: tt = 60

         plt.plot(x, P[int(tt/dt)+1], label="Numerical")
         plt.plot(x, p_exact(x, tt), label="Exact")
         plt.legend(loc=3)

         plt.xlabel("$x$", fontsize = 17)
         plt.ylabel("Probability distribution")

Out[76]: <matplotlib.text.Text at 0x7f72fe193850>
```



We can look at what happens with the absolute and relative error of the numerical approximation.

```
In [77]: absolute_errors = []
         relative_errors = []

         for tt in range(100):
             abs_error = np.abs(p_exact(x,tt) - P[int(tt/dt)])
             rel_error = abs_error / p_exact(x,tt)

             absolute_errors.append(abs_error)
             relative_errors.append(rel_error)

         absolute_errors=np.array(absolute_errors)
         relative_errors=np.array(relative_errors)
```

```
-c:3: RuntimeWarning: divide by zero encountered in divide
-c:3: RuntimeWarning: invalid value encountered in divide
-c:6: RuntimeWarning: invalid value encountered in divide
-c:6: RuntimeWarning: divide by zero encountered in divide
```

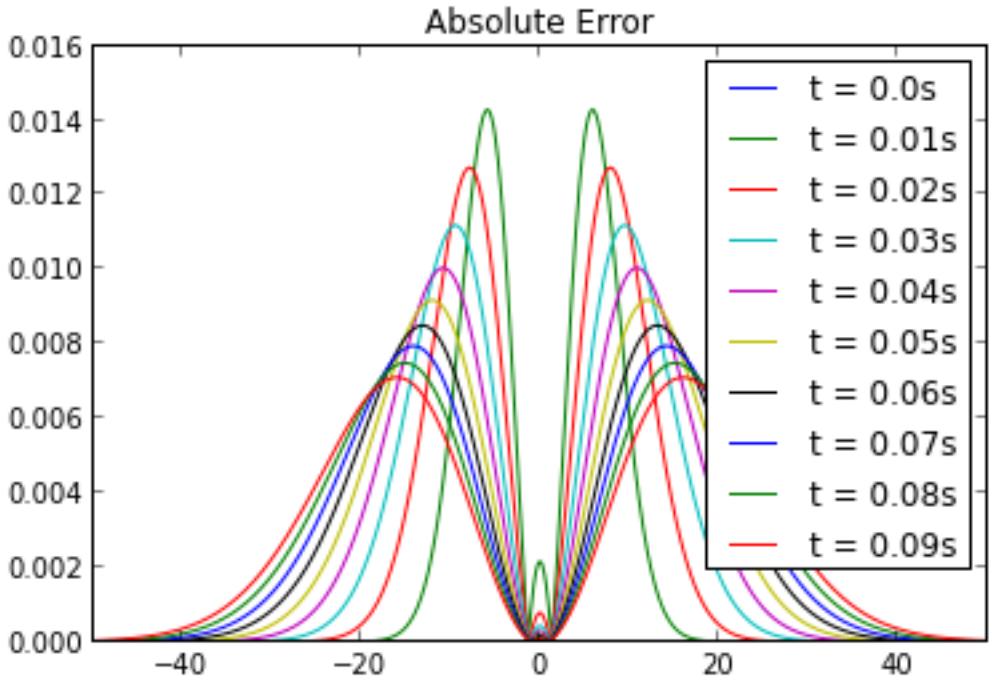
```
In [78]: len(absolute_errors)
```

```
Out[78]: 100
```

```
In [79]: for i in range(100):
             if i%10 == 0:

                 plt.plot(x, absolute_errors[i],  label = "t = {}s".format(i*dt))

         plt.title("Absolute Error")
```

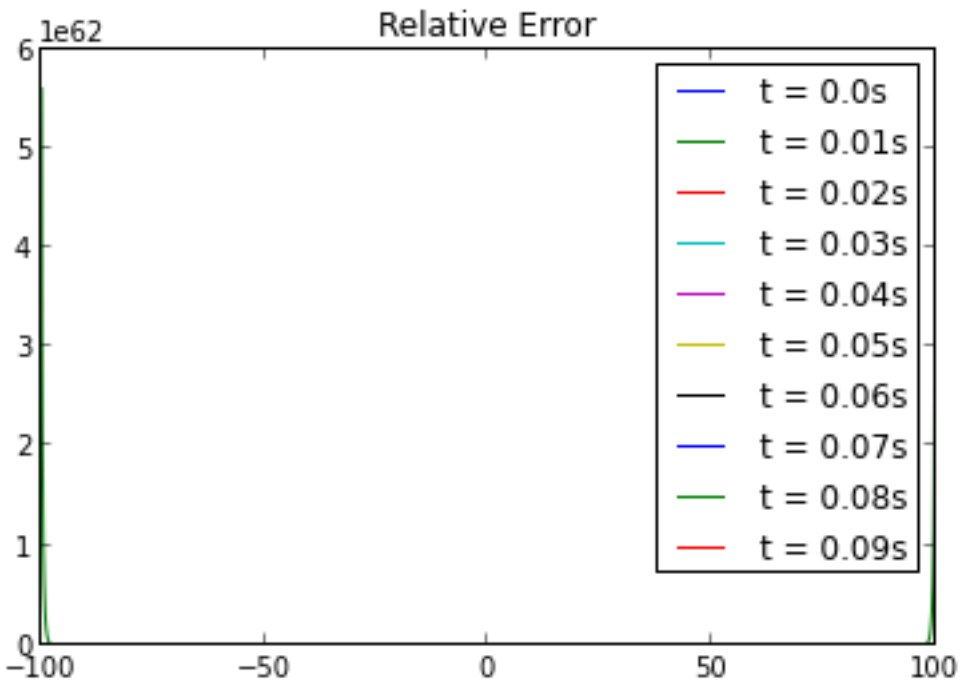
```
plt.xlim([-50,50])
plt.legend()
```

Out[79]: <matplotlib.legend.Legend at 0x7f72fe199650>



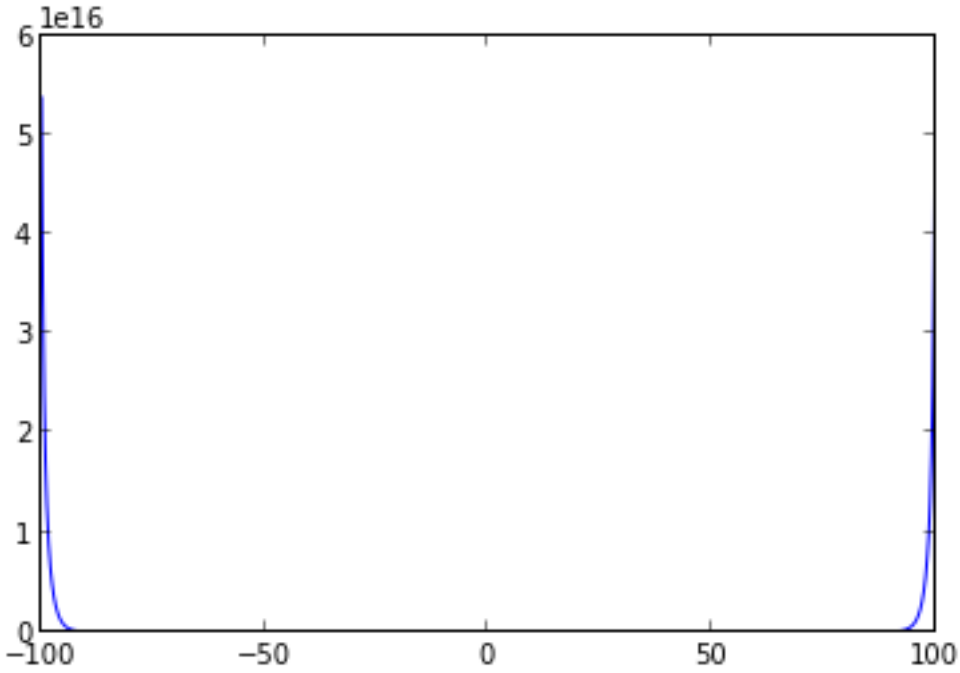
In [80]:
```
for i in range(100):
    if i%10 == 0:

        plt.plot(x, relative_errors[i],  label = "t = {}s".format(i*dt))

plt.title("Relative Error")
plt.legend()
```

Out[80]: <matplotlib.legend.Legend at 0x7f72fde99a10>



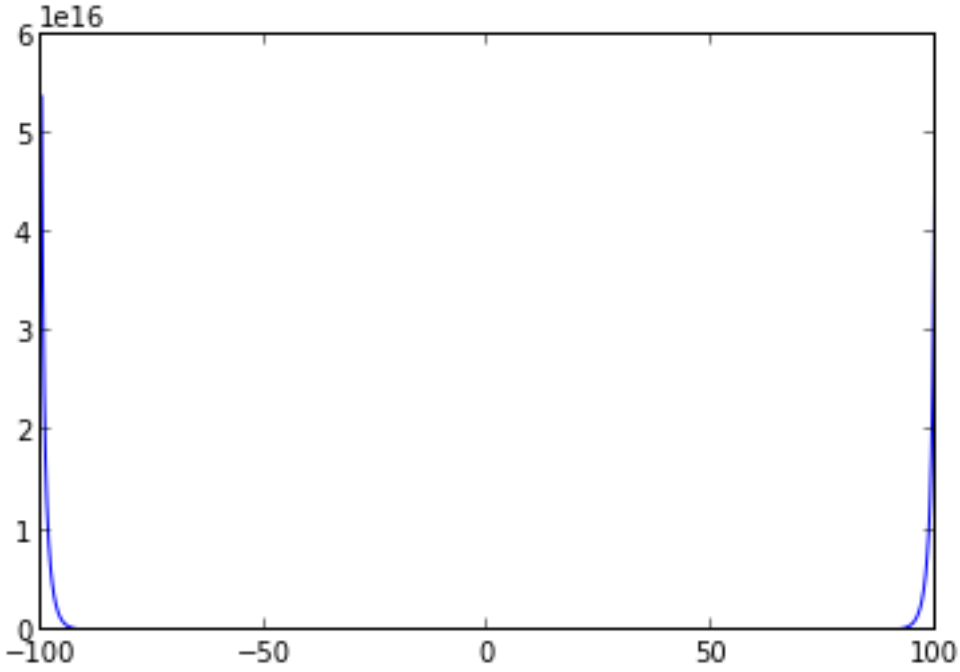
In [81]: `plt.plot(x, relative_errors[60])`

Out[81]: `[<matplotlib.lines.Line2D at 0x7f72fe1810d0>]`



```
In [82]: plt.plot(x, relative_errors[60])
```

```
Out[82]: [<matplotlib.lines.Line2D at 0x7f72feb94090>]
```



As we can see,

We will now look at what happens to the instablilities as the value of $\frac{2D\Delta t}{(\Delta x)^2}$ grows.

We will keep $\Delta x = 0.1$ and we will vary Δt .

```
In [102]: dx = 0.1
          D = 1.0

          x = np.arange(-100, 100, dx)


          p_0 = np.exp(-x**2/2.0)/np.sqrt(2*np.pi)



          T_instability = []
          P_instability = []

          for dt in np.logspace(0.001, 0.01, 10):

              print "dt = ", dt

              t = np.arange(0,20, dt)
              T_instability.append(t)

              length = len(x)
```

```
            A = -2*np.eye(length)

                #boundary conditions:
            A[0,0] = -1
            A[0,1] = 1

            A[-1,-1] = -1
            A[-1, -2] = 1

            for i in range(1,np.shape(A)[0]-1):
                A[i,i-1] = 1
                A[i,i+1] = 1

            A = (D*dt/dx**2.0)*A


            P = []
            p_0 = np.exp(-x**2/2.0)/np.sqrt(2*np.pi)

            p_old = p_0.copy()

            for i in range(len(t)):

                p_new = p_old + np.dot(A,p_old)
                p_old = p_new.copy()

                P.append(p_old)

            P_instability.append(P[-1])
```

```
dt =  1.00230523808
dt =  1.00461579028
dt =  1.00693166885
dt =  1.00925288608
dt =  1.01157945426
dt =  1.01391138574
dt =  1.01624869287
dt =  1.01859138805
dt =  1.02093948371
dt =  1.02329299228
```

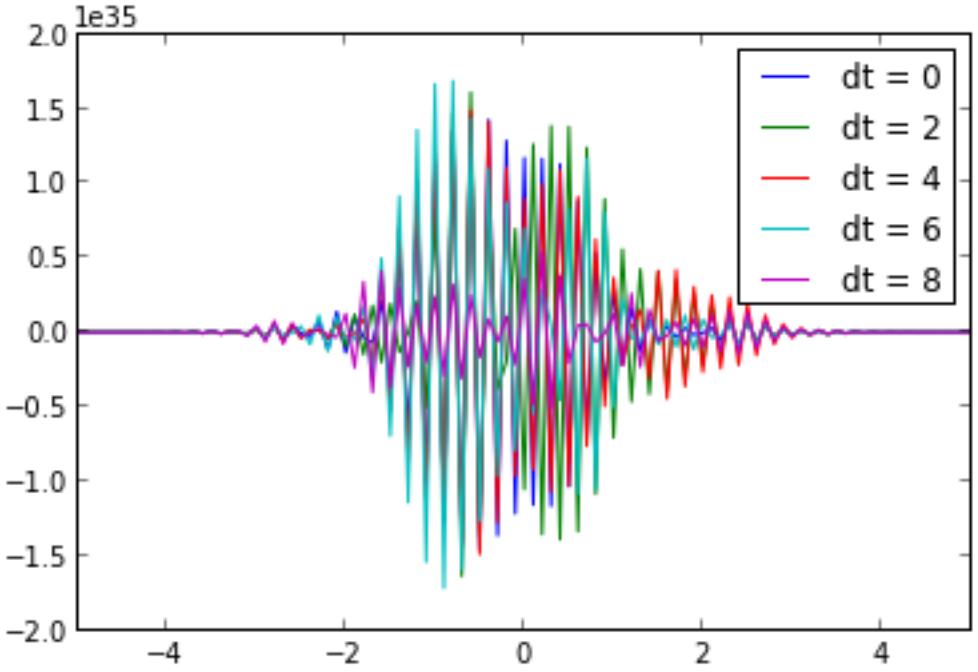
```
In [103]: P_instability
```

```
Out[103]: [array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.]),
           array([ 0.,  0.,  0., ...,  0.,  0.,  0.])]
```

```
In [112]: for i in range(len(P_instability)):
              if i%2==0:
```

```
            plt.plot(x, P_instability[i], label ="dt = {}".format(int(t[i])))

        plt.xlim([-5, 5])
        plt.legend()
```

```
Out[112]: <matplotlib.legend.Legend at 0x7f725f5b4e50>
```



We can see how, when we vary Δt, while keeping Δx constant, the solutions take wave forms that explode. These correspond to the Fourier modes that are unstable when the Courant-Freidrichs-Lewy condition is not satisfied.