Problem 1

Brute Force Simulation

We will consider a simple two-state (A and B) stochastic process, characterized by the dynamics:

$$A \xrightarrow{\omega_{AB}} B;$$

$$B \xrightarrow{\omega_{BA}} A,$$

with transition rates $\omega_{AB} = 0.5$ and $\omega_{BA} = 1.5$.

At first, we are going to simulate it using a simple brute force algorithm for a time $T = 100$ time units (t.u.), with a time-step $\mathrm{d}t = 10^{-3}$.

First, we import the necessary packages:

In [1]:

```
import numpy as np # Dealing with arrays
import matplotlib.pyplot as plt # Plotting
from tqdm import tqdm # Printing progressbars
from numba import njit # Pre-compilating functions for better performance
from scipy.optimize import curve_fit # Fitting curves
import time # Counting time
```

Now we write two functions for:

1. Generating an array with the evolution of the system;
2. Running multiple realizations while saving the statistics of state occupation, over multiple realizations of the process.

In [2]:

```
# Generating array with time evolution of a two-state stochastic process via
@njit # Numba pre-compilation decorator
def BF2S_evol(T, dt, w_01, w_10): # We identify the states as A -> 0, B -> 1
    # T: total time
    # dt: time-step for integration
    # w_01: transition rate \omega_{AB}
    # w_10: transition rate \omega_{BA}

    nt = int(T/dt) # Number of time points
    t = np.linspace(0, T, nt) # Array with time data
    state = np.zeros(nt) # Array to store states through time, and initial st

    # Loop over time
    for n in range(1, nt):
        if state[n-1] == 0: # If on state A, transition to B with rate \omega
            if np.random.random() < w_01*dt:
                state[n] = 1 # Change state
            else:
                state[n] = 0 # Maintain state
        else: # If on state B, transition to A with rate \omega_{BA}
            if np.random.random() < w_10*dt:
                state[n] = 0 # Change state
            else:
                state[n] = 1 # Maintain state
```

```
    return t, state # Return time and state data
BF2S_evol(1, 0.1, 0, 0); # Runs once for pre compilation

# Running multiple realizations while saving the statistics of state occupati
# Brute Force algortithm for Two State System
@njit
def BF2S_hist(T, dt, w_01, w_10, realizations):
    # T: total time
    # dt: time-step for integration
    # w_01: transition rate \omega_{AB}
    # w_10: transition rate \omega_{BA}
    # realizations: number of realizations of the process to go over

    nt = int(T/dt) # Number of time points

    state_hist = np.zeros((realizations, 2)) # Array to store proportion of c

    for i in range(realizations):
        state = 0 # Initial state = 0 (A)

        state_hist[i,0] = 1 # Count firs state

        # Loop over time
        for n in range(1, nt):
            if state == 0: # If on state A, transition to B with rate \omega_
                if np.random.random() < w_01*dt:
                    state = 1 # Change state
                    state_hist[i,1] += 1 # Count state
                else:
                    state_hist[i,0] += 1 # Count state
            else: # If on state B, transition to A with rate \omega_{BA}
                if np.random.random() < w_10*dt:
                    state = 0 # Change state
                    state_hist[i,0] += 1 # Count state
                else:
                    state_hist[i,1] += 1 # Count state

    return state_hist / nt # Return state occupations statistics for each rea
BF2S_hist(1, 0.1, 0, 0, 1);
```

Let's now generate a few realizations and look at the time evolution:

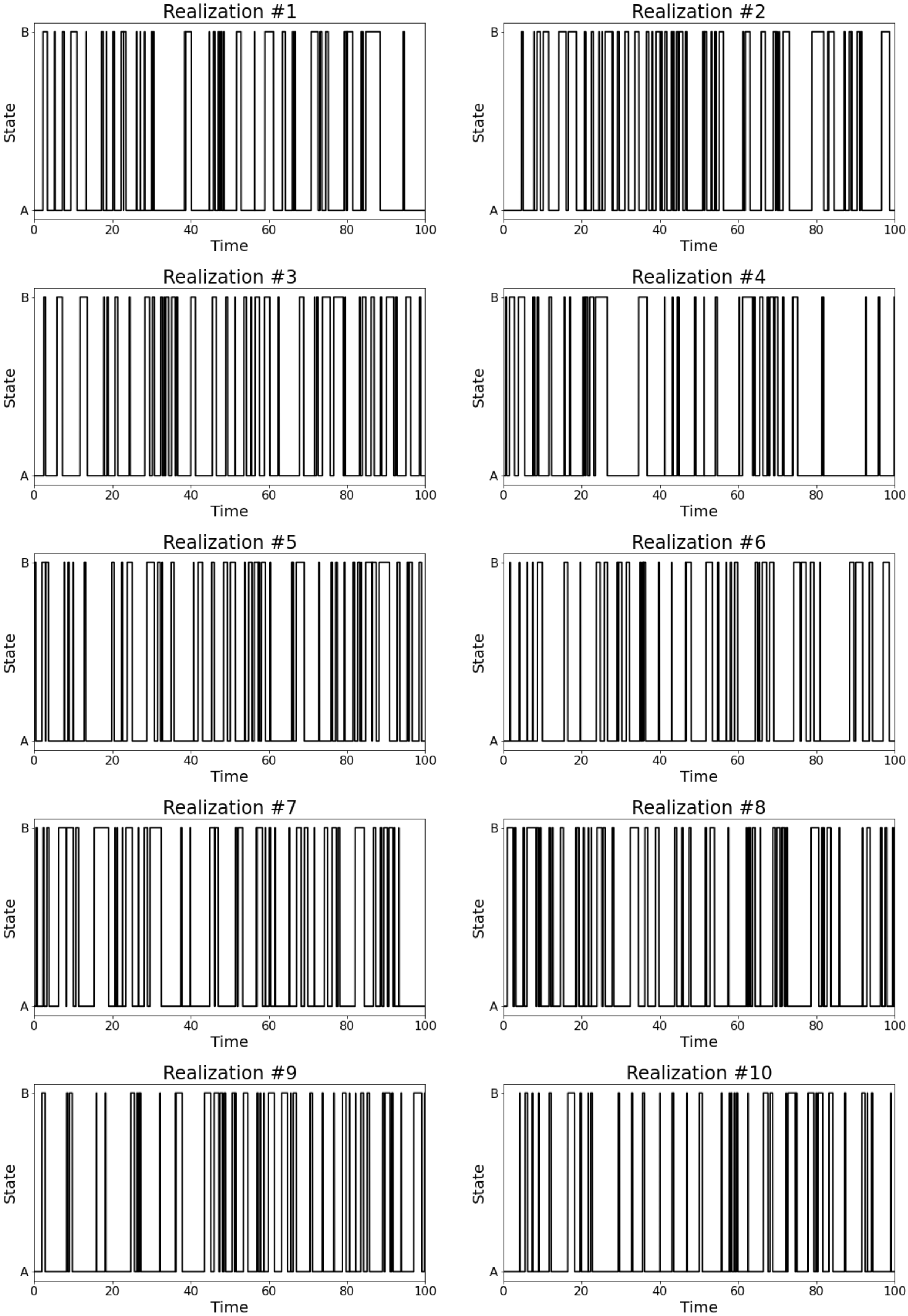
In [3]:

```
# Set parameters
T = 100 # Simulation duration (t.u.)
dt = 1e-3 # Integration time-step

w_01 = 0.5 # \omega_{AB}
w_10 = 1.5 # \omega_{BA}
```

In [4]:

```
# Plotting
plt.subplots(5, 2, figsize=(20, 30)) # Create subplot grid
plt.subplots_adjust(hspace=0.35) # Adjust space between subplots
for i in range(10): # For 10 realizations
    plt.subplot(5, 2, i+1) # Choose subplot to cover

    t, s = BF2S_evol(T, dt, w_01, w_10) # Generate a realization of the proce

    # Plot data and edit graph
    plt.plot(t, s, c="k", lw=2)
    plt.xlabel("Time", fontsize=20)
    plt.ylabel("State", fontsize=20)
```

```
    plt.xticks(fontsize=16)
    plt.yticks([0, 1], labels=["A", "B"], fontsize=16)
    plt.title(f"Realization #{i+1}", fontsize=24)
    plt.xlim(0, T)
plt.show() # Show plot
```

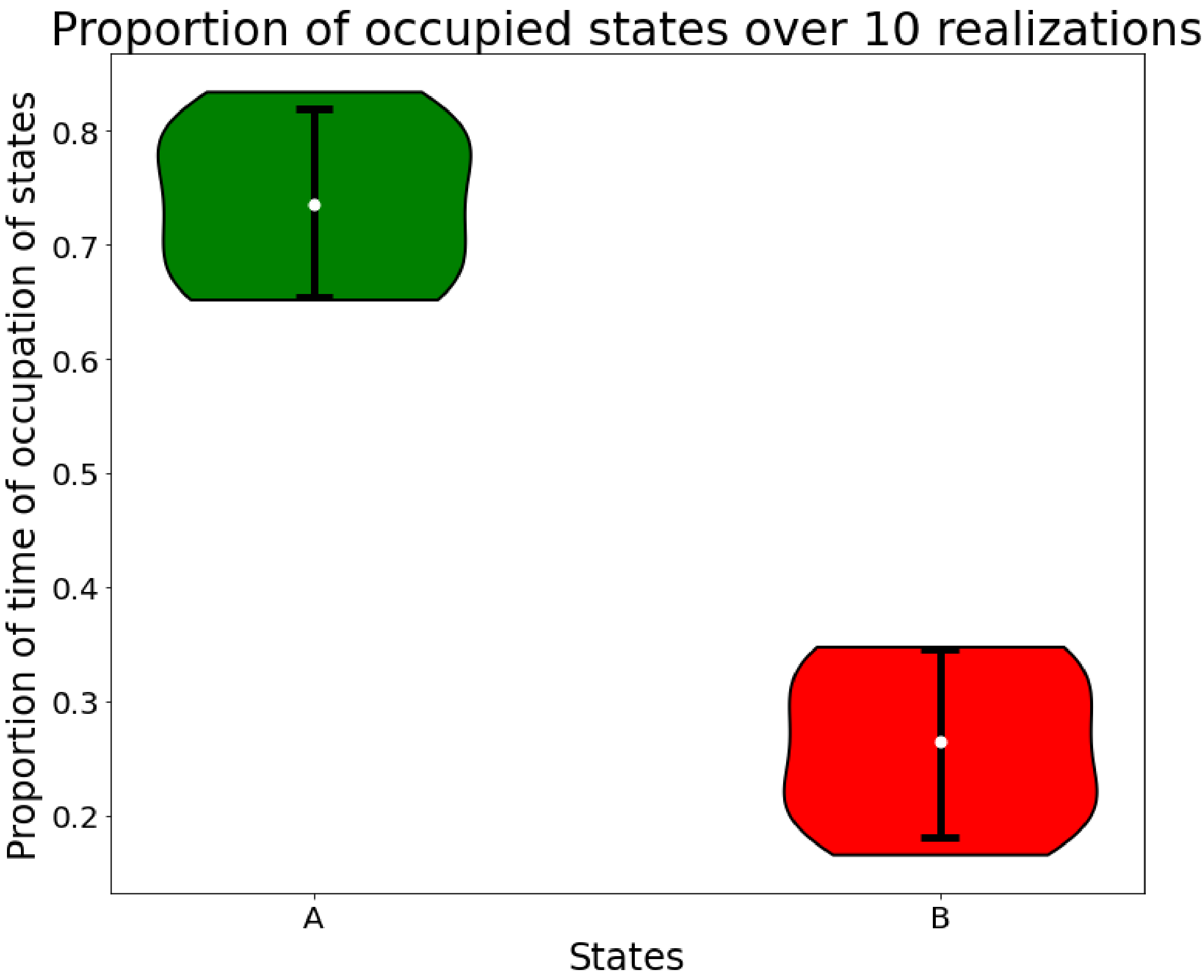


We can clearly see the behavior of the system, but its hard to do a quantitative analysis of the results looking at the data like this. Let's take the statistics of how many of the points are on each state, considering all figures. That is, let's look at the state occupation statistics.

As $\omega_{BA} = 3\omega_{BA}$, we would expect the system to spend 75% of the time on state A, and the rest on B. Let's look at the actual results.

In [5]:

```
realizations = 10 # Choose number of realizations to consider

state_hist = BF2S_hist(T, dt, w_01, w_10, realizations) # Generate state occu

# The rest is just plot editting

# Calculate quantiles to show
quantiles0 = np.percentile(state_hist[:,0], [5, 50, 95])
quantiles1 = np.percentile(state_hist[:,1], [5, 50, 95])

plt.figure(figsize=(12,10)) # Create figure

violin = plt.violinplot(state_hist, showextrema=False, showmeans=False) # Ger

# Edit violins
violin["bodies"][0].set_facecolor("green")
violin["bodies"][0].set_edgecolor("k")
violin["bodies"][0].set_linewidth(2)
violin["bodies"][0].set_alpha(1)
violin["bodies"][1].set_facecolor("red")
violin["bodies"][1].set_edgecolor("k")
violin["bodies"][1].set_linewidth(2)
violin["bodies"][1].set_alpha(1)

# More plot editing

plt.xticks([1, 2], labels=["A", "B"], fontsize=20)
plt.yticks(fontsize=20)
plt.title(f"Proportion of occupied states over {realizations} realizations",

plt.vlines(1, quantiles0[0], quantiles0[-1], color="k", lw=5)
plt.hlines(quantiles0[0], 0.97, 1.03, color="k", lw=5)
plt.hlines(quantiles0[-1], 0.97, 1.03, color="k", lw=5)
plt.scatter(1, quantiles0[1], c="w", zorder=3, s=50)

plt.vlines(2, quantiles1[0], quantiles1[-1], color="k", lw=5)
plt.hlines(quantiles1[0], 1.97, 2.03, color="k", lw=5)
plt.hlines(quantiles1[-1], 1.97, 2.03, color="k", lw=5)
plt.scatter(2, quantiles1[1], c="w", zorder=3, s=50)

plt.xlabel("States", fontsize=24)
plt.ylabel("Proportion of time of occupation of states", fontsize=24)

plt.show();
```

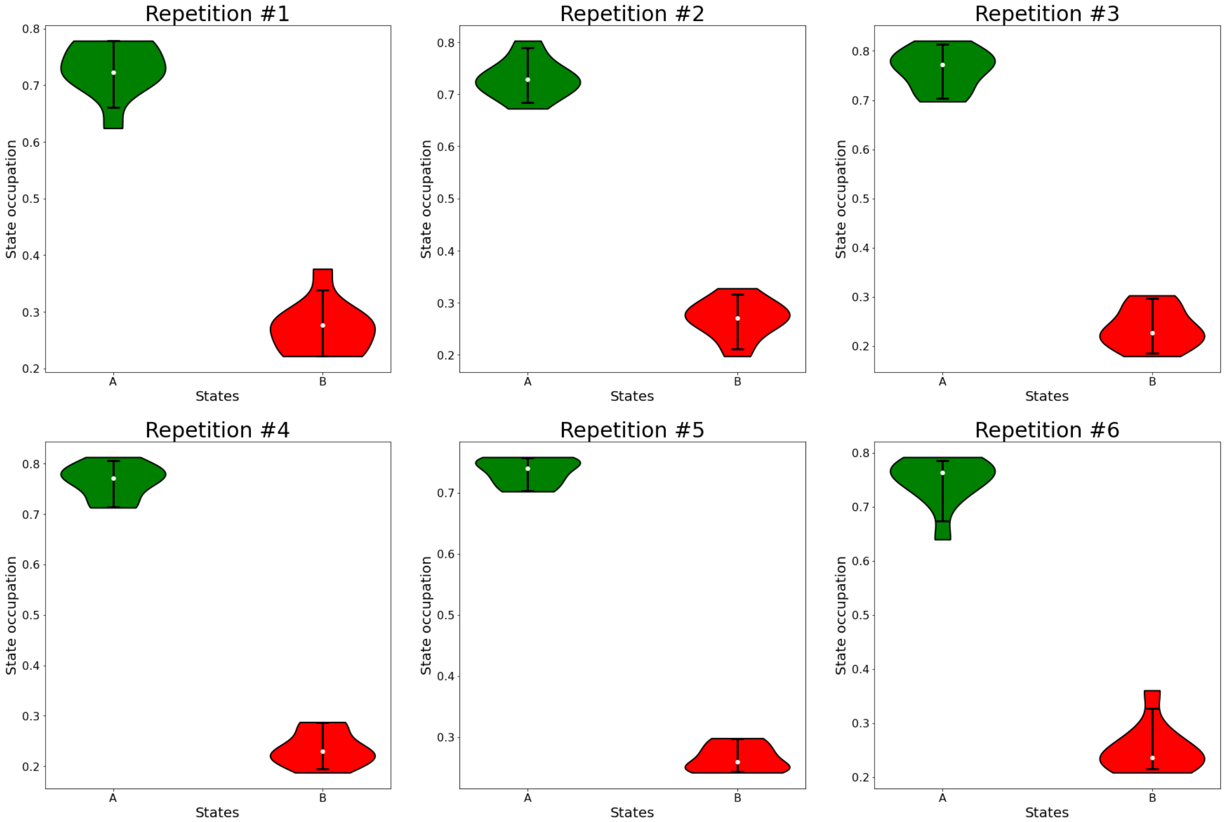


We see that the resulting distributions of state occupations are close to the expected values, but the distributions are rather uneven. Notice that the white points indicate the average, and the bars indicate the condifence interval from 5 to 95 percentile. Lets run this a few more times and look at how it varies.

In [6]:

```
plt.subplots(2, 3, figsize=(30, 20)) # Create subplot grid
for i in range(6): # Loop over repetitions to see possible variation in resul
    realizations = 10

    state_hist = BF2S_hist(T, dt, w_01, w_10, realizations) # Calculate state

    # Plotting and figure editting

    quantiles0 = np.percentile(state_hist[:,0], [5, 50, 95])
    quantiles1 = np.percentile(state_hist[:,1], [5, 50, 95])

    plt.subplot(2, 3, i+1)
    violin = plt.violinplot(state_hist, showextrema=False, showmeans=False)

    violin["bodies"][0].set_facecolor("green")
    violin["bodies"][0].set_edgecolor("k")
    violin["bodies"][0].set_linewidth(2)
    violin["bodies"][0].set_alpha(1)
    violin["bodies"][1].set_facecolor("red")
    violin["bodies"][1].set_edgecolor("k")
    violin["bodies"][1].set_linewidth(2)
    violin["bodies"][1].set_alpha(1)

    plt.xticks([1, 2], labels=["A", "B"], fontsize=16)
    plt.yticks(fontsize=16)
    plt.title(f"Repetition #{i+1}", fontsize=30)
```

```
    plt.vlines(1, quantiles0[0], quantiles0[-1], color="k", lw=3)
    plt.hlines(quantiles0[0], 0.97, 1.03, color="k", lw=3)
    plt.hlines(quantiles0[-1], 0.97, 1.03, color="k", lw=3)
    plt.scatter(1, quantiles0[1], c="w", zorder=3, s=30)

    plt.vlines(2, quantiles1[0], quantiles1[-1], color="k", lw=3)
    plt.hlines(quantiles1[0], 1.97, 2.03, color="k", lw=3)
    plt.hlines(quantiles1[-1], 1.97, 2.03, color="k", lw=3)
    plt.scatter(2, quantiles1[1], c="w", zorder=3, s=30)

    plt.xlabel("States", fontsize=20)
    plt.ylabel("State occupation", fontsize=20)
plt.show(); # Show plot
```

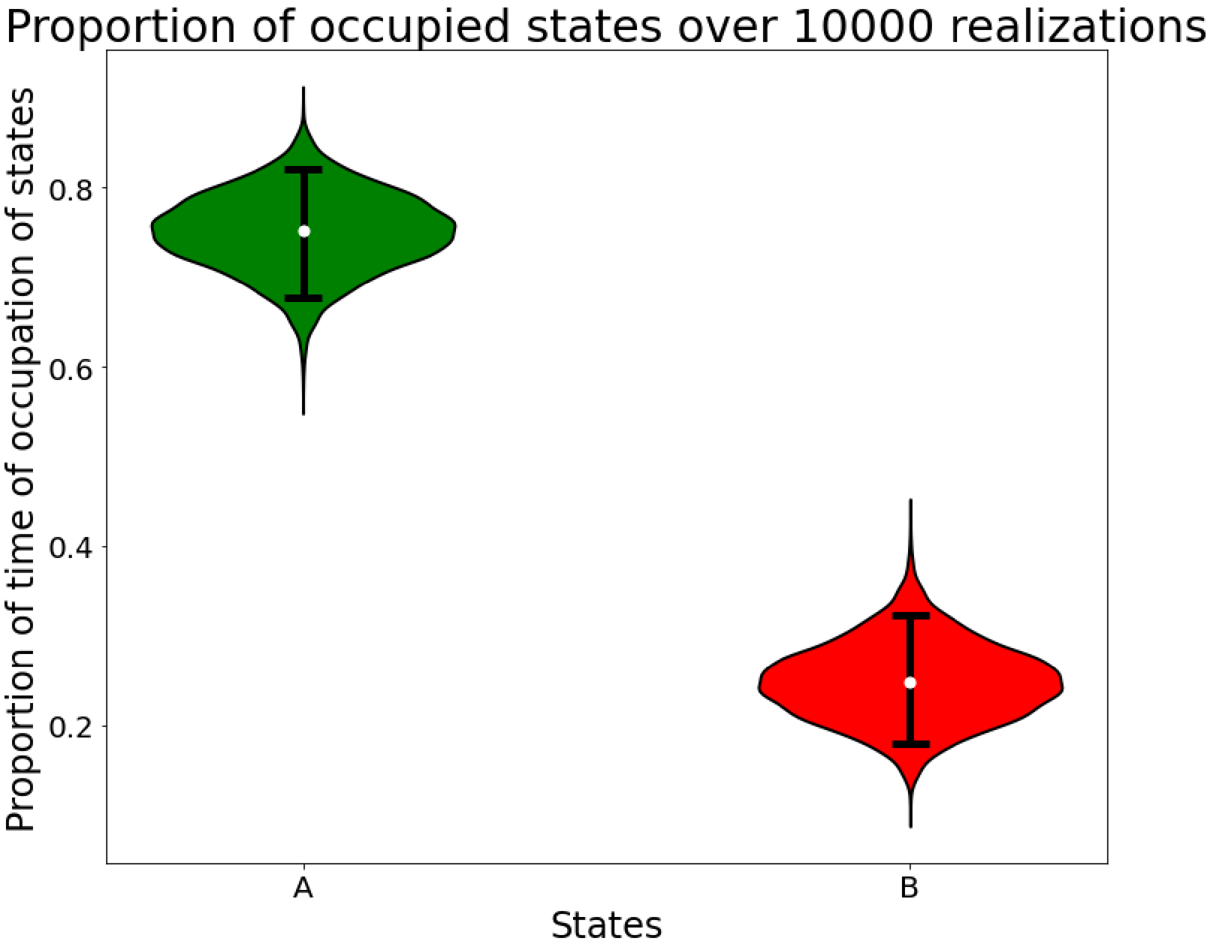


There is a noticeable variation over different repetitions, maybe we can get less variating results by increasing the number of realization of the process we take into account. Let's consider 10^4 realizations:

In [7]:

```
realizations = 10000 # Choose number of realizations to consider

state_hist = BF2S_hist(T, dt, w_01, w_10, realizations) # Generate state occu

# The rest is just plot editting

# Calculate quantiles to show
quantiles0 = np.percentile(state_hist[:,0], [5, 50, 95])
quantiles1 = np.percentile(state_hist[:,1], [5, 50, 95])

plt.figure(figsize=(12,10)) # Create figure

violin = plt.violinplot(state_hist, showextrema=False, showmeans=False) # Ger

# Edit violins
violin["bodies"][0].set_facecolor("green")
violin["bodies"][0].set_edgecolor("k")
violin["bodies"][0].set_linewidth(2)
```

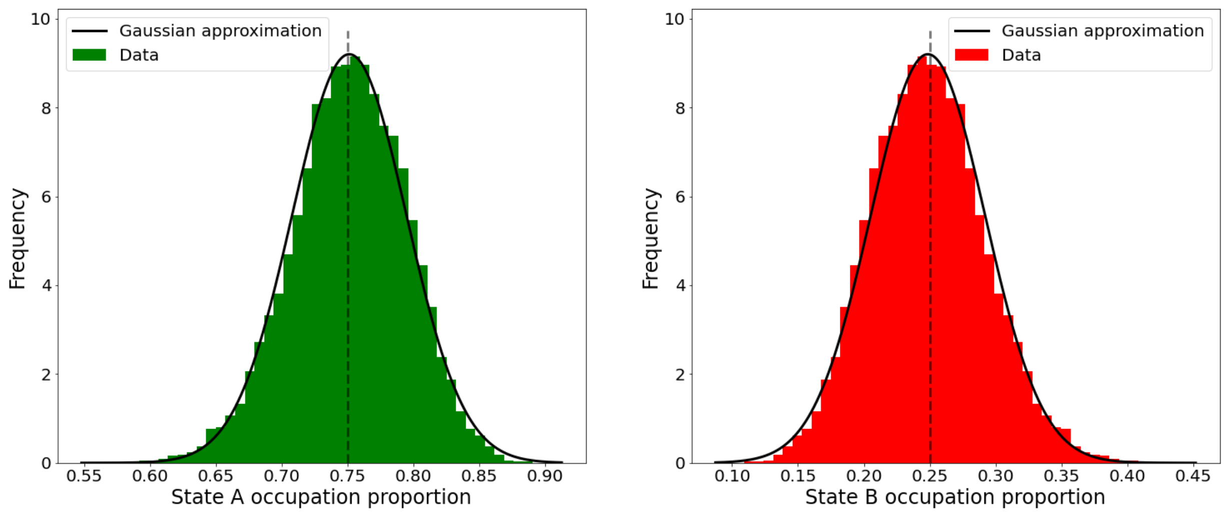
```python
violin["bodies"][0].set_alpha(1)
violin["bodies"][1].set_facecolor("red")
violin["bodies"][1].set_edgecolor("k")
violin["bodies"][1].set_linewidth(2)
violin["bodies"][1].set_alpha(1)

# More plot editing

plt.xticks([1, 2], labels=["A", "B"], fontsize=20)
plt.yticks(fontsize=20)
plt.title(f"Proportion of occupied states over {realizations} realizations",

plt.vlines(1, quantiles0[0], quantiles0[-1], color="k", lw=5)
plt.hlines(quantiles0[0], 0.97, 1.03, color="k", lw=5)
plt.hlines(quantiles0[-1], 0.97, 1.03, color="k", lw=5)
plt.scatter(1, quantiles0[1], c="w", zorder=3, s=50)

plt.vlines(2, quantiles1[0], quantiles1[-1], color="k", lw=5)
plt.hlines(quantiles1[0], 1.97, 2.03, color="k", lw=5)
plt.hlines(quantiles1[-1], 1.97, 2.03, color="k", lw=5)
plt.scatter(2, quantiles1[1], c="w", zorder=3, s=50)

plt.xlabel("States", fontsize=24)
plt.ylabel("Proportion of time of occupation of states", fontsize=24)

plt.show();
```



These results look better, and we can even look at the distribution of occupation for each state, over all realizations:

In [16]:
```python
# Calculate average and standard deviation of occupation of both states to sl
mean0 = np.mean(state_hist[:,0])
std0 = np.std(state_hist[:,0])
```

```
mean1 = np.mean(state_hist[:,1])
std1 = np.std(state_hist[:,1])

plt.subplots(1, 2, figsize=(25, 10)) # Create subplot grid

# Plot for state A
plt.subplot(1, 2, 1)
plt.hist(state_hist[:,0], 50, color="g", density=True, label="Data")
x = np.linspace(np.min(state_hist[:,0]), np.max(state_hist[:,0]), 1000)
plt.plot(x, np.exp(-(x-mean0)**2/(2*std0**2))/np.sqrt(2*np.pi*std0**2), c="k'
plt.vlines(0.75, 0, np.max(np.histogram(state_hist[:,0], density=True)[0])*1.
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("State A occupation proportion", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.legend(fontsize=20)

# Plot for state B
plt.subplot(1, 2, 2)
plt.hist(state_hist[:,1], 50, color="r", density=True, label="Data")
x = np.linspace(np.min(state_hist[:,1]), np.max(state_hist[:,1]), 1000)
plt.plot(x, np.exp(-(x-mean1)**2/(2*std1**2))/np.sqrt(2*np.pi*std1**2), c="k'
plt.vlines(0.25, 0, np.max(np.histogram(state_hist[:,1], density=True)[0])*1.
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("State B occupation proportion", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.legend(fontsize=20)

plt.show() # Show plots
```



We can see that the distribution of occupation of each state over multiple realizations can be nicely approximated by a normal distribution, and their centers are close to the expected values. Let's look at the actual values:

In [18]:

```
print("State A occupation distribution:")
print(f"\tAverage: {mean0}")
print(f"\tStandard Deviation: {std0}")
print("State B occupation distribution:")
print(f"\tAverage: {mean1}")
print(f"\tStandard Deviation: {std1}")
```

```
State A occupation distribution:
        Average: 0.7513724030000001
        Standard Deviation: 0.04336470490428351
State B occupation distribution:
```

```
Average: 0.248627597
Standard Deviation: 0.04336470490428351
```

Waiting time statistics

Now, let's look at the statistics of the time between transitions during a single realization of the process. Let's define Δt_{AB} as the time the system spends on state A before transitioning to B, and Δt_{BA} as the time the system spends on state B before transitioning to A. First, let's define a function to calculate this, still using a brute force integration algorithm.

In [46]:

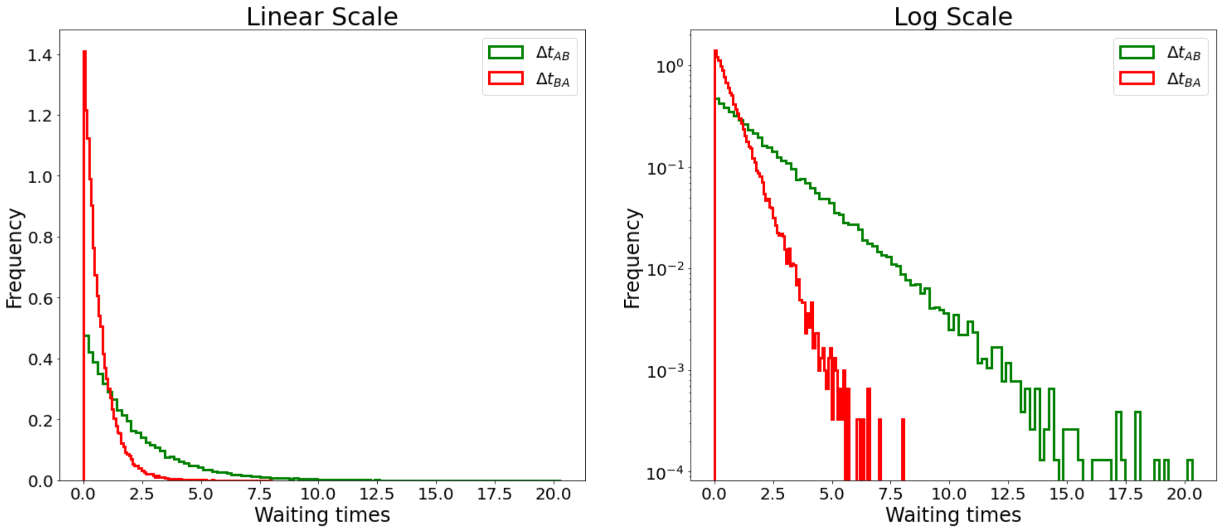
```
# Function to calculate waiting time statistics of the stochastic process
@njit # Numba pre-compilation decorator
def BF2S_transition_time(T, dt, w_01, w_10):
    nt = int(T/dt) # Number of time points

    t01 = [dt] # Array to store waiting times of A to B
    t10 = [] # Array to store waiting times of B to A

    state = 0 # Initial state = A

    # Loop over time
    for n in range(1, nt):
        if state == 0: # If on state A, transition to be with probability \om
            if np.random.random() < w_01*dt:
                state = 1 # Change state
                t10.append(dt) # Count time step
            else:
                t01[-1] += dt # Count time step
        else: # If on state B, transition to be with probability \omega_{BA}*
            if np.random.random() < w_10*dt:
                state = 0 # Change state
                t01.append(dt) # Count time step
            else:
                t10[-1] += dt # Count time step

    return t01, t10 # Return waiting times data
BF2S_transition_time(1, 0.1, 0, 0);
```

In order to have better statistics, let's consider a longer realization of the experiment, for a time of 10^5 t.u.

In [77]:

```
T = 1e5 # Set new duration
t01, t10 = BF2S_transition_time(T, dt, w_01, w_10) # Calculate waiting times

# Plotting ...

plt.subplots(1, 2, figsize=(25,10)) # Create subplot grid

# Plot waiting time histograms on linear scale
plt.subplot(1, 2, 1)
plt.hist(t01, 100, density=True, histtype="step", color="g", lw=3, label="$\D
plt.hist(t10, 100, density=True, histtype="step", color="r", lw=3, label="$\D
plt.legend(fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("Linear Scale", fontsize=30)
```

```
# Plot waiting time histograms on log scale
plt.subplot(1, 2, 2)
plt.hist(t01, 100, density=True, histtype="step", color="g", lw=3, label="$\D
plt.hist(t10, 100, density=True, histtype="step", color="r", lw=3, label="$\D
plt.legend(fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("Log Scale", fontsize=30)
plt.yscale("log")
plt.show();
```



One can see these distribution really behave as exponential, as one would expect. Let's fit them and try to recover the transition rates.

In [51]:

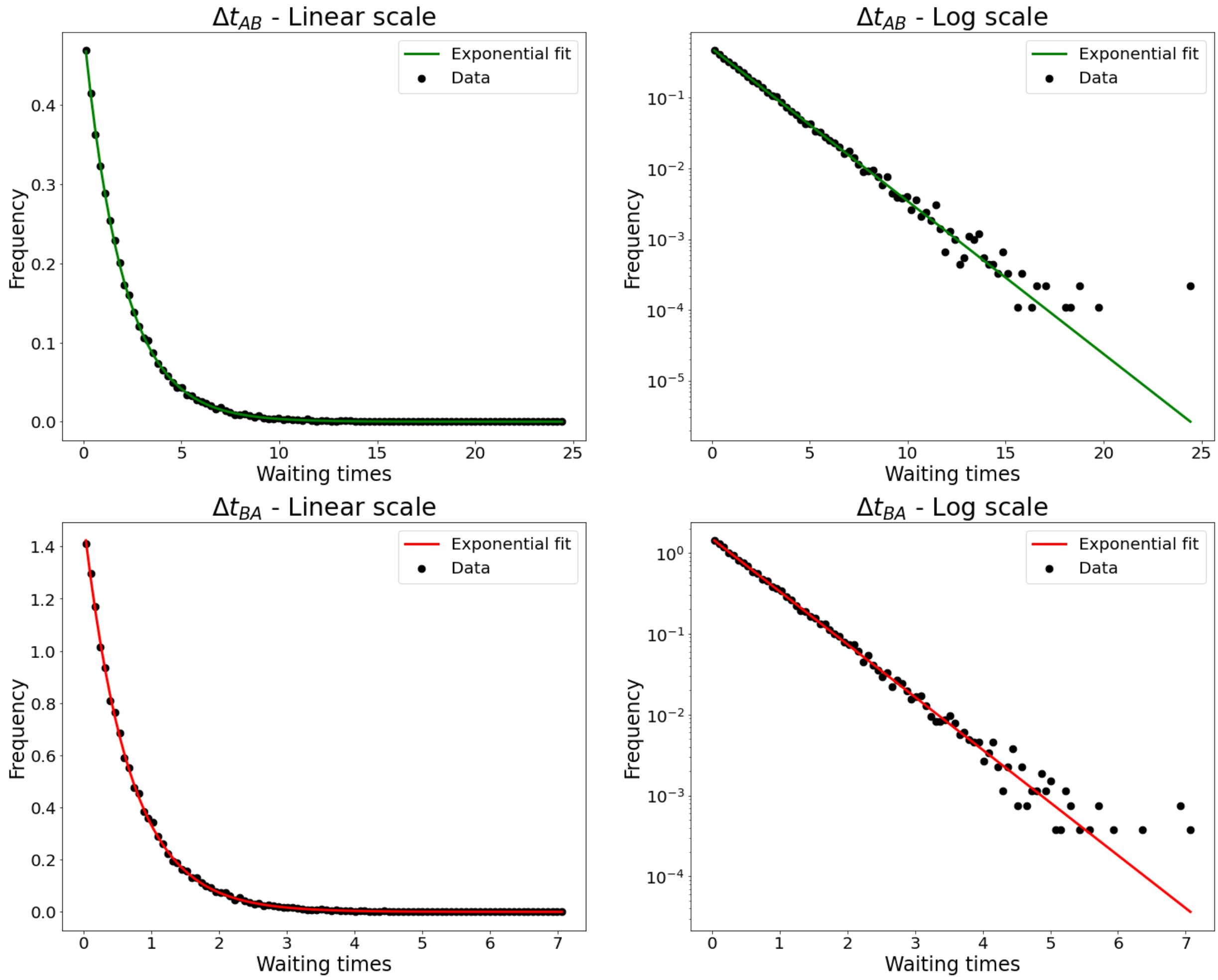
```
# Calculate histogram, and set values to the middle of the bins
# for waiting time of transition A -> B
t01_values, t01_bins = np.histogram(t01, bins=100, density=True)
t01_points = (t01_bins[1:]+t01_bins[:-1])/2

# and for waiting time of transition B -> A
t10_values, t10_bins = np.histogram(t10, bins=100, density=True)
t10_points = (t10_bins[1:]+t10_bins[:-1])/2

# Define exponential PDF to be fit
def exp(x, a):
    return a*np.exp(-a*x)

# Fit exponential PDF and save transition rates
p01 = curve_fit(exp, t01_points, t01_values)[0]
p10 = curve_fit(exp, t10_points, t10_values)[0]

# Print obtained transition rates
print(f"Exponential fit parameter for state A waiting times: w_01 = {p01}")
print(f"Exponential fit parameter for state B waiting times: w_10 = {p10}")

# Plotting...

plt.subplots(2, 2, figsize=(25,20)) # Create grid of subplots

# Plot for waiting time on trantision A -> B, in linear scale
plt.subplot(2, 2, 1)
plt.scatter(t01_points, t01_values, s=75, c="k", label="Data")
plt.plot(t01_points, exp(t01_points, p01[0]), lw=3, c="g", label="Exponential
```

```
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("$\Delta t_{AB}$ - Linear scale", fontsize=30)
plt.legend(fontsize=20)

# Plot for waiting time on trantision A -> B, in log scale
plt.subplot(2, 2, 2)
plt.scatter(t01_points, t01_values, s=75, c="k", label="Data")
plt.plot(t01_points, exp(t01_points, p01[0]), lw=3, c="g", label="Exponential
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("$\Delta t_{AB}$ - Log scale", fontsize=30)
plt.yscale("log")
plt.legend(fontsize=20)

# Plot for waiting time on trantision B -> A, in linear scale
plt.subplot(2, 2, 3)
plt.scatter(t10_points, t10_values, s=75, c="k", label="Data")
plt.plot(t10_points, exp(t10_points, p10[0]), lw=3, c="r", label="Exponential
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("$\Delta t_{BA}$ - Linear scale", fontsize=30)
plt.legend(fontsize=20)

# Plot for waiting time on trantision B -> A, in log scale
plt.subplot(2, 2, 4)
plt.scatter(t10_points, t10_values, s=75, c="k", label="Data")
plt.plot(t10_points, exp(t10_points, p10[0]), lw=3, c="r", label="Exponential
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel("Waiting times", fontsize=24)
plt.ylabel("Frequency", fontsize=24)
plt.title("$\Delta t_{BA}$ - Log scale", fontsize=30)
plt.yscale("log")
plt.legend(fontsize=20)
```

```
Exponential fit parameter for state A waiting times: w_01 = [0.4973042]
Exponential fit parameter for state B waiting times: w_10 = [1.50326923]
```

Out[51]:

```
<matplotlib.legend.Legend at 0x7f37594a4b20>
```



The transition rates obtained from the fit $\omega_{AB} = 0.497$ and $\omega_{BA} = 1.503$ are in good agreement with the actual transition rates we set for the process.

First Reaction Method

Let's now implement the same stochastic process, but using a different, optimized algorithm: the first reaction method. It's implementation is below:

In [23]:

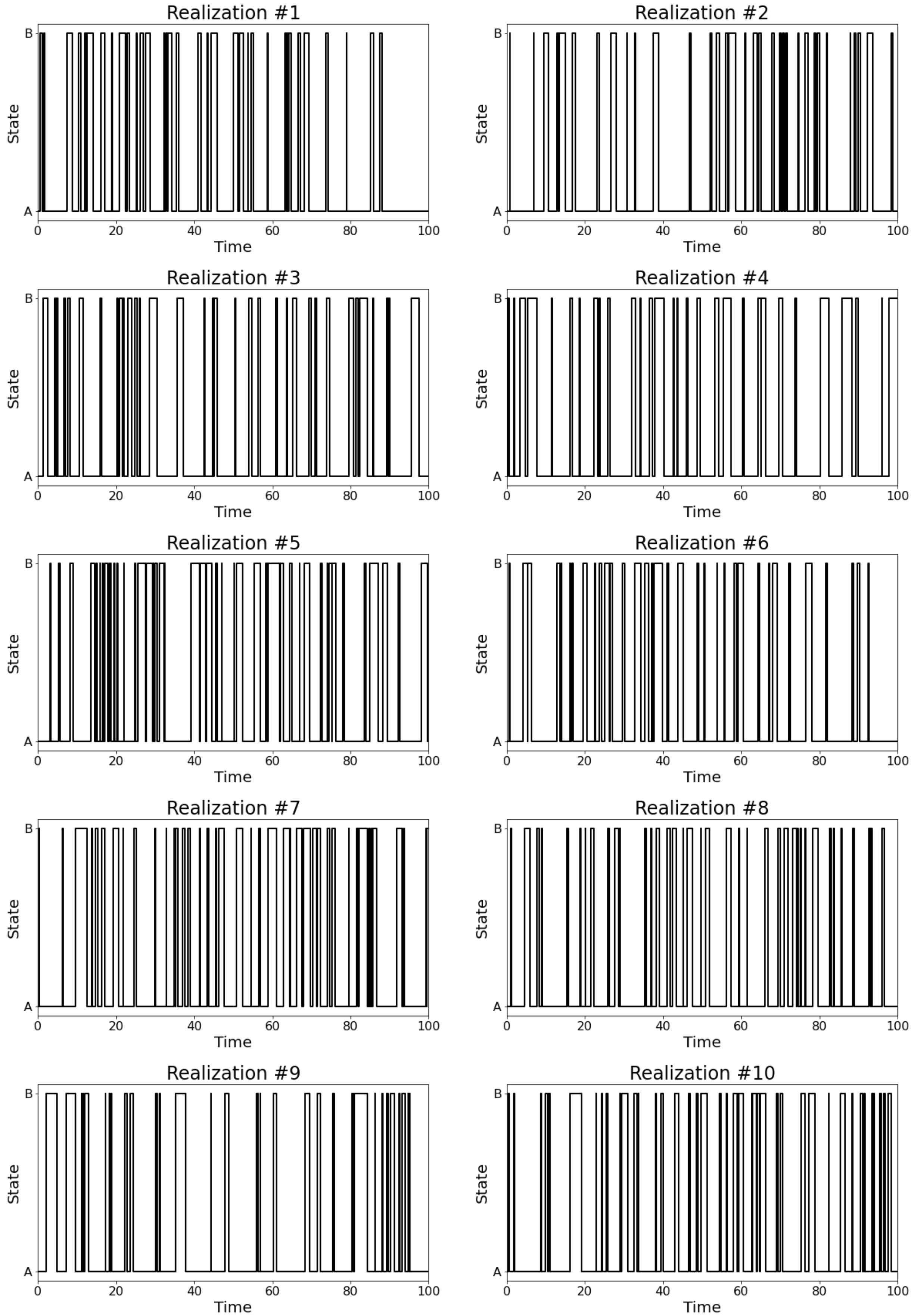
```
# Time evolution, with a First Reaction implementation for a Two State stocha
@njit
def FR2S_evol(T, w_01, w_10):
    t = [0] # Array to store time
    state = [0] # Array to store state through time

    while t[-1] < T: # Runs until the end of set duration
        if state[-1] == 0: # If state == A, set transtion to happen in t + a
            t.append(t[-1]+np.random.exponential(1/w_01)) # Update time
            state.append(1) # Change state
        else: # If state == B, set transtion to happen in t + a step derived
            t.append(t[-1]+np.random.exponential(1/w_10)) # Update time
            state.append(0) # Change state

    return np.array(t), np.array(state) # Return time and state arrays
FR2S_evol(0, 0, 0);

# Generation of state occupation time statistics, over multiple realizations
@njit
def FR2S_hist(T, w_01, w_10, realizations):
    state_hist = np.zeros((realizations, 2)) # Array to store state occupatic
```

```python
    # Loop over realizations
    for i in range(realizations):
        t = 0 # Set initial time
        state = 0 # Set initial state

        while t < T: # Runs until the end of set duration
            if state == 0: # If state == A, set transtion to happen in t + a
                dt = np.random.exponential(1/w_01) # Calculates step size
                t += dt # Update time
                state = 1 # Change state
                state_hist[i,0] += dt # Count time spent in A
            else: # If state == B, set transtion to happen in t + a step der
                dt = np.random.exponential(1/w_10) # Calculate step size
                t += dt # Update time
                state = 0 # Change state
                state_hist[i,1] += dt # Count time spent in B

        state_hist[i] /= np.sum(state_hist[i])

    return state_hist # Return state occupation time statistics
FR2S_hist(0, 0, 0, 1);
```

We can now compare the realizations to the ones we've seen generated via brute force:

In [24]:

```python
T = 100 # Set duration back to 100 t.u.

# Run and plot realizations

plt.subplots(5, 2, figsize=(20, 30)) # Create subplot grid
plt.subplots_adjust(hspace=0.35)
for i in range(10):
    plt.subplot(5, 2, i+1)

    t, s = FR2S_evol(T, w_01, w_10) # Generate realizations

    plt.step(t, s, where="post", c="k", lw=2)
    plt.xlabel("Time", fontsize=20)
    plt.ylabel("State", fontsize=20)
    plt.xticks(fontsize=16)
    plt.yticks([0, 1], labels=["A", "B"], fontsize=16)
    plt.title(f"Realization #{i+1}", fontsize=24)
    plt.xlim(0, T)
plt.show();
```

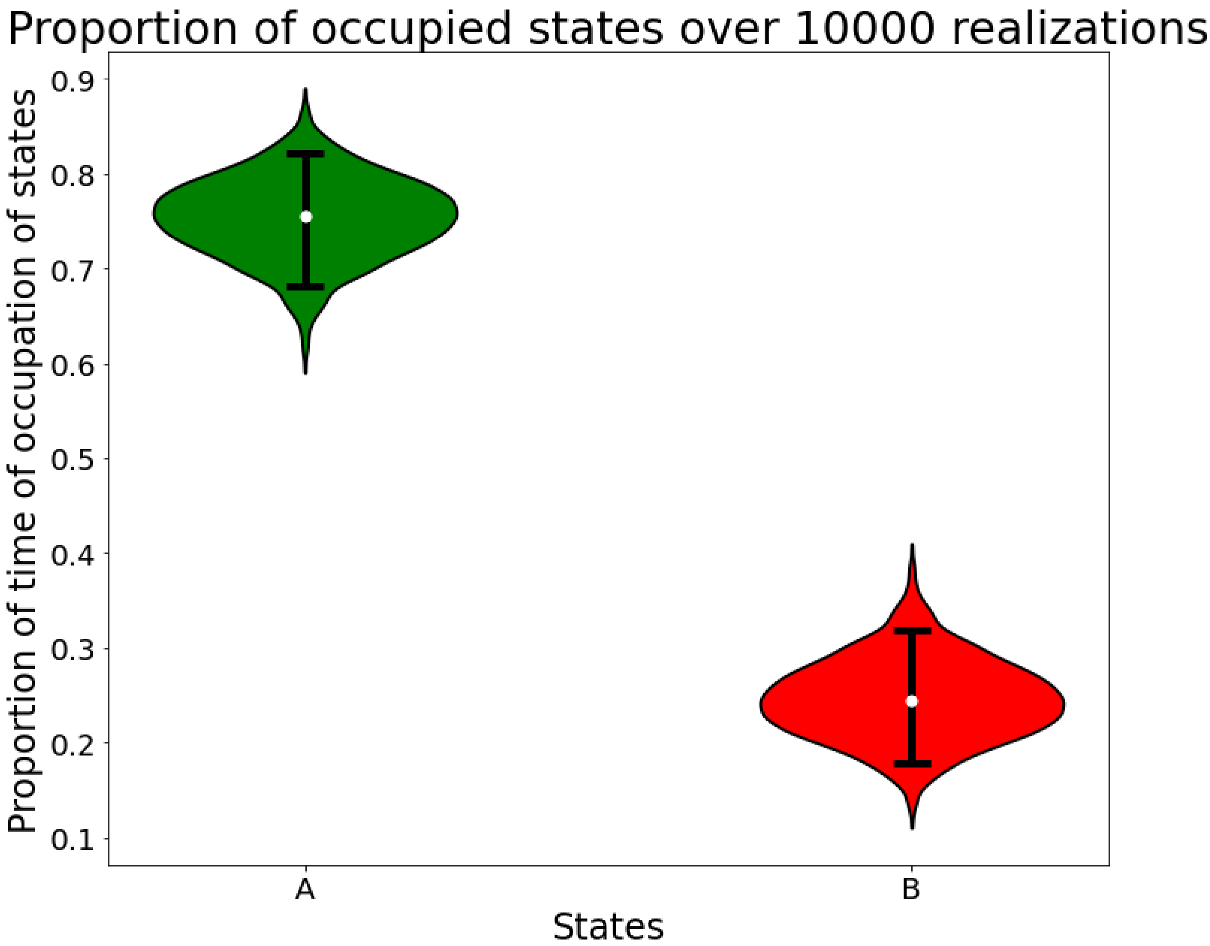


The realizations look the same, but what about the state occupation statistics?

In [25]:

```
realizations = 10000 # Choose number of realizations to consider

state_hist = FR2S_hist(T, w_01, w_10, realizations) # Generate state occupati

# The rest is just plot editting
```

```
# Calculate quantiles to show
quantiles0 = np.percentile(state_hist[:,0], [5, 50, 95])
quantiles1 = np.percentile(state_hist[:,1], [5, 50, 95])

plt.figure(figsize=(12,10)) # Create figure

violin = plt.violinplot(state_hist, showextrema=False, showmeans=False) # Ger

# Edit violins
violin["bodies"][0].set_facecolor("green")
violin["bodies"][0].set_edgecolor("k")
violin["bodies"][0].set_linewidth(2)
violin["bodies"][0].set_alpha(1)
violin["bodies"][1].set_facecolor("red")
violin["bodies"][1].set_edgecolor("k")
violin["bodies"][1].set_linewidth(2)
violin["bodies"][1].set_alpha(1)

# More plot editing

plt.xticks([1, 2], labels=["A", "B"], fontsize=20)
plt.yticks(fontsize=20)
plt.title(f"Proportion of occupied states over {realizations} realizations",

plt.vlines(1, quantiles0[0], quantiles0[-1], color="k", lw=5)
plt.hlines(quantiles0[0], 0.97, 1.03, color="k", lw=5)
plt.hlines(quantiles0[-1], 0.97, 1.03, color="k", lw=5)
plt.scatter(1, quantiles0[1], c="w", zorder=3, s=50)

plt.vlines(2, quantiles1[0], quantiles1[-1], color="k", lw=5)
plt.hlines(quantiles1[0], 1.97, 2.03, color="k", lw=5)
plt.hlines(quantiles1[-1], 1.97, 2.03, color="k", lw=5)
plt.scatter(2, quantiles1[1], c="w", zorder=3, s=50)

plt.xlabel("States", fontsize=24)
plt.ylabel("Proportion of time of occupation of states", fontsize=24)

plt.show();
```



The state occupation statistics over multiple runs also looks the same.

Now, let's compare the performance of the first reaction method to the one of the brute force algorithm. Let's generate 10 realization with each algorithm and check how long it takes, and let's repeat it five times, to check if there's any fluctuation.

In [57]:

```
# Run for pre-compilation, just in case it was already deleted from cache
BF2S_evol(T, dt, w_01, w_10)
FR2S_evol(T, w_01, w_10)

for i in range(5):
    t = time.time()
    for i in range(10):
        BF2S_evol(T, dt, w_01, w_10)
    bf_time = time.time()-t
    print(f"Time taken by brute force algorithm for 10 realizations: {bft_tim

    t = time.time()
    for i in range(10):
        FR2S_evol(T, w_01, w_10)
    fr_time = time.time()-t
    print(f"Time taken by first reaction method for 10 realizations: {fr_time

    print(f"\nFirst reaction method is {bf_time/fr_time:.3f} times faster tha

    print("\n"+"-"*100+"\n")
```

```
Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.000157

First reaction method is 70.368 times faster than brute force algorithm.

----------------------------------------------------------------------------
```

```
-----------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.000115

First reaction method is 113.795 times faster than brute force algorithm.

-----------------------------------------------------------------------------
-----------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.000113

First reaction method is 168.292 times faster than brute force algorithm.

-----------------------------------------------------------------------------
-----------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.000116

First reaction method is 106.154 times faster than brute force algorithm.

-----------------------------------------------------------------------------
-----------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.000113

First reaction method is 124.553 times faster than brute force algorithm.

-----------------------------------------------------------------------------
-----------------------
```

The first reaction method is clearly faster, but there's some fluctuation to how much. Let's increase the number of realizations to 10^3

In [59]:

```
# Run for pre-compilation, just in case it was already deleted from cache
BF2S_evol(T, dt, w_01, w_10)
FR2S_evol(T, w_01, w_10)

for i in range(5):
    t = time.time()
    for i in range(1000):
        BF2S_evol(T, dt, w_01, w_10)
    bf_time = time.time()-t
    print(f"Time taken by brute force algorithm for 10 realizations: {bft_tim

    t = time.time()
    for i in range(1000):
        FR2S_evol(T, w_01, w_10)
    fr_time = time.time()-t
    print(f"Time taken by first reaction method for 10 realizations: {fr_time

    print(f"\nFirst reaction method is {bf_time/fr_time:.3f} times faster tha

    print("\n"+"-"*100+"\n")
```

```
Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.007477

First reaction method is 147.204 times faster than brute force algorithm.

-----------------------------------------------------------------------------
-----------------------
```

```
Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.007295

First reaction method is 148.521 times faster than brute force algorithm.

------------------------------------------------------------------------------------------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.007411

First reaction method is 146.967 times faster than brute force algorithm.

------------------------------------------------------------------------------------------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.007718

First reaction method is 140.955 times faster than brute force algorithm.

------------------------------------------------------------------------------------------------------

Time taken by brute force algorithm for 10 realizations: 0.016721
Time taken by first reaction method for 10 realizations: 0.007365

First reaction method is 148.462 times faster than brute force algorithm.

------------------------------------------------------------------------------------------------------
```

The results seem to have stabilized, and we can see that the first reaction method is more than 100 times faster than the brute force algorithm.

Problem 2

Analytic solution for the average

Now we are going to consider another two-state stochastic process, with the same transition rates $\omega_{AB} = 0.5$ and $\omega_{BA} = 1.5$, but now with $N = 1000$ particles instead of only one, with an initial condition of 500 particles in state A, and another 500 in B

First, let's calculate analytically the time evolution of the average number of particles in A. If we have n particles in A, we know that the transition rate to $n-1$ should be $\Omega(n \to n-1) = n\omega_{AB}$, and the rate from n to $n+1$ is $\Omega(n \to n+1) = (N-n)\omega_{BA}$. We can then write the master equation for this process:

$$\frac{\mathrm{d}P(n,t)}{\mathrm{d}t} = (E-1)\left[\omega_{AB} n P(n,t)\right] + \left(E^{-1}-1\right)\left[\omega_{BA}(N-n)P(n,t)\right], \qquad (1)$$

where the operator E is such that $E^m f(n) = f(n+m)$.

Let's calculate the average value $\langle n(t) \rangle$, by multiplying the master equation by n, and summing for $n = 0, 1, 2, \ldots, N$:

$$\sum_{n=0}^{N} n\frac{\mathrm{d}P(n,t)}{\mathrm{d}t} = \sum_{n=0}^{N} \omega_{AB} n(n+1)P(n+1,t) + \sum_{n=0}^{N} \omega_{BA} n(N-n+1)P(n-1,t) + \\ - \sum_{n=0}^{N} \left(\omega_{AB} n^2 + \omega_{BA} n(N-n)\right) P(n,t).$$

This can be rewritten as:

$$\frac{\mathrm{d}}{\mathrm{d}t}\sum_{n=0}^{N} nP(n,t) = \sum_{n=0}^{N} \omega_{AB} n(n-1)P(n,t) + \sum_{n=0}^{N} \omega_{BA} (n+1)(N-n)P(n,t) + \quad (4)$$

$$- \sum_{n=0}^{N} \left(\omega_{AB} n^2 + \omega_{BA} n(N-n)\right) P(n,t), \quad (5)$$

and simplified to

$$\frac{\mathrm{d}}{\mathrm{d}t}\langle n(t)\rangle = \sum_{n=0}^{N} \left[-\omega_{AB} n + \omega_{BA}(N-n)\right] P(n,t). \quad (6)$$

Finally, we get the differential equation for the average number of particles in A:

$$\frac{\mathrm{d}}{\mathrm{d}t}\langle n(t)\rangle = \omega_{BA} N - \omega \langle n(t)\rangle, \quad (7)$$

with $\omega = \omega_{AB} + \omega_{BA}$. To solve this equation, we can define $x(t) = \langle n(t)\rangle - \frac{\omega_{BA}}{\omega}N$, such that:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\omega x \Rightarrow x(t) = Ae^{-\omega t} \quad (8)$$

and the solution for the number of particles must then be:

$$\langle n(t)\rangle = \frac{\omega_{BA}}{\omega} N \left(1 - e^{-\omega t}\right) + \langle n(0)\rangle e^{-\omega t}. \quad (9)$$

Simulation

New we are going so simulate this stochastic process, again using the first reaction method, with a little difference form before, as we are now considering multiple particles. Let's write the function for the time evolution of the system:

In [61]:

```
# First Reaction method for simulation of Multiple Particles in a two-state s
@njit # Pre-compilation decorator
def FRMP2S_evol(T, n, initial_ratio, w_01, w_10):
    t = [0] # Array to store time
    np0 = [int(n*initial_ratio)] # Array to store the number of particles in

    # Loop over time
    while t[-1] < T:

        weight_01 = (np0[-1]*w_01)/((np0[-1]*w_01)+((n-np0[-1])*w_10)) # Calc
        w = np0[-1]*w_01 + (n-np0[-1])*w_10 # Sum of rates
```

```
        t.append(t[-1] + np.random.exponential(1/w)) # Update time

        if np.random.random() < weight_01: # Choose which transition will hap
            np0.append(np0[-1]-1) # Move one particle from A to B
        else:
            np0.append(np0[-1]+1) # Move one particle from B to A

    return np.array(t), np.array(np0) # Return time and number of particles e
FRMP2S_evol(0, 1, 1, 1, 1);
```

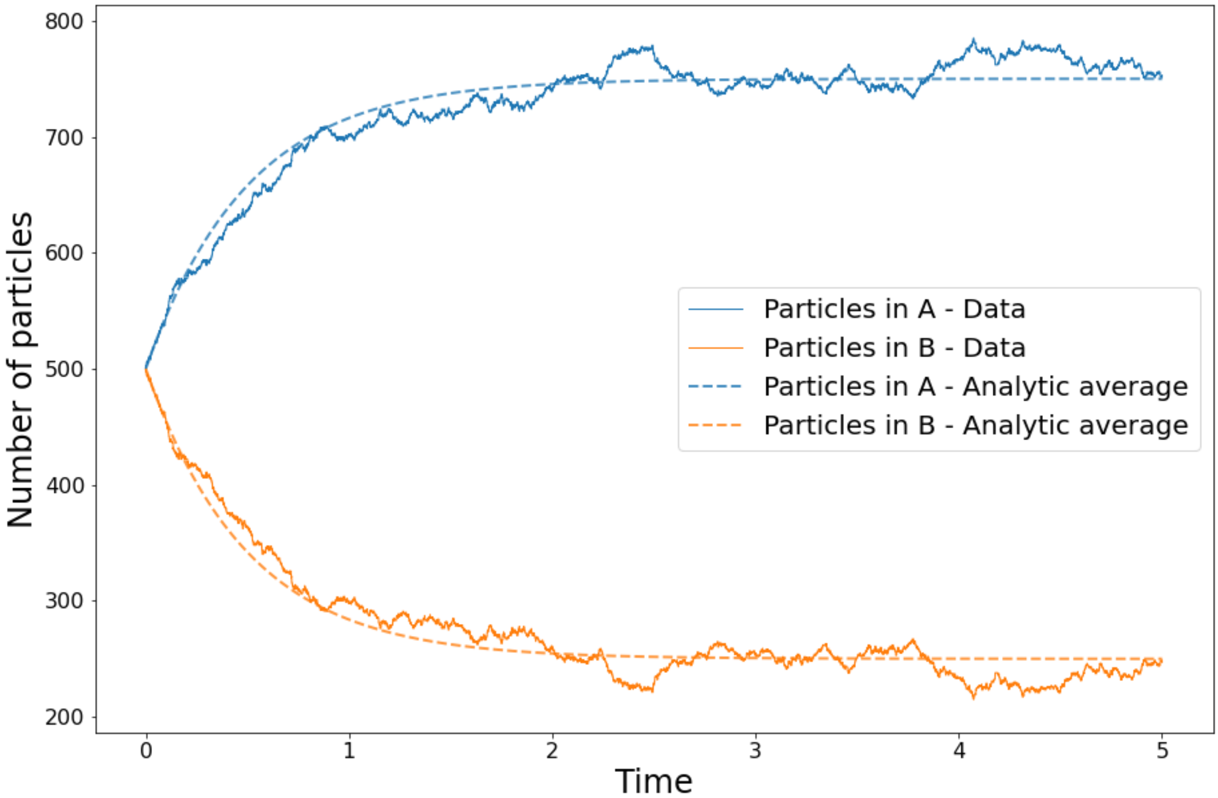
Let's give it a try, and compare the results to the analytically derived expression for the average:

In [64]:

```
# Simulation parameters
T = 5 # Time to consider
n = 1000 # Total number of particles
w_01 = 0.5 # \omega_{AB}
w_10 = 1.5 # \omega_{BA}
initial_ratio = 0.5 # Proportion of particles initially in state A

t, np0 = FRMP2S_evol(T, n, initial_ratio, w_01, w_10) # Run stochastic proces
np1 = n-np0 # Calculate number of particles in state B
w = w_01 + w_10 # Sum of rates
np0_analytic = w_10/w*n*(1-np.exp(-w*t)) + n*initial_ratio*np.exp(-w*t) # Ana
np1_analytic = n - np0_analytic # Analytical average number of particles in B

# Plotting...

plt.figure(figsize=(15,10))
plt.step(t, np0, where="post", lw=1, label="Particles in A - Data")
plt.step(t, np1, where="post", lw=1, label="Particles in B - Data")

plt.plot(t, np0_analytic, c="C0", ls="--", lw=2, alpha=0.8, label="Particles
plt.plot(t, np1_analytic, c="C1", ls="--", lw=2, alpha=0.8, label="Particles

plt.legend(fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xlabel("Time", fontsize=24)
plt.ylabel("Number of particles", fontsize=24)
```

Out[64]:

```
Text(0, 0.5, 'Number of particles')
```



As we can see, the stochastic process fluctuates aroung the expected average. But let's calculate the average from an ensemble of simulations, and see how it relates to our analytical expectation. As the time points of each execution are stochastic, we will interpolate the results to a same time mesh, so that we can calculate averages. We will take the average over 100 realizations of our experiment, and plot the average, together with the analytical solution, and five of the realizations themselves so we can have an idea of the fluctuations.

In [75]:

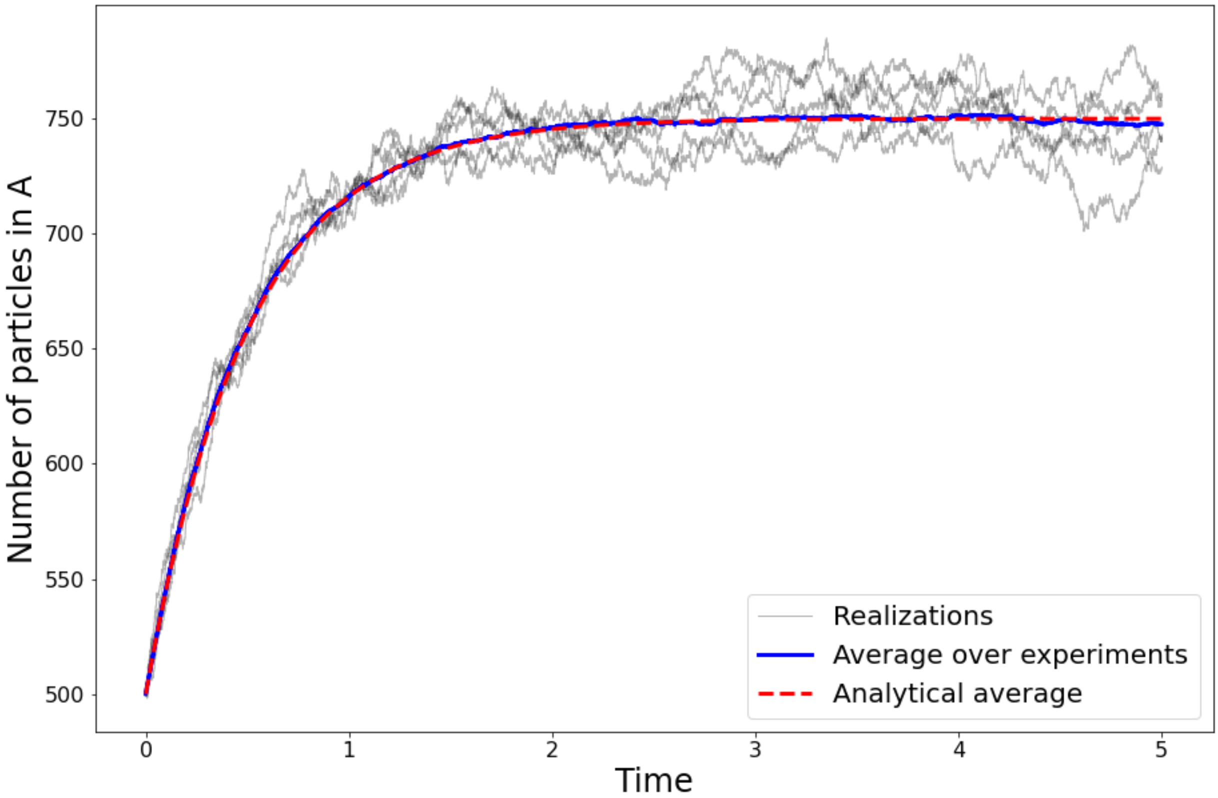
```
t_ = np.linspace(0, 5, 10000) # Time points to which we will interpolate resu
np0_ = np.zeros(10000) # Array to store results
realizations = 100 # Number of realizations to take the average of

n_plot = 5 # Number of realizations to plot

# Running and plotting

plt.figure(figsize=(15,10)) # Create figure

# Loop over realizations
for i in tqdm(range(realizations)):
    t, np0 = FRMP2S_evol(T, n, initial_ratio, w_01, w_10) # Generate realizat
    np0_ += np.interp(t_, t, np0) # Add the actual realization the take the a

    # Plot some of the executions
    if i < n_plot:
        if i == 0:
            plt.step(t, np0, where="post", lw=1, c="k", alpha=0.3, label="Rea
        else:
            plt.step(t, np0, where="post", lw=1, c="k", alpha=0.3)

# Take the average over realizations
np0_ /= realizations

# Plotting

plt.step(t_, np0_, where="post", lw=3, c="b", label="Average over experiments
```

```
np0_analytic = w_10/w*n*(1-np.exp(-w*t_)) + n*initial_ratio*np.exp(-w*t_)
plt.plot(t_, np0_analytic, lw=3, ls="--", c="r", label="Analytical average")

plt.legend(fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xlabel("Time", fontsize=24)
plt.ylabel("Number of particles in A", fontsize=24)
```

```
100%|██████████| 100/100 [00:00<00:00, 1106.05it/s]
```

Out[75]:

```
Text(0, 0.5, 'Number of particles in A')
```



We can see a very good correspondence between the numerical and the analytical averages, as we should expect. We can also look at all of the realizations, forming a "cloud" of curves around the average.

In [76]:

```
t_ = np.linspace(0, 5, 10000) # Time points to which we will interpolate resu
np0_ = np.zeros(10000) # Array to store results
realizations = 100 # Number of realizations to take the average of

n_plot = 100 # Number of realizations to plot

# Running and plotting

plt.figure(figsize=(15,10)) # Create figure

# Loop over realizations
for i in tqdm(range(realizations)):
    t, np0 = FRMP2S_evol(T, n, initial_ratio, w_01, w_10) # Generate realizat
    np0_ += np.interp(t_, t, np0) # Add the actual realization the take the a

    # Plot some of the executions
    if i < n_plot:
        if i == 0:
            plt.step(t, np0, where="post", lw=1, c="k", alpha=0.3, label="Rea
        else:
            plt.step(t, np0, where="post", lw=1, c="k", alpha=0.3)
```

```python
# Take the average over realizations
np0_ /= realizations

# Plotting

plt.step(t_, np0_, where="post", lw=3, c="b", label="Average over experiments

np0_analytic = w_10/w*n*(1-np.exp(-w*t_)) + n*initial_ratio*np.exp(-w*t_)
plt.plot(t_, np0_analytic, lw=3, ls="--", c="r", label="Analytical average")

plt.legend(fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xlabel("Time", fontsize=24)
plt.ylabel("Number of particles in A", fontsize=24)
```

```
100%|██████████| 100/100 [00:00<00:00, 526.57it/s]
```

Out[76]:

```
Text(0, 0.5, 'Number of particles in A')
```