main_OU_Code

February 11, 2025

0.1 Applications of Stochastic Processes to Population Modelling: Ornstein-Uhlenbeck Process (OU)

This Note aims to show how the Ornstein-Uhlenbeck process, a mean-reverting SDE, can be used to model birth/death rates of a population. We aim to:

- Discuss population models (birth rate, death rate, overall population) and how we can use stochastic models to reflect real world data
- Give an overview of the mathematics underlying the Stochastic Process (Derivation, Mean/Variance/Covariance, Limiting Distribution)
- Demonstrate how to identify a mean-reverting process using Unit Root Tests from Time Series Analysis (Dickey-Fuller test for AR(1) Process)
- Use Maximum Likelihood Estimation to estimate the parameters for the SDE based on a sample (historical data for birth/death rates of a country)
- Collect and store the data and use it to identify an OU process and calibrate our model
- Display the results on some graphs

A mean-reverting Ornstein-Uhlenbeck process X_t with parameters μ, θ, σ is characterised by the stochastic differential equation

$$dX_t = \theta(\mu - X_t)dt + \sigma dW_t$$

where W_t is a standard Brownian Motion and $X_0 = x_0$. The OU process is part of a family of diffusion processes widely used to model stochastic dynamics, most notably in finance for interest rates. The model is also used in population dynamics, which we will delve into later.

The SDE above is interpreted as describing a linear drift process where the drift term (dt component) prescribes a mean-reversion of X_t towards the long term mean μ with an additional parameter θ describing the rate/speed of reversion.

Solving the Ornstein-Uhlenbeck SDE One can show that the analytical solution of the OU process is

$$X_t = \mu + e^{-\theta T}(X_0 - \mu) + \sigma \int_0^T e^{-\theta(T-t)} dW_t$$

We can conclude that X_t is normally distributed with the following for the mean, variance and covariance:

$$\mathbb{E}[X_t] = \mu + e^{-\theta T}(X_0 - \mu)$$

$$V[X_t] = \frac{\sigma^2}{2\theta}\left(1 - e^{-2\theta T}\right)$$

$$C[X_t, X_S] = \frac{\sigma^2}{2\theta}\left(e^{-\theta|T-S|} - e^{-\theta(T+S)}\right)$$

Derivations for these formulae can be found in the Appendix. (INCLUDE LIMITING DISTRIBUTION LATER ON) #### Discretisation of Ornstein-Uhlenbeck SDE

While it's good to have the analytical formulae, if we want to simulate the SDE on a graph a discretised version is used instead. Discretised analogs are also often used because analytical solutions to an SDE can't be found, so one has to obtain a numerical solution instead. One widely used approach to obtain this is the **Euler-Maruyama** approximation, where for a general SDE

$$X_t = a(X_t, t)dt + b(X_t, t)dW_t$$

where $X_0 = x_0$, we can approximation the solution on a time interval $[0, T]$ by using the following Markov chain:

- Partition the interval into N equal subintervals of width $\Delta t = T/N$ and $0 = \tau_0 < \tau_1 < ... < \tau_N = T$.
- With $Y_0 = x_0$, using recursion define Y_n as

$$Y_{n+1} = Y_n + a(Y_n, \tau_n)\Delta t + b(Y_n, \tau_n)(W_{\tau_n+1} - W_{\tau_n})$$

By the properties of Brownian Motion, one can write $(W_{\tau_n+1} - W_{\tau_n}) = \sqrt{dt}\epsilon_t$, where $\epsilon_t \sim N(0, 1)$. Applying this to our OU process, the Euler-Maruyama discretisation is

$$x_{t+1} = x_t + \theta(\mu - x_t)\Delta t + \sigma\sqrt{dt}\epsilon_t$$

This formula will be used later on in Python code.

Parameter Estimation for Ornstein-Uhlenbeck SDE In our model we have three parameters that need to be set to simulate our SDE: - θ : speed of reversion - μ : long-term mean - σ : variance (volatility)

Often we will have real world data that can be fitted to a particular mathematical model. The reason why we might fit a model to data is because this can allow us to make predictions for future values of the data. If we determine that such a dataset (in our case, a time series) follows a mean-reverting process, **we can use the data observed to infer what the values of our parameters might be.** Once we have determined the correct values, we can fit the model as close as possible to our dataset. The inference method we will use here is **Maximum Likelihood Estimation (MLE)**

To give a more formal statement; given $n + 1$ samples $\{x_0, x_1, ..., x_n\} = \vec{\mathbf{x}}$ at times $t_0, t_1, ..., t_n$ respectively, the vector $\Theta = [\theta, \mu, \sigma]$ can be estimated using maximum likelihood.

Going back to our OU process, we can start by defining the Conditional Distribution:

$$X_t|X_{t-1} \sim N\left(X_{t-1}e^{-\theta\Delta t} + \mu\left(1 - e^{-\theta\Delta t}\right), \frac{\sigma^2}{2\theta}\left(1 - e^{-2\theta\Delta t}\right)\right)$$

where we define $\Delta t = t_{i+1} - t_i \ \forall i$ such that $1 \leq i \leq n$. We assume that all time observations are evenly spaced for simplicity.

```
[116]: import pandas as pd
       import matplotlib.pyplot as plt
       import numpy as np
       import statistics as stats
       import ipywidgets as widgets
       from ipywidgets import interact, interactive, fixed, interact_manual, Layout
       import statsmodels.tsa.stattools as ts
       from scipy.stats import norm
       from scipy.optimize import minimize
       import re
```

```
[ ]: data = pd.read_csv("EcuadorPopulation.csv")
     #data = pd.read_csv("finland.csv")
     #data = pd.read_csv("world-population.csv")
     population = data.loc[0]
     birthrate = data.loc[1]
     deathrate = data.loc[2]

     start_year = 1970
     end_year = 2019
     years = [str(year) + ' [YR' + str(year) + ']' for year in range(start_year,␣
      ↪end_year)]


     def extract_data(df):
         df = df[years]   # Extract the specific row and filter columns
         df = df.reset_index()
         df.columns = ['years', 'population']
         df = df["population"].tolist()
         return df

     birthrate = extract_data(birthrate)
     deathrate = extract_data(deathrate)
     population = extract_data(population)

     '''birthrate = [i/1000 for i in birthrate]
     deathrate = [i/1000 for i in deathrate]'''

     #print(birthrate)
     #print(deathrate)
     #print(population)
```

```
#print(type(birthrate))
```

```
[ ]: 'birthrate = [i/1000 for i in birthrate]\ndeathrate = [i/1000 for i in
deathrate]'
```

```
[106]: def augmented_dickey_fuller(goog):
           # Output the results of the Augmented Dickey-Fuller test for Google
           # with a lag order value of 1
           adf = ts.adfuller(goog, 1)
           print(adf)

       print("DF Test for Death Rate:")
       augmented_dickey_fuller(deathrate)
       print("\n")
       print("DF Test for Birth Rate:")
       augmented_dickey_fuller(birthrate)
```

```
DF Test for Death Rate:
(-3.526479403011003, 0.007332525846549103, 1, 47, {'1%': -3.5778480370438146,
'5%': -2.925338105429433, '10%': -2.6007735310095064}, -175.62897132822167)


DF Test for Birth Rate:
(-1.2816239969526604, 0.6375368675439361, 1, 47, {'1%': -3.5778480370438146,
'5%': -2.925338105429433, '10%': -2.6007735310095064}, -78.80126913246113)
```

```
[107]: def get_mean_birth_rate(df):
           sum = 0
           for i in range(len(df)):
               sum += df[i]
           sum = sum/len(df)
           return sum

       mean = get_mean_birth_rate(birthrate)
       print(mean)

       def get_mean_death_rate(df):
           sum = 0
           for i in range(len(df)):
               sum += df[i]
           sum = sum/len(df)
           return sum

       mean_birth_rate = get_mean_birth_rate(birthrate)
       print("Mean Birth Rate from " + str(start_year) + " to " + str(end_year) + ": "␣
        ↪+ str(round(mean_birth_rate, 5)))

       mean_death_rate = get_mean_death_rate(deathrate)
```

```
print("Mean Death Rate from " + str(start_year) + " to " + str(end_year) + ": "␣
↪+ str(round(mean_death_rate, 5)))
```

```
28.597551020408158
Mean Birth Rate from 1970 to 2019: 28.59755
Mean Death Rate from 1970 to 2019: 6.52465
```

[108]:
```
def OU(x1, x2, dt, theta, mu, sigma):
    sigma0 = sigma**2 * (1 - np.exp(-2*mu*dt)) / (2 * mu)
    sigma0 = np.sqrt( sigma0 )

    prefactor = 1 / np.sqrt(2 * np.pi * sigma0**2)

    f =  prefactor * np.exp( -(x2 - x1 * np.exp(-mu*dt) - \
                    theta * (1-np.exp(-mu*dt)) )**2 / (2 * sigma0**2) )

    return f

#  Calculate the negative of the log likelihood
def log_likelihood_OU(p, X, dt):

    theta = p[0]
    mu = p[1]
    sigma = p[2]

    N = len(X)

    f = np.zeros( (N-1, ) )

    for i in range( 1, N ):
        x2 = X[i]
        x1 = X[i-1]

        f[i-1] = OU(x1, x2, dt, theta, mu, sigma)

    ind = np.where(f == 0)
    ind = ind[0]
    if ind.size > 0:
        f[ind] = 10**-8

    f = np.log(f)
    f = np.sum(f)

    return -f

#  mu and sigma must be greater than zero.  We use these contraint functions␣
↪with minimze
```

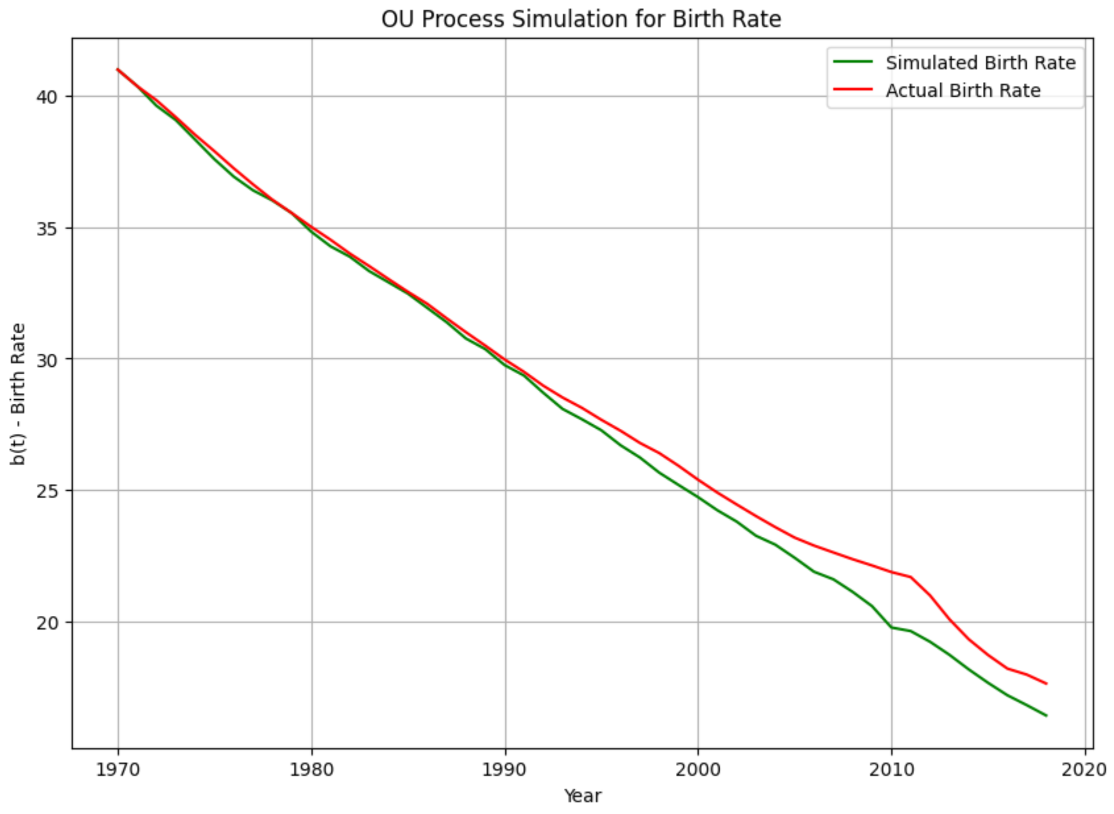
```python
def constraint1( p ):
    return p[1]

def constraint2( p ):
    return p[2]


#  Add constraint function to a dictionary
cons = ( {'type':'ineq', 'fun': constraint1},
         {'type':'ineq', 'fun': constraint2} )

#  Initial guess for our parameters
p0 = [1, 1, 1]

#  Call minimize

output_deathrate = minimize(log_likelihood_OU, p0, args = (deathrate, 1/
 ↪len(deathrate)), constraints=cons)
print(output_deathrate)
[mu_optimised_death, gamma_optimised_death, sigma_optimised_death] =␣
 ↪output_deathrate["x"]
print(mu_optimised_death)

#  Add constraint function to a dictionary
cons = ( {'type':'ineq', 'fun': constraint1},
         {'type':'ineq', 'fun': constraint2} )

#  Initial guess for our parameters
p0 = [1, 1, 1]

output_birthrate = minimize(log_likelihood_OU, p0, args = (birthrate, 1/
 ↪len(birthrate)), constraints=cons)
print(output_birthrate)
[mu_optimised_birth, gamma_optimised_birth, sigma_optimised_birth] =␣
 ↪output_birthrate["x"]
print(mu_optimised_birth)
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -65.14612163159057
       x: [ 4.287e+00  3.361e+00  4.510e-01]
     nit: 14
     jac: [ 1.656e-03 -3.510e-04  4.286e-03]
    nfev: 67
    njev: 14
4.28733258104319
```

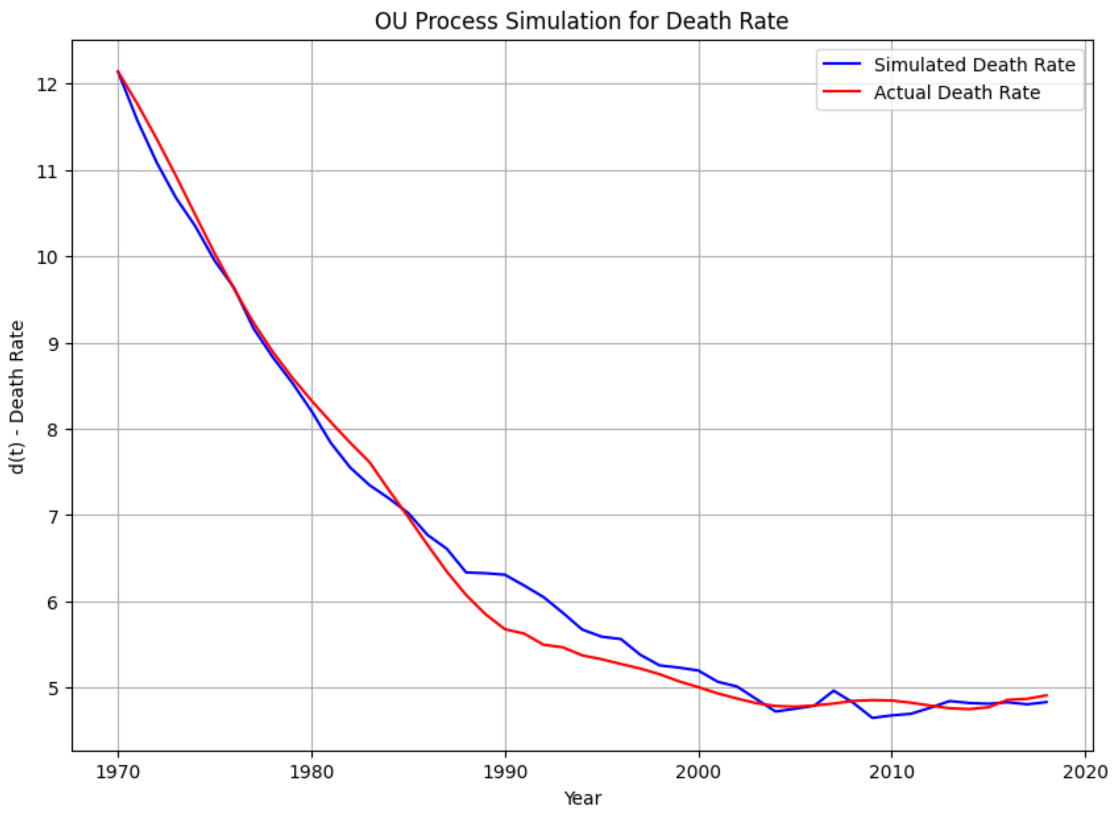
```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -30.248598021887588
       x: [-2.786e+01  4.218e-01  9.058e-01]
     nit: 22
     jac: [ 1.721e-04 -1.935e-02 -1.684e-03]
    nfev: 95
    njev: 22
-27.862241606163078
```

[109]:
```python
def plot_results_birthrate(gamma_b, sigma_b):

    #gamma_b = 0.7
    #b_e = mean_birth_rate

    b_e = mu_optimised_birth

    X_0 = birthrate[0]
    T = len(birthrate)
    dt = 1/T
    #N = int(T/dt)
    N = len(birthrate)
    print(N)
    X = np.zeros(N)
    X[0] = X_0


    X_actual = birthrate
    X_actual = np.array(X_actual)

    #print(X_actual.shape)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]

    #print(X_actual)
    for t in range(1,N):
        dW = np.sqrt(dt) * np.random.normal(0,1)
        X[t] = X[t-1] + gamma_b * (b_e - X[t-1]) * dt + sigma_b * dW

    plt.figure(figsize=(10,7))
    plt.plot(year_labels, X, color="g", label = "Simulated Birth Rate")
    plt.plot(year_labels, X_actual, color="r", label = "Actual Birth Rate")
    plt.title("OU Process Simulation for Birth Rate")
    plt.legend()
    plt.grid(True)
    plt.xlabel("Year")
    plt.ylabel("b(t) - Birth Rate")

    return X, X_actual, year_labels

X, X_actual, year_labels = plot_results_birthrate(gamma_optimised_birth,␣
 ↪sigma_optimised_birth)
```

```
49
```



[110]:
```
def plot_results_deathrate(gamma_d,sigma_d):

    d_e = mu_optimised_death
    gamma_d = gamma_optimised_death
    sigma_d = sigma_optimised_death

    X_0 = deathrate[0]
    T = len(deathrate)
    dt = 1/T
    N = len(deathrate)
    X = np.zeros(N)
    X[0] = X_0

    X_actual = deathrate
    X_actual = np.array(X_actual)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]

    for t in range(1,N):
        dW = np.sqrt(dt) * np.random.normal(0,1)
        X[t] = X[t-1] + gamma_d * (d_e - X[t-1]) * dt + sigma_d * dW

    plt.figure(figsize=(10,7))
    plt.plot(year_labels, X, color="b", label = "Simulated Death Rate")
    plt.plot(year_labels, X_actual, color="r", label = "Actual Death Rate")
    plt.title("OU Process Simulation for Death Rate")
    plt.legend()
    plt.grid(True)
    plt.xlabel("Year")
    plt.ylabel("d(t) - Death Rate")
    plt.show()


    return X, X_actual, year_labels

X, X_actual, year_labels = plot_results_deathrate(gamma_optimised_death,␣
 ↪sigma_optimised_death)


def get_results_deathrate(gamma_d,sigma_d):

    d_e = mu_optimised_death
    gamma_d = gamma_optimised_death
    sigma_d = sigma_optimised_death

    X_0 = deathrate[0]
    T = len(deathrate)
    dt = 1/T
    N = len(deathrate)
    X = np.zeros(N)
    X[0] = X_0

    X_actual = deathrate
    X_actual = np.array(X_actual)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]
    for t in range(1,N):
        dW = np.sqrt(dt) * np.random.normal(0,1)
        X[t] = X[t-1] + gamma_d * (d_e - X[t-1]) * dt + sigma_d * dW

    return X, X_actual, year_labels


def get_results_birthrate(gamma_b, sigma_b):

    b_e = mu_optimised_birth
    gamma_b = gamma_optimised_birth
    sigma_b = sigma_optimised_birth

    X_0 = birthrate[0]
    T = len(birthrate)
    dt = 1/T
    #N = int(T/dt)
    N = len(birthrate)
    X = np.zeros(N)
    X[0] = X_0


    X_actual = birthrate
    X_actual = np.array(X_actual)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]

    for t in range(1,N):
        dW = np.sqrt(dt) * np.random.normal(0,1)
        X[t] = X[t-1] + gamma_b * (b_e - X[t-1]) * dt + sigma_b * dW

    return X, X_actual, year_labels
```



```
[111]: def sim_d(gamma_d, sigma_d):
           simulations = 100
           plt.figure(figsize=(10, 7))
           for i in range(simulations):
               X, X_actual, year_labels = get_results_deathrate(gamma_d, sigma_d)
               plt.plot(year_labels, X, alpha=0.25)

           plt.plot(year_labels, X_actual, color="g", label="Actual Death Rate")

           plt.title("Simulation for Death Rate")
           plt.legend()
           plt.grid(True)
           plt.xlabel("Year")
           plt.ylabel("D(t) - Death Rate")
           plt.show()


       # Define your sliders
       gamma_d_slider = widgets.FloatSlider(min=gamma_optimised_death-10,␣
        ↪max=gamma_optimised_death+10, step=0.1, value=gamma_optimised_death,␣
        ↪description="Rate of Reversion")
```

```
sigma_d_slider = widgets.FloatSlider(min=sigma_optimised_death - 1,␣
↪max=sigma_optimised_death + 1, step=0.01, value=sigma_optimised_death,␣
↪description="Volatility")

# Create a grid layout for the sliders (2x3 layout)
grid_layout = widgets.GridBox(
    children=[gamma_d_slider, sigma_d_slider],
    layout=Layout(grid_template_columns="repeat(2, 300px)",␣
↪grid_template_rows="repeat(1, auto)", grid_gap="10px")
)



# Display the interactive widgets and buttons in a grid layout
ui = widgets.VBox([grid_layout])
out = widgets.interactive_output(sim_d, {
    'gamma_d': gamma_d_slider,
    'sigma_d': sigma_d_slider
})

display(ui, out)
```

```
VBox(children=(GridBox(children=(FloatSlider(value=3.3613415649934395,␣
↪description='Rate of Reversion', max=13…
```

```
Output()
```

[ ]:
```
def sim_b(gamma_b, sigma_b):
    simulations = 100
    plt.figure(figsize=(10, 7))
    for i in range(simulations):
        X, X_actual, year_labels = get_results_birthrate(gamma_b, sigma_b)
        plt.plot(year_labels, X, alpha=0.25)
    plt.plot(year_labels, X_actual, color="g")

    plt.title("Simulation for Birth Rate")
    plt.legend()
    plt.grid(True)
    plt.xlabel("Year")
    plt.ylabel("B(t) - Birth Rate")
    plt.show()


# Define your sliders
gamma_b_slider = widgets.FloatSlider(min=gamma_optimised_birth-10,␣
↪max=gamma_optimised_birth +10, step=0.1, value=gamma_optimised_birth,␣
↪description="Rate of Reversion")
```

```
sigma_b_slider = widgets.FloatSlider(min = sigma_optimised_birth - 1, max =␣
 ↪sigma_optimised_birth+1, step=0.01, value=sigma_optimised_birth,␣
 ↪description="Volatility")

# Create a grid layout for the sliders (2x3 layout)
grid_layout = widgets.GridBox(
    children=[gamma_b_slider, sigma_b_slider],
    layout=Layout(grid_template_columns="repeat(2, 300px)",␣
 ↪grid_template_rows="repeat(1, auto)", grid_gap="10px")
)



# Display the interactive widgets and buttons in a grid layout
ui = widgets.VBox([grid_layout])
out = widgets.interactive_output(sim_b, {
    'gamma_b': gamma_b_slider,
    'sigma_b': sigma_b_slider
})

display(ui, out)
```

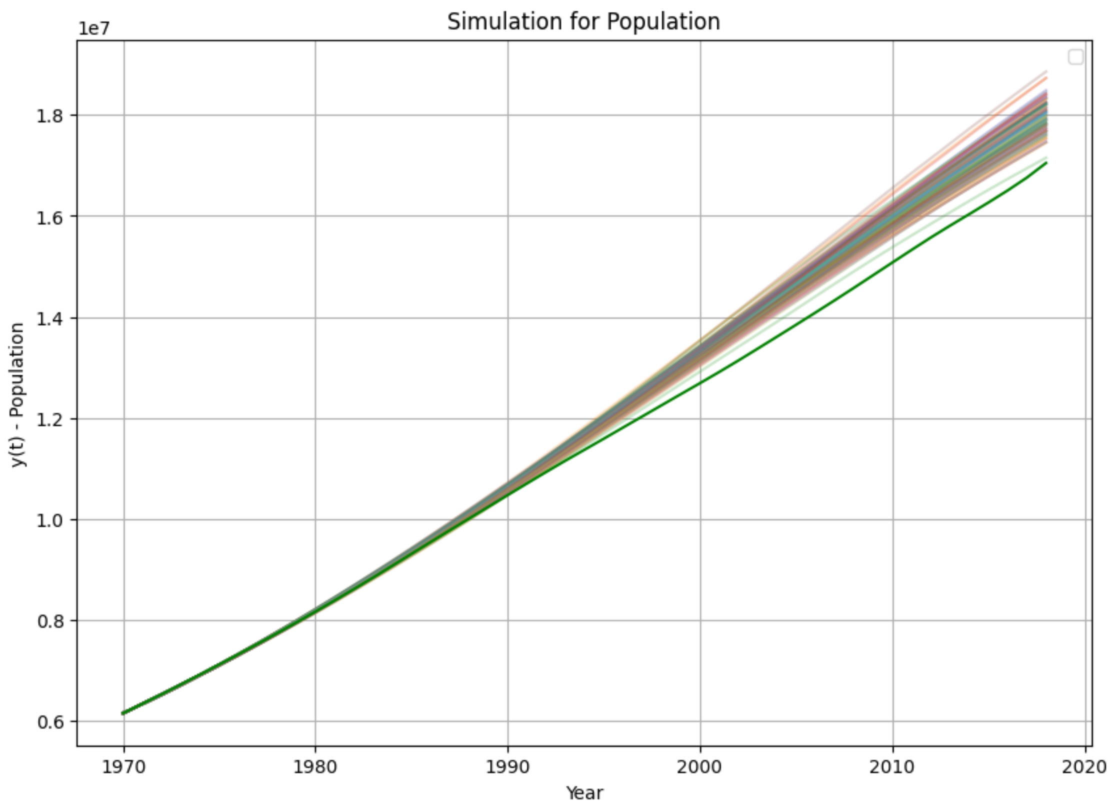
```
VBox(children=(GridBox(children=(FloatSlider(value=0.42176906743735226,␣
 ↪description='Rate of Reversion', max=1…

Output()
```

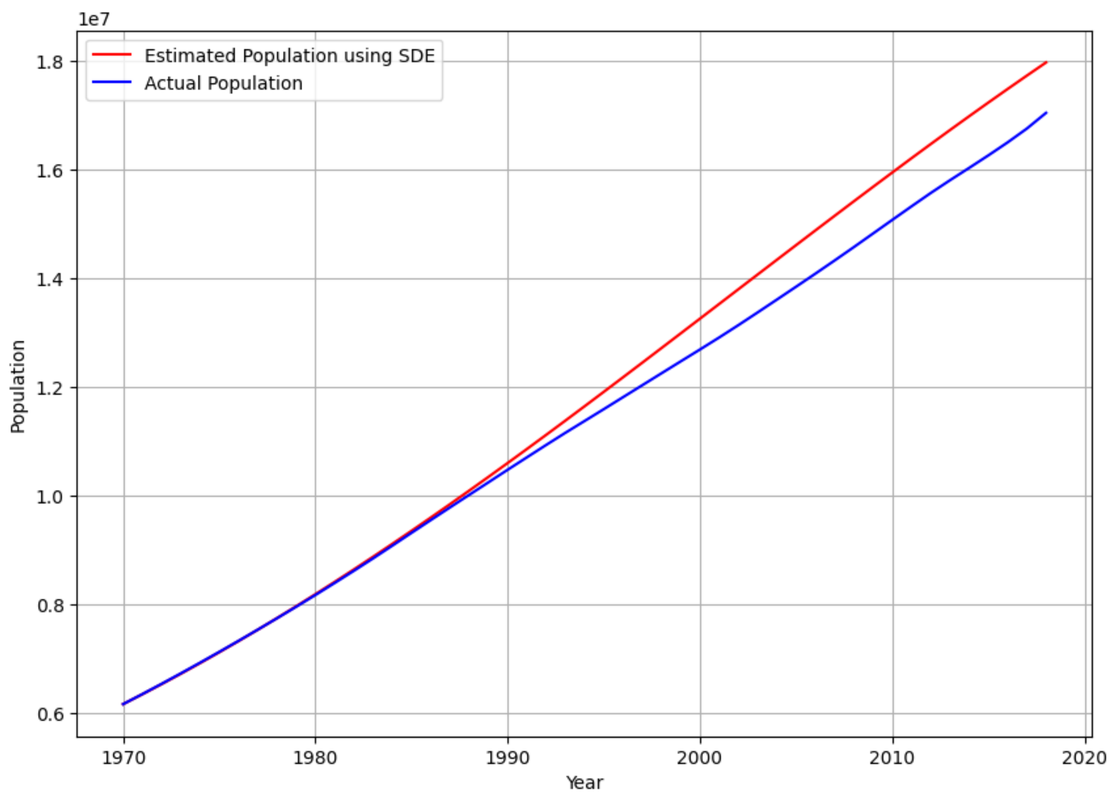
[113]:
```
def get_population_model(factor):



    b_e = mu_optimised_birth*factor
    sigma_b = sigma_optimised_birth*factor
    gamma_b = gamma_optimised_birth

    d_e = mu_optimised_death*factor
    gamma_d = gamma_optimised_death
    sigma_d = sigma_optimised_death*factor

    B_0 = birthrate[0]*factor
    D_0 = deathrate[0]*factor
    Y_0 = population[0]
    T = len(deathrate)
    #dt = 1/T
    #N = int(T/dt)
    dt = 1/T
    N = len(deathrate)

    Y = np.zeros(N)
    Y[0] = Y_0

    B = np.zeros(N)
    B[0] = B_0

    D = np.zeros(N)
    D[0] = D_0

    Y_actual = population
    Y_actual = np.array(Y_actual)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]

    for t in range(1,N):
        dW1 = np.sqrt(dt) * np.random.normal(0,1) # Population Weiner Process
        dW2 = np.sqrt(dt) * np.random.normal(0,1) # Birth Rate Weiner Process
        dW3 = np.sqrt(dt) * np.random.normal(0,1) # Death Rate Weiner Process

        B[t] = max(B[t-1] + gamma_b * (b_e - B[t-1]) * dt + sigma_b * dW2,0)
        D[t] = max(D[t-1] + gamma_d * (d_e - D[t-1]) * dt + sigma_d * dW3,0)

        Y[t] = Y[t-1] + (B[t-1] - D[t-1]) * Y[t-1] * (dt) + np.
 ↪sqrt(Y[t-1]*(B[t-1] + D[t-1])) * dW1

    return Y

Y = get_population_model(1/20)
```

[114]:
```
def main(factor):

    simulations = 100
    plt.figure(figsize=(10,7))
    Y_actual = population
    Y_actual = np.array(Y_actual)

    x_vals = list(range(len(years)))
    year_labels = [start_year + i for i in x_vals]
    Y_vault = []
    for i in range(simulations):
        Y = get_population_model(factor)
        Y_vault.append(Y)
        plt.plot(year_labels, Y, alpha=0.25)
    plt.plot(year_labels, Y_actual, color="g")

    plt.title("Simulation for Population")
    plt.legend()
```

```python
    plt.grid(True)
    plt.xlabel("Year")
    plt.ylabel("y(t) - Population")
    plt.show()

    return Y_vault

Y_vault = main(1/20)

mean_pop_per_year = []
for i in range(len(Y_vault[0])):

    year_slice = []

    for j in range(len(Y_vault)):
        year_slice.append(Y_vault[j][i])

    mean_pop_per_year.append(np.mean(year_slice))

#print(mean_pop_per_year)

plt.figure(figsize=(10,7))
plt.plot(year_labels, mean_pop_per_year, color='r', label="Estimated Population␣
  ↪using SDE")
plt.plot(year_labels, population, color="b", label="Actual Population")
plt.legend()
plt.grid(True)
plt.xlabel("Year")
plt.ylabel("Population")
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```


Simulation for Population
1e7
1.8
1.6
1.4
1.2
1.0
0.8
0.6
y(t) - Population
1970
1980
1990
2000
2010
2020
Year



[115]:
```
# Define your sliders
factor_slider = widgets.FloatSlider(min=0.001, max=1, step=0.001, value=0.1,␣
 ↪description="Factor")

# Create a grid layout for the sliders (2x3 layout)
grid_layout = widgets.GridBox(
    children=[factor_slider],
    layout=Layout(grid_template_columns="repeat(1, 300px)",␣
 ↪grid_template_rows="repeat(1, auto)", grid_gap="10px")
)



# Display the interactive widgets and buttons in a grid layout
ui = widgets.VBox([grid_layout])
out = widgets.interactive_output(main, {
    'factor': factor_slider
})

display(ui, out)
```

```
VBox(children=(GridBox(children=(FloatSlider(value=0.1, description='Factor', 
↪max=1.0, min=0.001, step=0.001),…
```

```
Output()
```