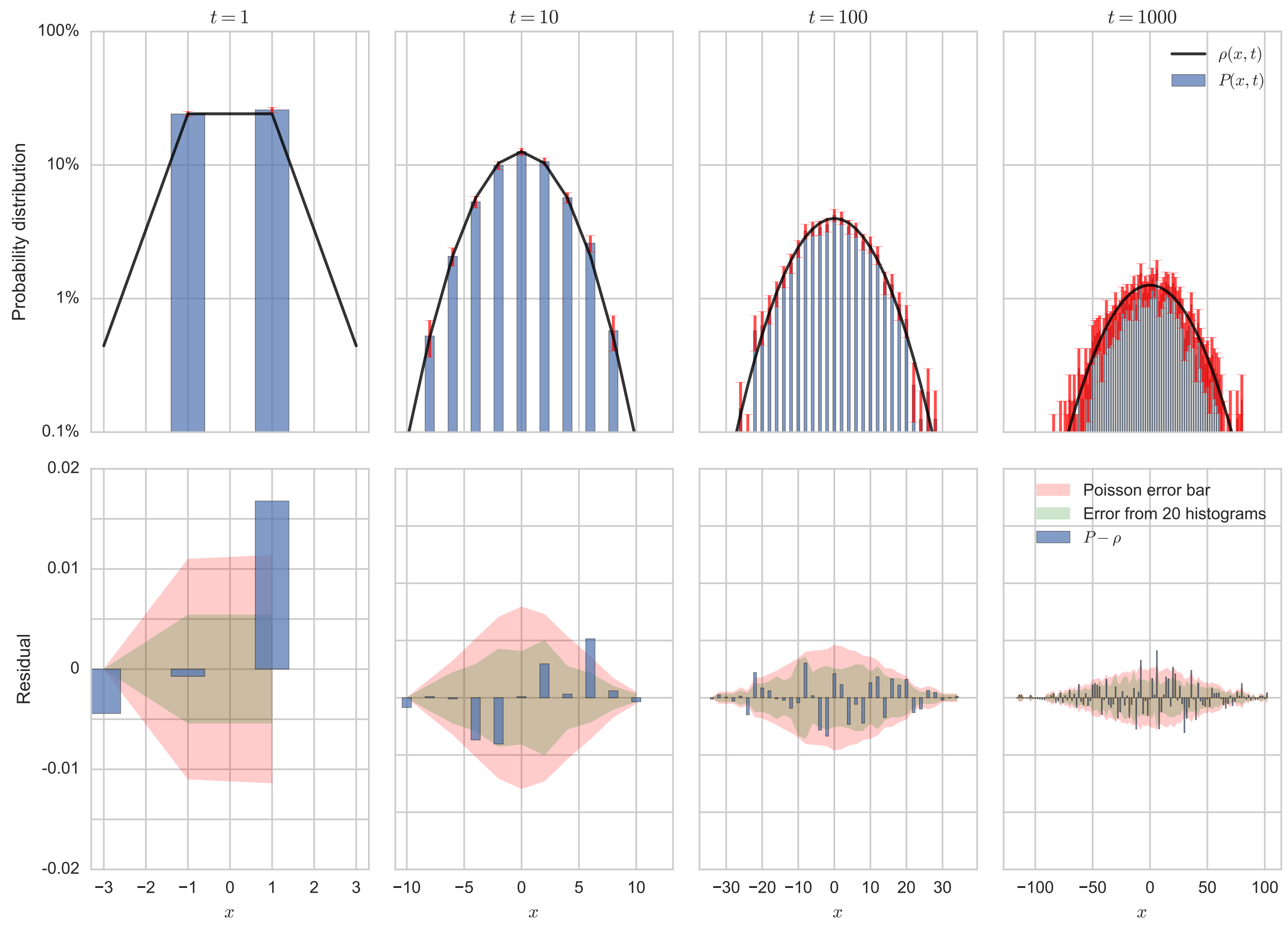
1 Random Walk

```python
import numpy as np
import random as ran
import matplotlib.pylab as plt
import seaborn as sns # Makes plots look nicer
import time

def step():
    # Discrete, unbiased step in 1 dimension
    return ran.choice([-1, 1])

def walk(t_sample):
    # Given sample time points, performs a discrete 1d random walk and returns
    #  net displacement x at those times/steps
    t_max = np.max(t_sample)

    x = 0
    x_sample = []
    for t in range(1, t_max+1):
        x += step()

        if t in t_sample:
            x_sample.append(x)

    return x_sample

def sample(N, t_sample):
    # Performs N random walks, sampling at the steps given by t_sample

    # This array will hold all measurements, with time separated by columns and
    #  each row corresponding to a single random walk
    samples = np.empty([N, len(t_sample)])

    for n in range(N):
        x_sample = np.array(walk(t_sample))

        samples[n,:] = x_sample

    return samples

def plot_dist():
    sns.set(style='whitegrid') # Comment out if not using seaborn
    N = 2000
    t_sample = [1, 10, 100, 1000]

    # Probability distributions on top row, residuals on bottom row,
    #  and a column per time/step sample point
    fig, axes = plt.subplots(ncols=len(t_sample), nrows=2, sharex='col',
                             sharey='row', figsize=(11,8))

    samples = sample(N, t_sample)

    # The diffusion coefficient with step size and time step 1
    D = 1 / 2
```

```python
# The gaussian describing the diffusion equation density
p_diffusion = lambda x,t: np.exp(-x**2/(4*D*t)) / np.sqrt(4*np.pi*D*t)

for i,t in enumerate(t_sample):
    x = samples[:,i]
    bins = np.arange(np.min(x)-2, np.max(x)+4, 2)

    p_meas,_ = np.histogram(x, bins=bins, normed=True)

    error_bars = np.sqrt(p_meas / N)
    axes[0,i].bar(bins[:-1], p_meas, alpha=0.7, align='center',
                  yerr=error_bars, label='$P(x,t)$',
                  error_kw={'ecolor': 'r', 'alpha': 0.7})

    p_theory = p_diffusion(bins, t)
    axes[0,i].plot(bins, p_theory, label=r'$\rho (x,t)$', alpha=0.8,
                   color='k')

    # p_theory has one extra data point due to binning
    resid = p_meas - p_theory[:-1]
    axes[1,i].bar(bins[:-1], resid, alpha=0.7, align='center',
                  label=r'$P - \rho$')
    axes[1,i].fill_between(bins[:-1], y1=error_bars, y2=-error_bars,
                           color='r', alpha=0.2,
                           label='Poisson error bar')

    # An alternative way to get error bars is to measure the variance
    #  in >= ~20 smaller histograms
    hist = np.empty([20, len(bins)-1])
    # Find the indicies at which to cut the data into 20 histograms
    indices = np.linspace(0, N, 21).astype(int)
    for j in range(len(indices)-1):
        # Subset the data to get a new histogram
        sub_x = x[indices[j]:indices[j+1]]
        p_meas,_ = np.histogram(sub_x, bins=bins, normed=True)
        hist[j,:] = p_meas

    # Now get the standard deviation per column/bin to find the error bars
    #  in the histogram. The axis=0 argument means perform the np.std()
    #  calculation per column
    error_bars = np.std(hist, axis=0) / np.sqrt(20)
    axes[1,i].fill_between(bins[:-1], y1=error_bars, y2=-error_bars,
                           color='g', alpha=0.2,
                           label='Error from 20 histograms')

    axes[0,i].set_title(r'$t = %d$' % t)
    axes[1,i].set_xlim(1.1*bins[0], 1.1*bins[-1])
    axes[1,i].set_xlabel('$x$')

axes[0,0].set_yscale('log')
axes[0,0].set_ylim([0.001,1])
axes[0,0].set_yticks([0.001, 0.01, 0.1, 1])
axes[0,0].set_yticklabels(['0.1%', '1%', '10%', '100%'])
axes[1,0].set_yticks([-0.02, -0.015, -0.01, -0.005, 0, 0.005, 0.01,
                      0.015, 0.02])
    axes[1,0].set_yticklabels([-0.02, '', -0.01, '', 0, '', 0.01, '', 0.02])
    axes[0,-1].legend()
    axes[1,-1].legend()
    axes[0,0].set_ylabel('Probability distribution')
    axes[1,0].set_ylabel('Residual')
    plt.tight_layout()

    fig.savefig('random_walk.pdf', bbox_inches='tight')


start_time = time.time()
plot_dist()
print('Run time = %.1f seconds' % (time.time() - start_time))
```

The above code took 6.3 seconds to run.

For the most part, the residuals fall within error bars as expected. The two sets of error bars are close, but the Poisson error bars (red) are consistently larger than those from the histogram variance (green).



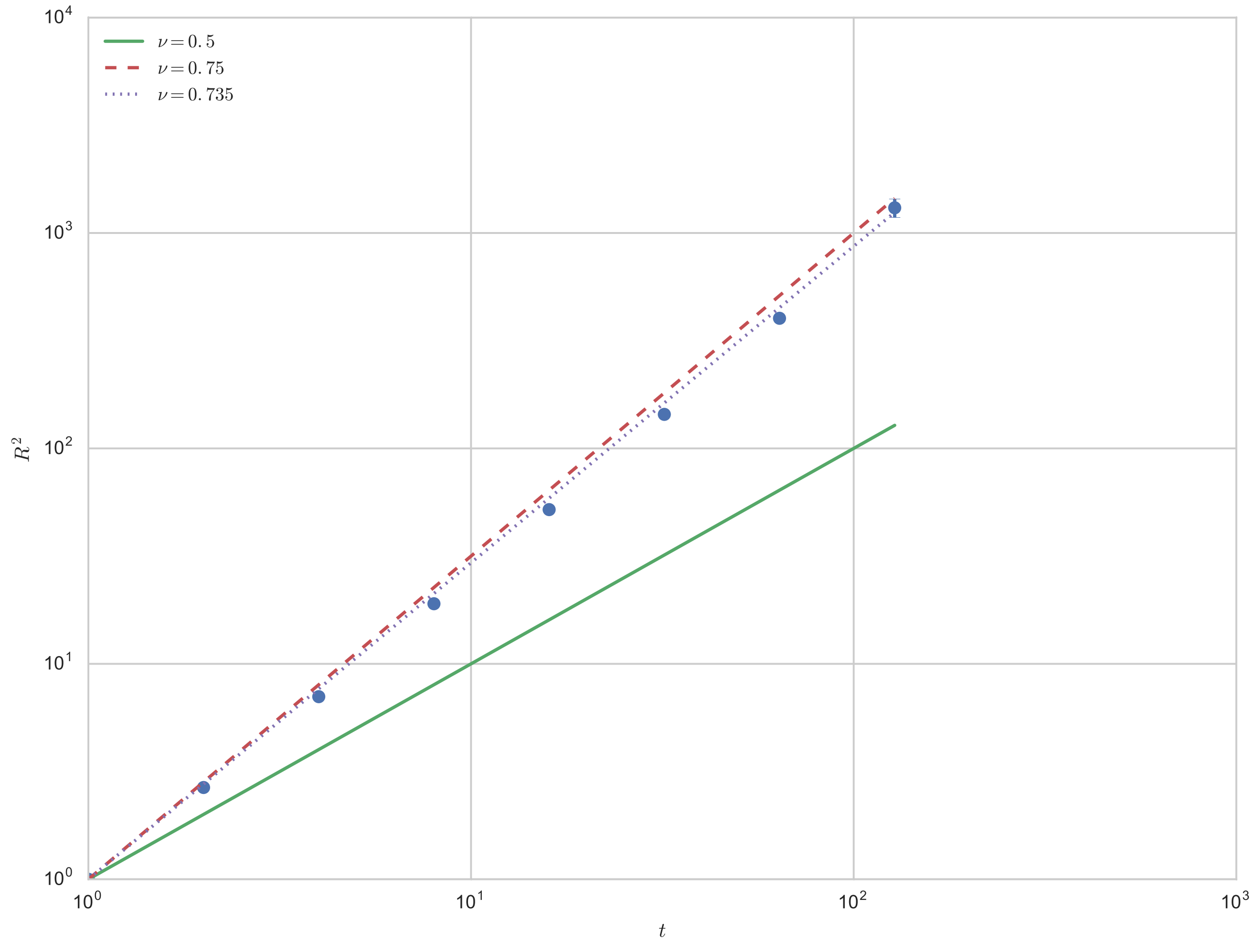
2 Self-avoiding Random Walk

```python
import numpy as np
import random as ran
import matplotlib.pylab as plt
import time
import seaborn as sns # Makes plots look nicer
from scipy.optimize import curve_fit

def step():
    # Discrete, unbiased step in 2 dimensions. Left, right, up or down
    dr = ran.choice([[-1, 0], [1, 0], [0, 1], [0, -1]])
    return np.array(dr)

def saw(t_sample):
    # Performs a self-avoiding random walk in 2 dimensions until in collides
    #  with itself. Returns R^2 values at the given sample steps

    # Start at the origin
    r = np.zeros(2)
    dr = np.zeros(2)
    r2_sample = []
    t_max = np.max(t_sample)

    # Keep track of all positions that have been visited
    visited = [list(r)]

    for t in range(1, t_max+1):
        # Generate a new 2d step
        new_dr = step()
        # Prevent backtracking to save computation time
        while (new_dr == -dr).all():
            new_dr = step()

        dr = new_dr
        r += dr

        # Break the loop if this position has already been visited
        if list(r) in visited:
            break
        else:
            visited.append(list(r))
            # Increment step number and record length if at a sample point
            if t in t_sample:
                r2_sample.append(r.dot(r))

    return r2_sample

def sample(N, t_sample):
    # Performs N self-avoiding random walks, sampling at the given steps

    # This list will contain R^2 samples at each time sample point. We use
    #  lists instead of arrays beause we don't know the array size a priori
    r2_samples = [ [] for i in range(len(t_sample)) ]
    for n in range(N):
        r2_sample = saw(t_sample)

        # Loop over time points and append to the appropriate list
        for t,r2 in enumerate(r2_sample):
            r2_samples[t].append(r2)

        # Print out progress at powers of 10
        if np.log10(n+1).is_integer():
            print('Step %d' % (n+1))

    return r2_samples

def plot_length():
    sns.set(style='whitegrid') # Comment out if not using seaborn
    N = int(1e8)
    t_sample = [2**i for i in range(11)]

    r2_samples = sample(N, t_sample)

    fig, ax = plt.subplots(figsize=(11,8))

    r2_mean = []
    r2_err = []
    r2_std = []
    counts = []
    for t,r2 in enumerate(r2_samples):
        if len(r2) > 1:
            r2_mean.append(np.mean(r2))
            r2_std.append(np.std(r2))
            r2_err.append(np.std(r2) / np.sqrt(len(r2)))
            counts.append(len(r2))

    ax.errorbar(t_sample[:len(r2_mean)], r2_mean, yerr=r2_err,
                marker='o', ls='None')
    # Save the data to a text file as well as plotting
    data = np.asarray([t_sample[:len(r2_mean)], r2_mean, r2_std, r2_err, counts]
    np.savetxt('saw.csv', data, delimiter=',')

    power_law = lambda x, nu: x**nu
    L = np.array(t_sample[:len(r2_mean)])
    # First, the scaling of a typical random walk
    R = power_law(L, 0.5)
    ax.plot(L, R**2, '-', label=r'$\nu = 0.5$')
    # Second, the two-dimensional theoretical exponent (given by Sethna)
    R = power_law(L, 0.75)
    ax.plot(L, R**2, '--', label=r'$\nu = 0.75$'))
    # Third, a least squares fit
    popt, pcov = curve_fit(power_law, L, np.sqrt(r2_mean))
    nu = popt[0]
    R = power_law(L, nu)
    ax.plot(L, R**2, ':', label=r'$\nu = %.3f$' % nu)

    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.set_ylabel('$R^2$')
    ax.set_xlabel('$t$')

    ax.legend()
    fig.savefig('saw.pdf', bbox_inches='tight')

start_time = time.time()
plot_length()
print('Run time = %.1f seconds' % (time.time() - start_time))
```

This program ran for approximately 10 hours to complete $N = 1e8$ random walks, but recorded no walks at times longer than 2^7 (128) steps. The counts per step were as follows:

t	random walks
1	100000000
2	100000000
4	92594705
8	67620778
16	30042667
32	4816814
64	99311
128	34
256	0

As expected, the scaling of $R \sim L^\nu$ was different from a simple random walk with $\nu = 0.5$. From the best fit, the data scales as $\nu = 0.735$, close to the given theoretical value $\nu = 0.75$.


$\nu = 0.5$
$\nu = 0.75$
$\nu = 0.735$
R^2
t
10^0
10^1
10^2
10^3
10^4