(Custom CSS files are not reliable for controlling Jupyter font style. To establish the same appearance as the original notebook, depend on the browser to control the font, by setting the desired font faces in the browser settings. For example, Chrome 135 or Firefox 134 can do this. In this notebook series, Bookerly font is for markdown and Monaco is for code.)

Chapter 31-13: Parabolic PDEs Using the Monte Carlo Method.

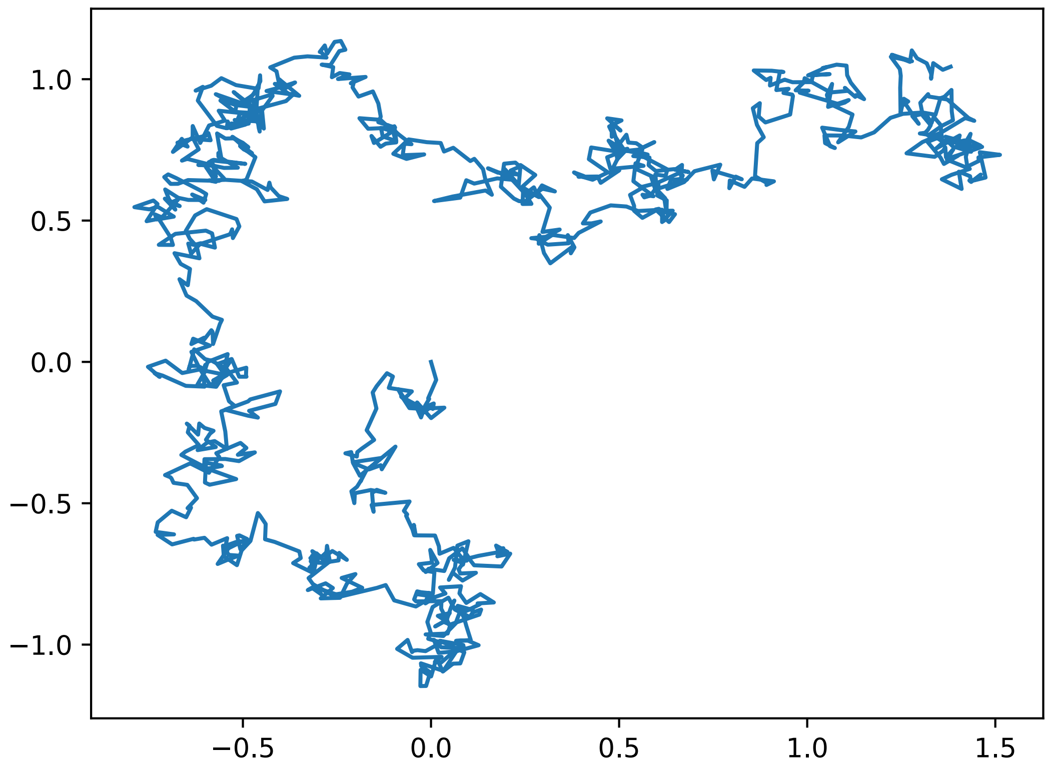
Monte Carlo Methods are a class of computational solutions that can be applied to various problems which rely on repeated random sampling to provide generally approximate solutions. Monte Carlo is a stochastic approach, in which a series of simulations (trials), representing the analyzed problem, with randomly selected input values, are performed. Among these trials, a specified number of properly defined successes is achieved. The ratio between the number of success trials to the number of all trials, scaled by dimensional quantity (e.g., area or function value) allows for the estimation of the unknown solution, providing the number of trials is large enough.

At the address: https://srome.github.io/On-Solving-Partial-Differential-Equations-with-Brownian-Motion-in-Python/ (https://srome.github.io/On-Solving-Partial-Differential-Equations-with-Brownian-Motion-in-Python/) there is an interesting blog about doing pde solving with Brownian motion approximation. However, BM is not exactly Monte Carlo, is it? Actually, the two stochastic approaches are sometimes covered in the same place, for example: http://www.columbia.edu/~mh2078/MonteCarlo/MCS_Generate_RVars.pdf (http://www.columbia.edu/~mh2078/MonteCarlo/MCS_Generate_RVars.pdf) . So it will be assumed here that using BM is essentially the same as using MC. And besides, it took 7.5 hrs for a 13700K processor to spit out the graphics for it, so they are not about to be wasted.

Perhaps an opportunity is here to reaffirm that the Laplace equation is in the parabolic category.

In [7]:
```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

t=1
n = 1000
delta_t = np.sqrt(t/n)

# Create each dW step
dW1 = delta_t * np.random.normal(0,1,size=n)
dW2 = delta_t * np.random.normal(0,1,size=n)
W = np.zeros((n+1,2))

# Add W_{j-1} + dW_{j-1}
W[1:,0] = np.cumsum(dW1)
W[1:,1] = np.cumsum(dW2)

plt.plot(W[:,0],W[:,1], '-')
```

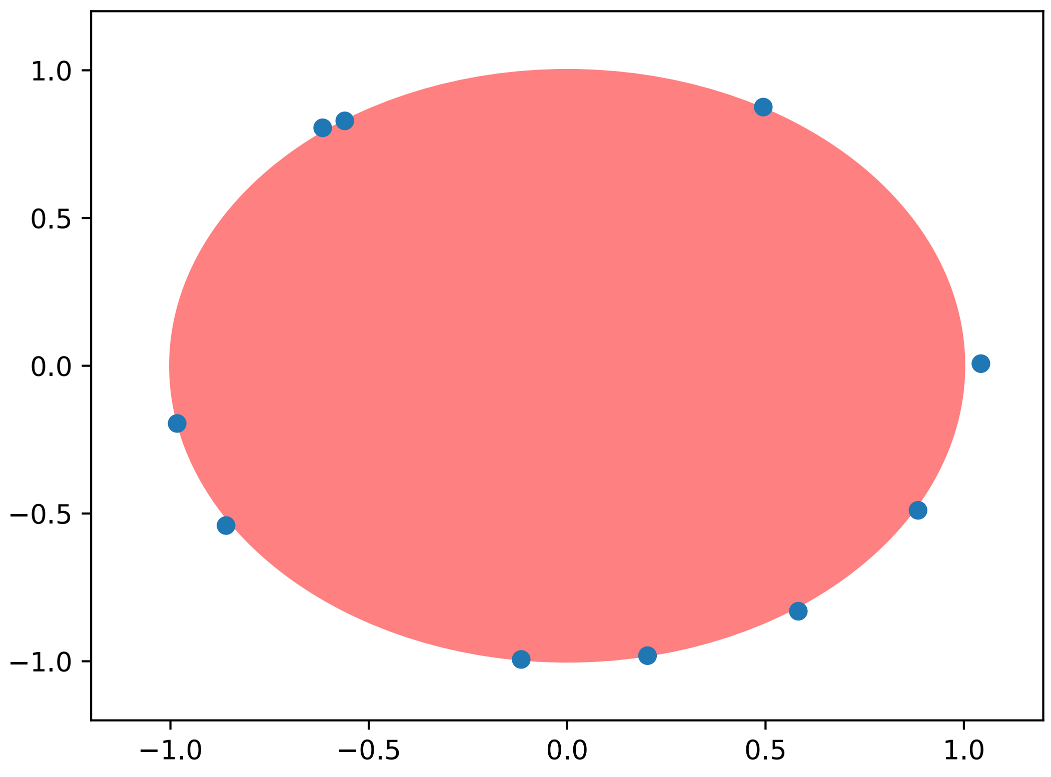
Out[7]:
```
[<matplotlib.lines.Line2D at 0x2a2a88dea90>]
```



Above: a random Brownian walk, which has the same appearance as a Monte Carlo generated walk.

In [8]:
```
def check_if_exit(v):
    # Takes a vector v=(x,y)
    # Checks if v has intersected with the boundary of D
    if np.linalg.norm(v,2) >=1:
        return True
    return False

def simulate_exit_time(v):
    # Simulates exit time starting at v=(x,y), returns exit position
    delta_t = np.sqrt(.001)
    exit = False
    x = v.copy()
    while not exit:
        x = x + delta_t * np.random.normal(0,1,size=2)
        exit = check_if_exit(x)
    return x

v=np.array((0,0)) # The origin
exit_times = np.array([simulate_exit_time(v) for k in range(0,10) ])


circle1=plt.Circle((0,0),1,color='r', alpha=.5)
plt.gcf().gca().add_artist(circle1)
plt.axis([-1.2, 1.2, -1.2, 1.2])
plt.scatter(exit_times[:,0],exit_times[:,1])
```

Out[8]:
```
<matplotlib.collections.PathCollection at 0x2a2a8a943a0>
```



Above: simulated exit points on an ellipse, fairly similar to the epsilon criterion for deciding when a random step reaches the domain boundary in the WOS (walk on spheres) version of popular pde solvers.

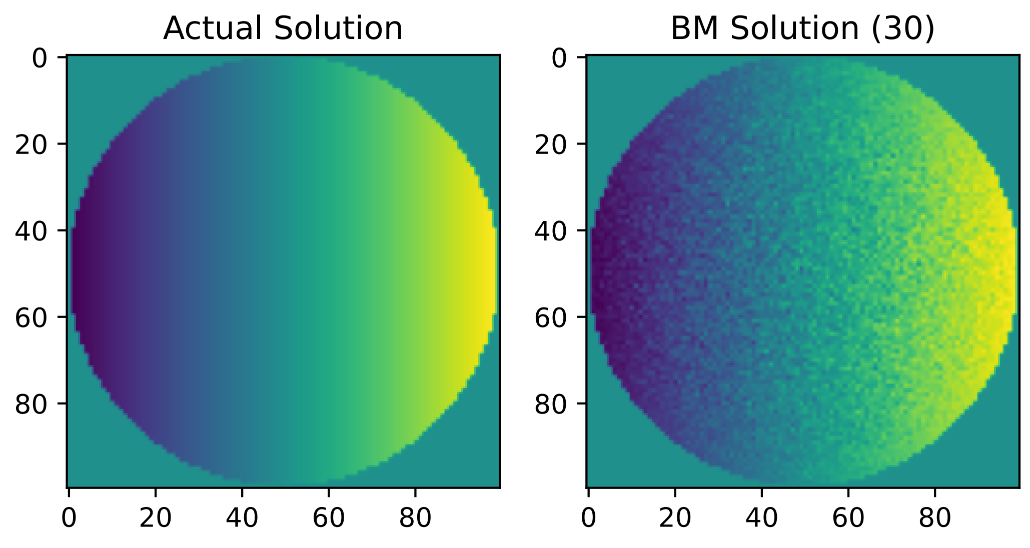
In [4]:
```
np.random.seed(8) #Side Infinity

def check_if_exit(v):
    # Takes a vector v=(x,y)
    # Checks if v has intersected with the boundary of D
    if np.linalg.norm(v,2) >=1:
        return True
    return False

def simulate_exit_time(v):
    # Simulates exit time starting at v=(x,y), returns exit position
    delta_t = np.sqrt(.001)
    exit = False

    # Copy because simulation modifies in place
    if hasattr(v,'copy'): # For NumPy arrays
        x = v.copy()
    else:
        x = np.array(v) # We input a non-NumPy array
    while not exit:
        x += delta_t * np.random.normal(0,1,size=2) # += modifies in place
        exit = check_if_exit(x)
    return x

v=np.array((.5,.5)) # The origin
u = lambda x : np.linalg.norm(x,2)*np.cos(np.arctan2(x[1],x[0]))
f = lambda x : np.cos(np.arctan2(x[1],x[0]))

def get_exp_f_exit(starting_point, n_trials):
    return np.mean([f(simulate_exit_time(starting_point)) for k in range(0,n_trials)])

exp_f_exit = get_exp_f_exit(v,2000) # Expected value of f(Exit(x,d))
print('The value u(v) = %s\nThe value of Exp(f(Exit))=%s' %(u(v), exp_f_exit))
```

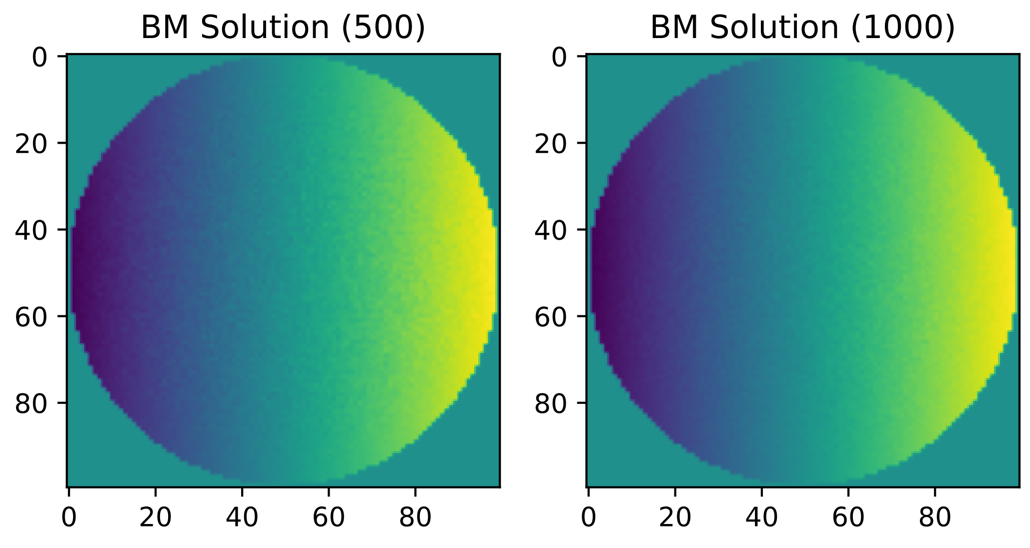
```
The value u(v) = 0.5000000000000001
The value of Exp(f(Exit))=0.4975601145395467
```

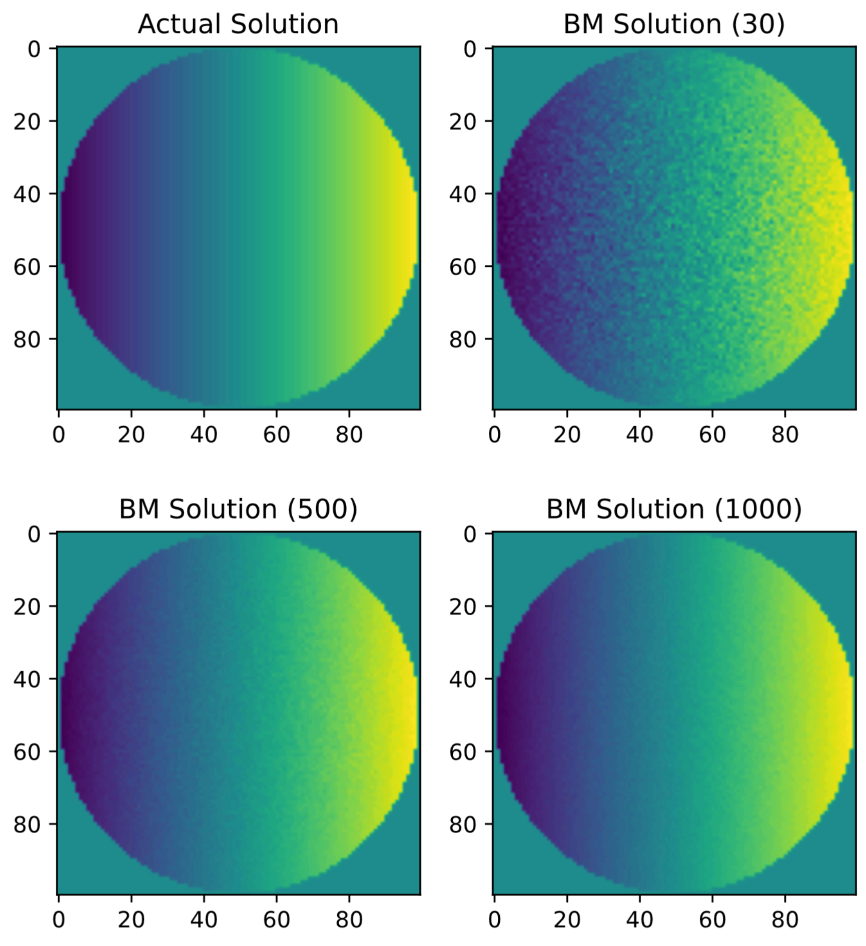
In [5]:
```
lin = np.linspace(-1, 1, 100)
x, y = np.meshgrid(lin, lin)
print(x.shape)
u_vec = np.zeros(x.shape)
bm_vec_30 = np.zeros(x.shape)
bm_vec_500 = np.zeros(x.shape)
bm_vec_1000 = np.zeros(x.shape)

# Convert u to a solution in x,y coordinates
u_x = lambda x,y : np.linalg.norm(np.array([x,y]),2)*np.cos(np.arctan2(y,x))

# Calculate actual and approximate solution for (x,y) in D
for k in range(0,x.shape[0]):
    for j in range(0,x.shape[1]):
        x_t = x[k,j]
        y_t = y[k,j]

        # If the point is outside the circle, the solution is undefined
        if np.sqrt((x_t)**2 + (y_t)**2) > 1:
            continue

        # Calculate function value at this point for each image
        u_vec[k,j] = u_x(x_t,y_t)
        bm_vec_30[k,j] = get_exp_f_exit((x_t,y_t),30)
        bm_vec_500[k,j] =  get_exp_f_exit((x_t,y_t),500)
        bm_vec_1000[k,j] =  get_exp_f_exit((x_t,y_t),1000)

fig = plt.figure()
ax = fig.add_subplot(121)
plt.imshow(u_vec)
plt.title('Actual Solution')

ax = fig.add_subplot(122)
plt.title('BM Solution (30)')
plt.imshow(bm_vec_30)

fig = plt.figure()

ax = fig.add_subplot(121)
plt.title('BM Solution (500)')
plt.imshow(bm_vec_500)

ax = fig.add_subplot(122)
plt.title('BM Solution (1000)')
plt.imshow(bm_vec_1000)
```

```
(100, 100)
```

Out[5]:
```
<matplotlib.image.AxesImage at 0x2a2a852cc10>
```







Above: Brownian motion solutions, rather similar to the walk on spheres output for a parabolic pde performed with Monte Carlo.

The long processing time practically guarantees that Github will never show this result, so it is shown here in static form.

```
In [ ]:
```