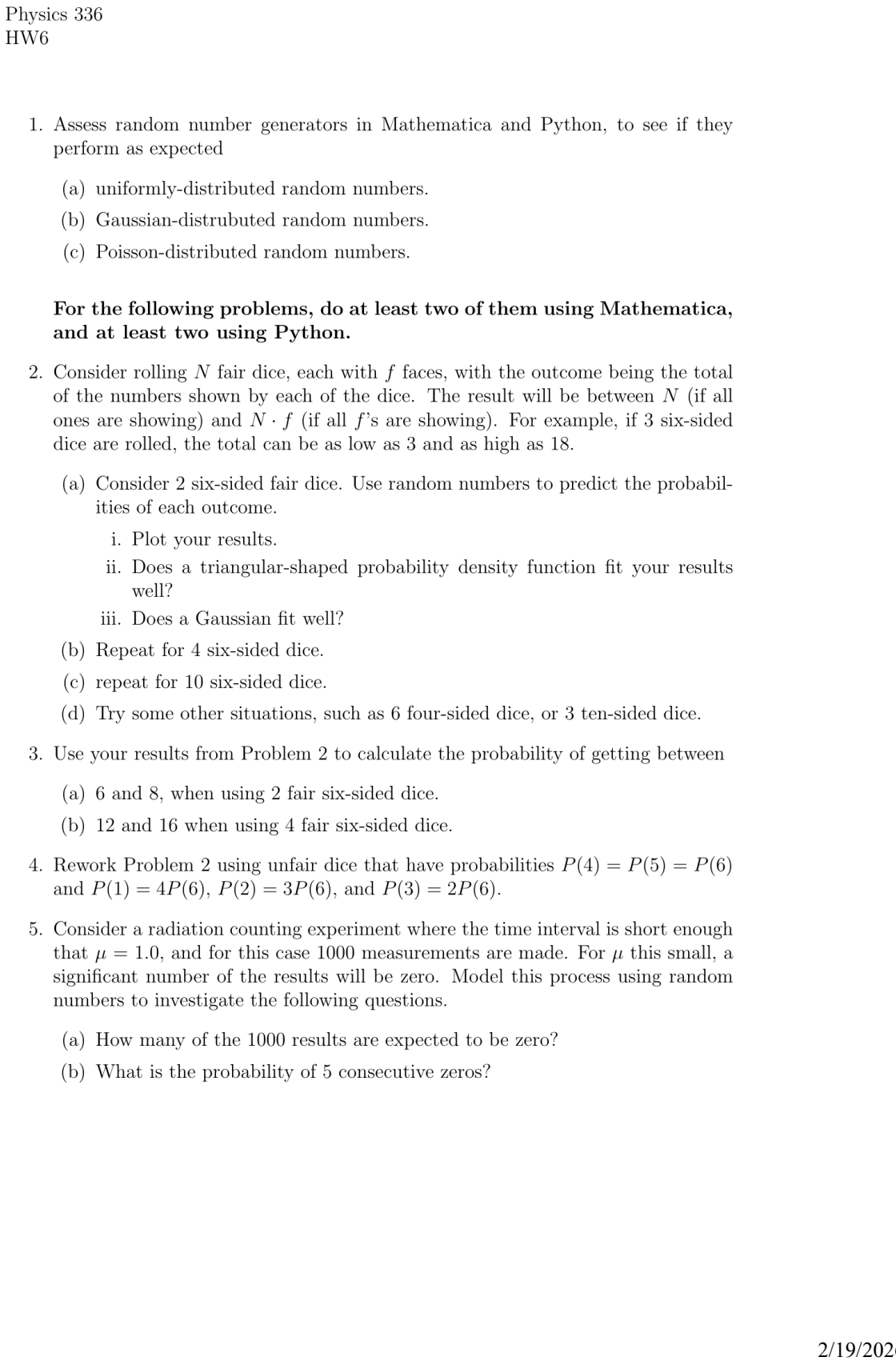
In [57]:
```
from pdf2image import convert_from_path
from IPython.display import display

images = convert_from_path(r"C:\Users\miles\BYU-Idaho\11th Semester\336 Advanced Lab\336-hw6.pdf")
for img in images:
    display(img)
```



Physics 336
HW6

1. Assess random number generators in Mathematica and Python, to see if they perform as expected
   (a) uniformly-distributed random numbers.
   (b) Gaussian-distrubuted random numbers.
   (c) Poisson-distributed random numbers.

   For the following problems, do at least two of them using Mathematica, and at least two using Python.

2. Consider rolling N fair dice, each with f faces, with the outcome being the total of the numbers shown by each of the dice. The result will be between N (if all ones are showing) and $N \cdot f$ (if all f's are showing). For example, if 3 six-sided dice are rolled, the total can be as low as 3 and as high as 18.
   (a) Consider 2 six-sided fair dice. Use random numbers to predict the probabilities of each outcome.
      i. Plot your results.
      ii. Does a triangular-shaped probability density function fit your results well?
      iii. Does a Gaussian fit well?
   (b) Repeat for 4 six-sided dice.
   (c) repeat for 10 six-sided dice.
   (d) Try some other situations, such as 6 four-sided dice, or 3 ten-sided dice.

3. Use your results from Problem 2 to calculate the probability of getting between
   (a) 6 and 8, when using 2 fair six-sided dice.
   (b) 12 and 16 when using 4 fair six-sided dice.

4. Rework Problem 2 using unfair dice that have probabilities $P(4) = P(5) = P(6)$ and $P(1) = 4P(6)$, $P(2) = 3P(6)$, and $P(3) = 2P(6)$.

5. Consider a radiation counting experiment where the time interval is short enough that $\mu = 1.0$, and for this case 1000 measurements are made. For μ this small, a significant number of the results will be zero. Model this process using random numbers to investigate the following questions.
   (a) How many of the 1000 results are expected to be zero?
   (b) What is the probability of 5 consecutive zeros?

(c) How many sequences of at least 5 consecutive zero results are expected?

(d) How many 1's are expected? Do the number of 1's follow a Gaussian distribution?

6. Simulate an ideal gas in a container of volume V_{total}, and determine the fraction of atoms that are within some subvolume $V < V_{total}$

   (a) in one dimension.

   (b) in two dimensions.

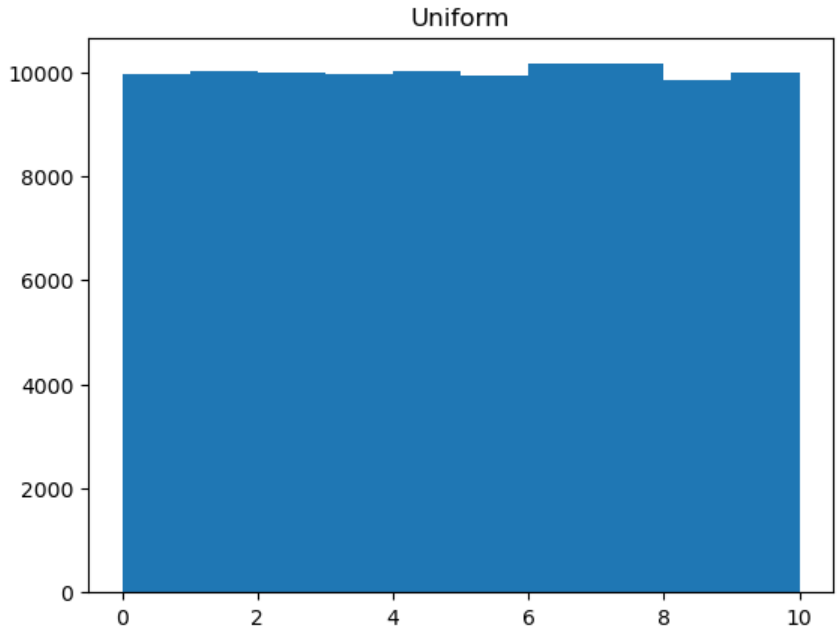
   (c) in three dimensions.

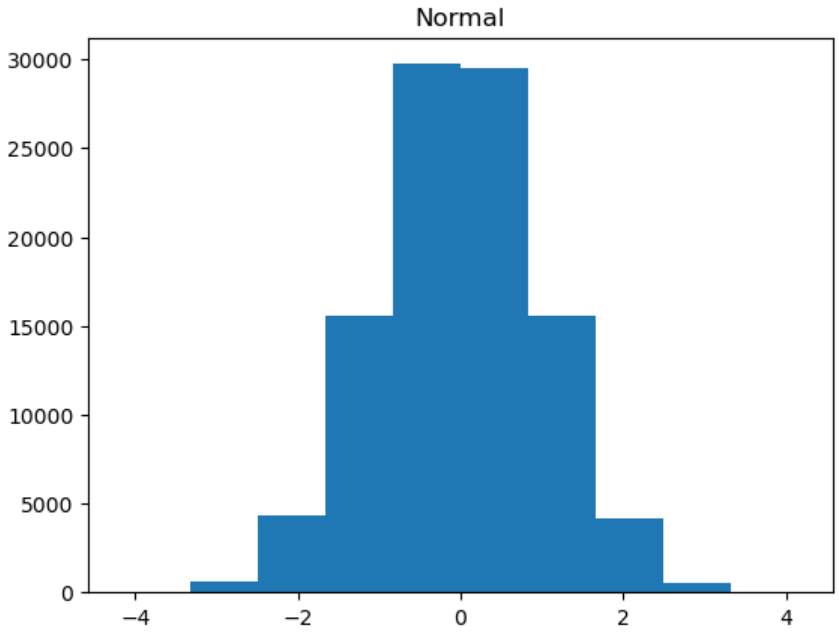
   (d) Comment on your results.

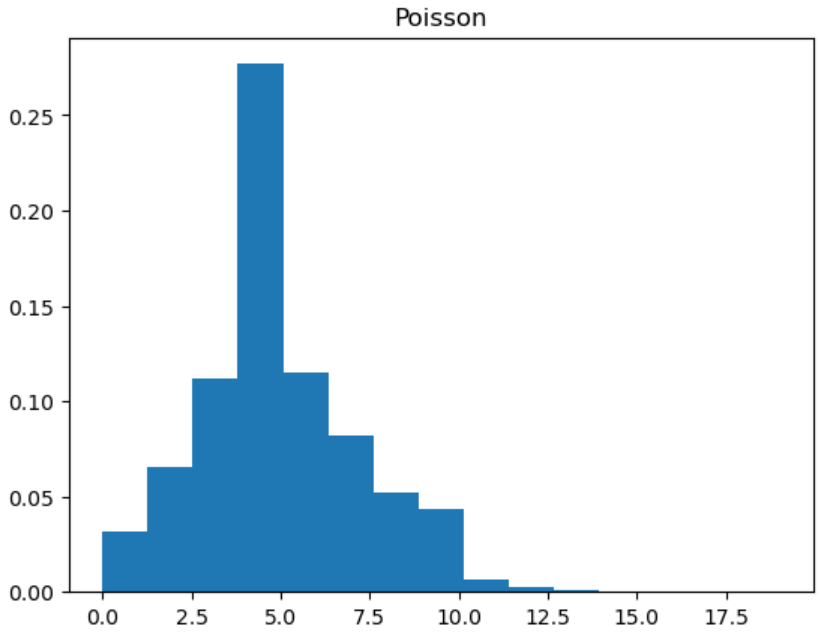
7. Invent a situation and generate some simulated data for it.

A, B and C for Python:

In [20]:
```
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson

N=100_000

uni=uniform(0,10,N)
gauss=randn(N)
poiss=poisson(5,N)

plt.hist(uni)
plt.title("Uniform")
plt.show()

plt.hist(gauss)
plt.title("Normal")
plt.show()

plt.hist(poiss,15,density=True)
plt.title("Poisson")
plt.show()
```



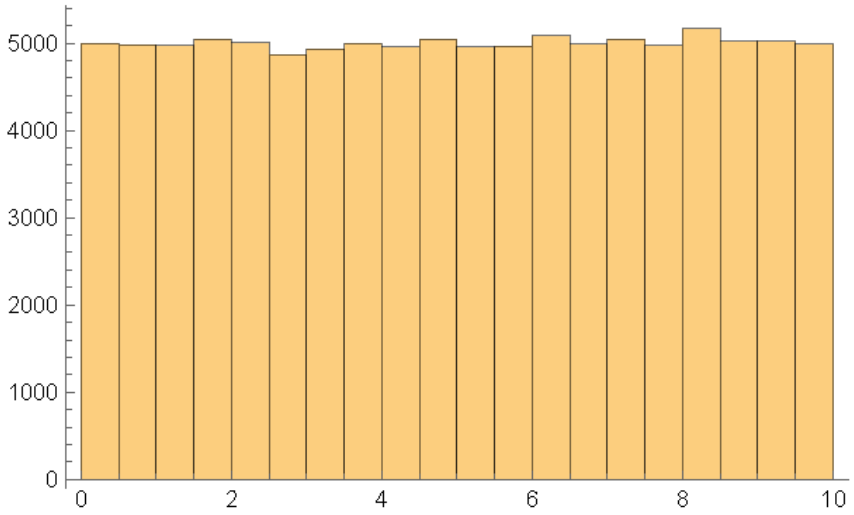




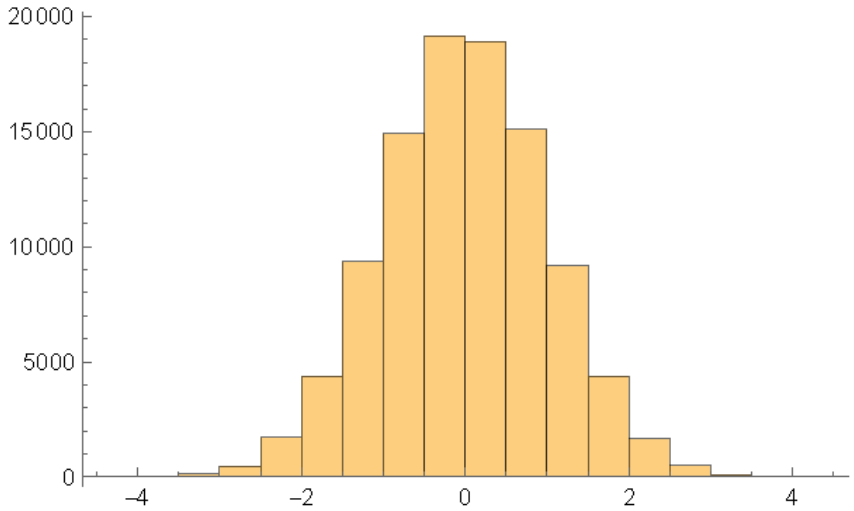
A, B, and C in Mathematica

In[8]:=
```
i = 100 000;
uni = RandomVariate[UniformDistribution[{0, 10}], i];
gauss = RandomVariate[NormalDistribution[0, 1], i];
poiss = RandomVariate[PoissonDistribution[5], i];

Histogram[uni]
Histogram[gauss]
Histogram[poiss, 15, "PDF"]
```

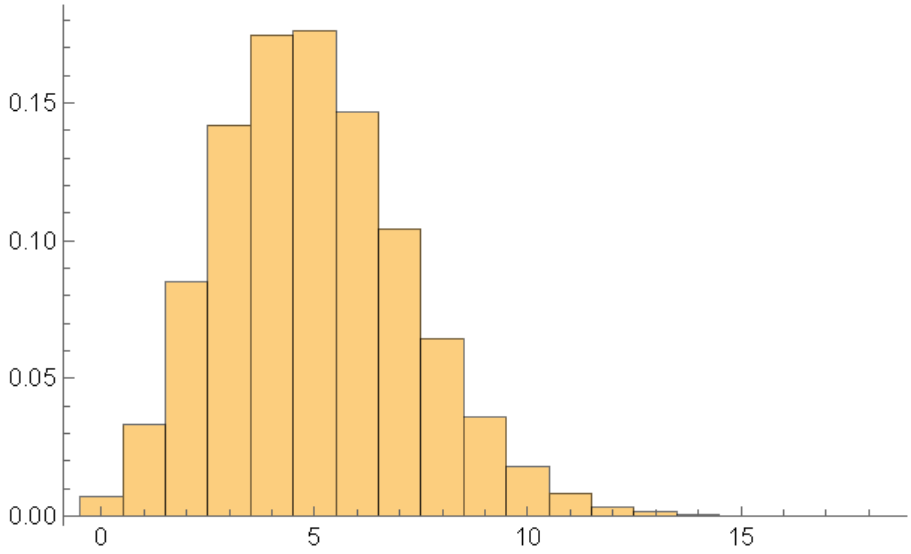
Out[12]=



Out[13]=



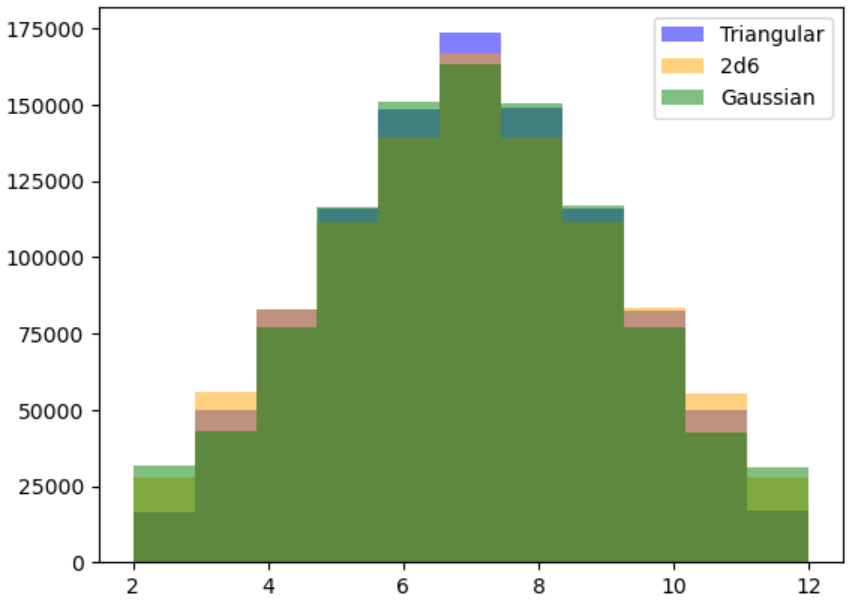
Out[14]=



2

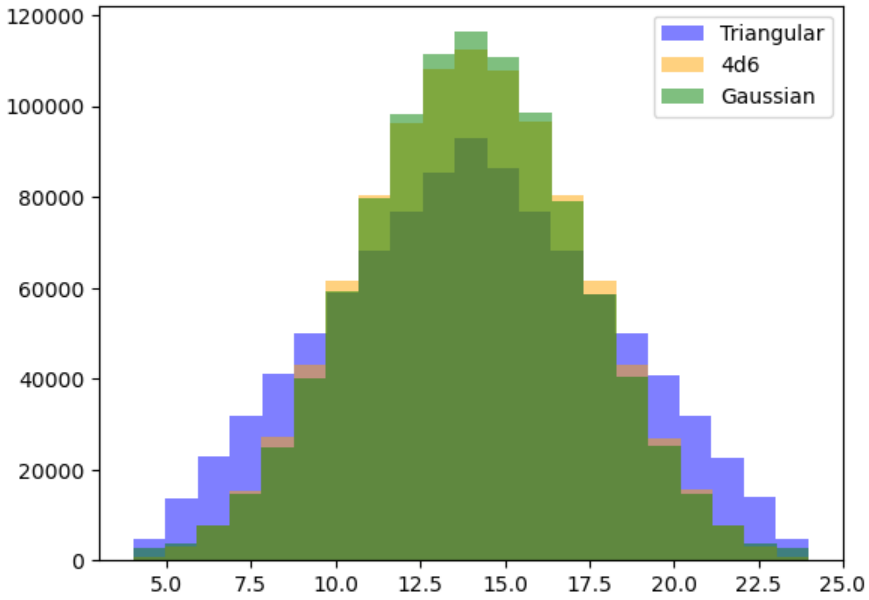
a

In [7]:
```
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

N=1_000_000

result=[]

for i in range(N):
    one_d6=randint(1,7)
    two_d6=randint(1,7)
    result.append(one_d6+two_d6)


tri = triangular(2, 7, 12, N)
gauss=randn(N)*2.42+7
gauss = clip(round(gauss), 2, 12)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins =11)
plt.hist(result, alpha=0.5, label='2d6', color='orange', bins=11)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=11)
plt.legend()
plt.show()
```



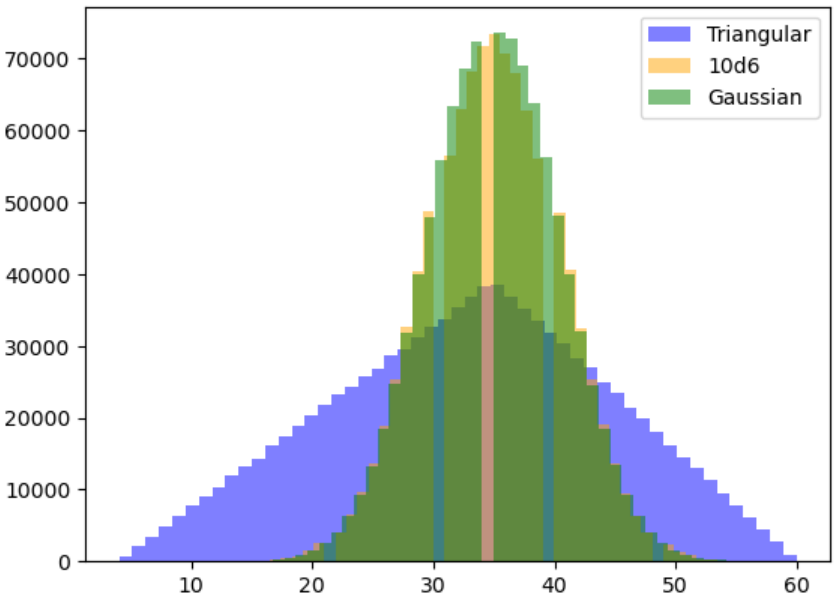
b

In [8]:
```
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

N=1_000_000

result=[]

for i in range(N):
    die_sum = sum([randint(1,7) for _ in range(4)])
    result.append(die_sum)

tri = triangular(4, 14, 24, N)
gauss=randn(N)*3.42+14
gauss = clip(round(gauss), 4, 24)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins = 21)
plt.hist(result, alpha=0.5, label='4d6', color='orange', bins=21)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=21)
plt.legend()
plt.show()
```



C

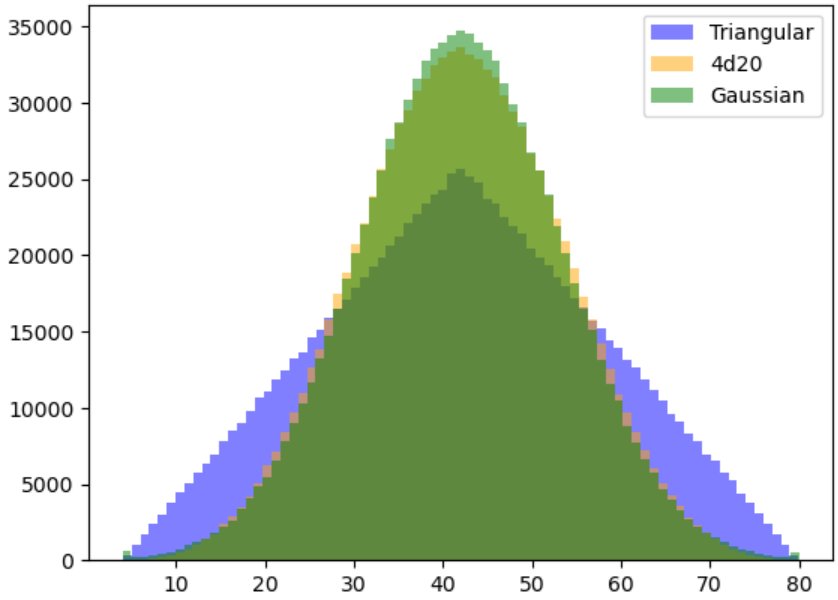
In [12]:
```
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

N=1_000_000

result=[]

for i in range(N):
    die_sum = sum([randint(1,7) for _ in range(10)])
    result.append(die_sum)

tri = triangular(4, 35, 60, N)
gauss=randn(N)*5.4+35
gauss = clip(round(gauss), 10, 60)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins =51)
plt.hist(result, alpha=0.5, label='10d6', color='orange', bins=51)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=51)
plt.legend()
plt.show()
```



We note that the trangular distribution matches perfectly, and the gaussian works well as long as we pick the correct standard deviation, center it, select the maximum and minimum values, and also only use d6 or lower dice. At 2-d10, we see the tail ends do not works well anymore.

d (4 d20 dice)

In [11]:
```
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

N=1_000_000

result=[]

for i in range(N):
    die_sum = sum([randint(1,21) for _ in range(4)])
    result.append(die_sum)


tri = triangular(4, 42, 80, N)
gauss=randn(N)*11.53+42
gauss = clip(round(gauss), 4, 80)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins =77)
plt.hist(result, alpha=0.5, label='4d20', color='orange', bins=77)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=77)
plt.legend()
plt.show()
```



Note:

With 4 d-20 dice, we see the triangular does not approximate the results very well. The gaussian becomes better at higher dice counts and higher possible rolls.

3

a

In [61]:
```
from numpy.random import randint

N=1_000_000

result=[]

for i in range(N):
    one_d6=randint(1,7)
    two_d6=randint(1,7)
    result.append(one_d6+two_d6)

count = sum((x>=6) & (x<=8) for x in result)
probability = count/N

print(f'Probability of getting between 6 and 8 using 2 fair d6 dice: {probability:.2f}')
```

```
Probability of getting between 6 and 8 using 2 fair d6 dice: 0.44
```

b

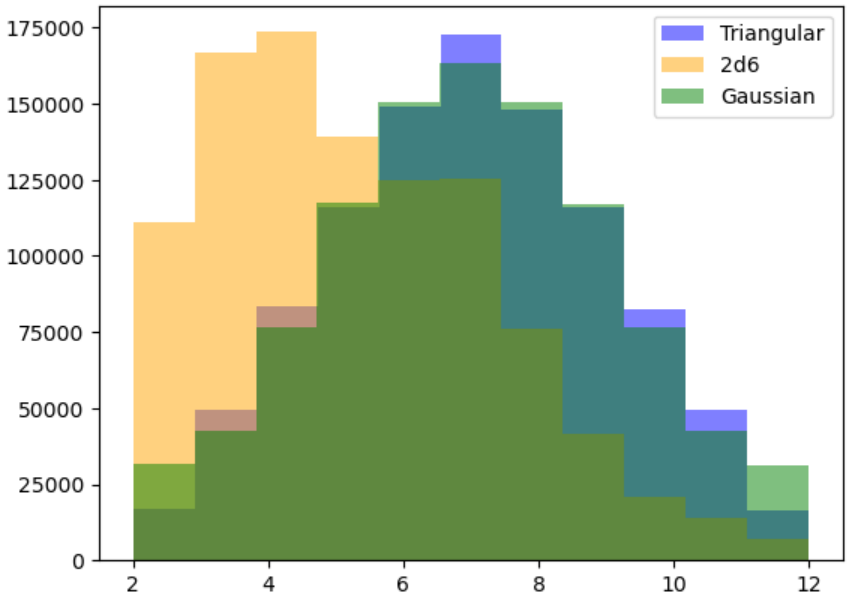
In [63]:
```
from numpy.random import randint

N=1_000_000

result=[]

for i in range(N):
    die_sum = sum([randint(1,7) for _ in range(4)])
    result.append(die_sum)


count = sum((x>=12) & (x<=16) for x in result)
probability = count/N

print(f'Probability of getting between 12 and 16 using 4 fair d6 dice: {probability:.2f}')
```

```
Probability of getting between 12 and 16 using 4 fair d6 dice: 0.52
```

4

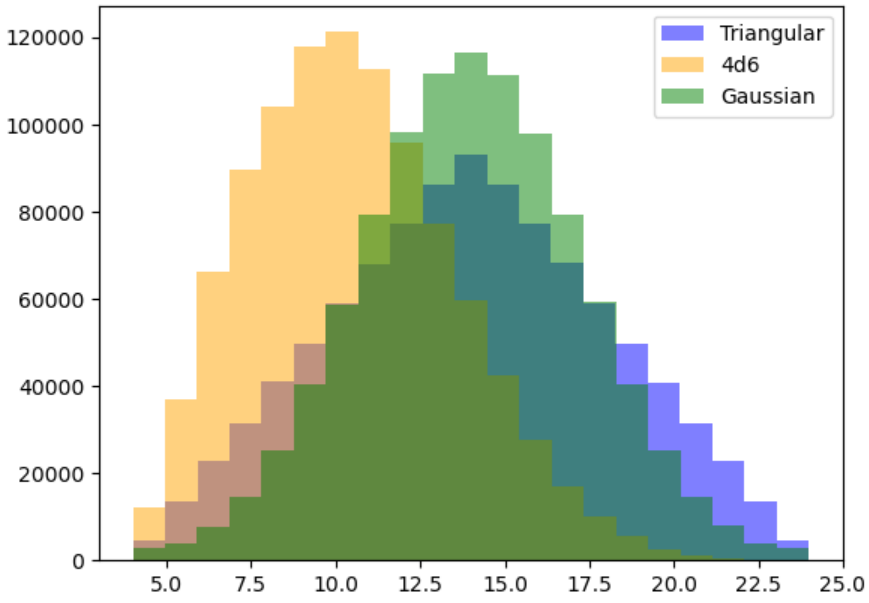
a

In [14]:
```
from numpy.random import choice
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

result = []
N=1_000_000

def dice_throw():
    sequence = [4,5,6,1,1,1,1,2,2,2,3,3]
    return choice(sequence)

for i in range(N):
    die_sum=sum([dice_throw() for _ in range(2)])
    result.append(die_sum)

tri = triangular(2, 7, 12, N)
gauss=randn(N)*2.42+7
gauss = clip(round(gauss), 2, 12)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins =11)
plt.hist(result, alpha=0.5, label='2d6', color='orange', bins=11)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=11)
plt.legend()
plt.show()
```



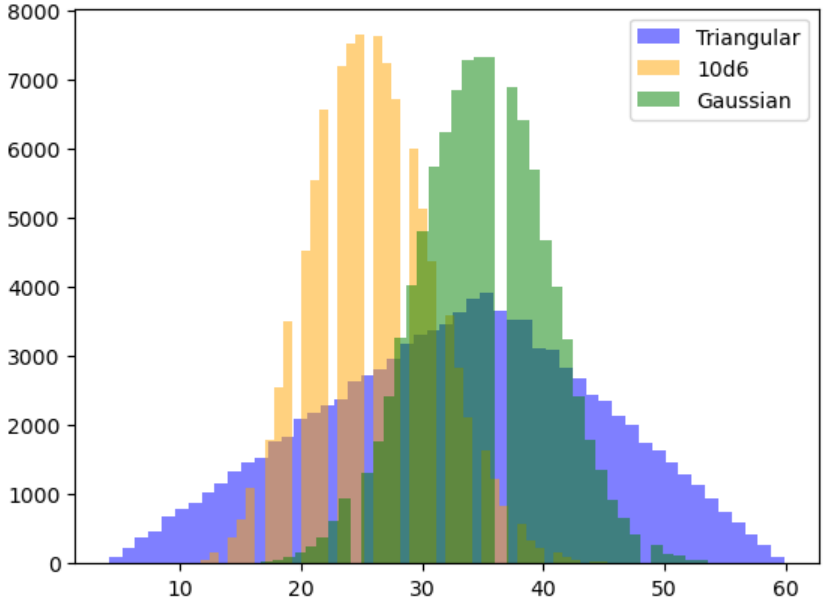
b

In [16]:
```
from numpy.random import choice
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

result = []
N=1_000_000

def dice_throw():
    sequence = [4,5,6,1,1,1,1,2,2,2,3,3]
    return choice(sequence)

for i in range(N):
    die_sum=sum([dice_throw() for _ in range(4)])
    result.append(die_sum)

tri = triangular(4, 14, 24, N)
gauss=randn(N)*3.42+14
gauss = clip(round(gauss), 4, 24)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins = 21)
plt.hist(result, alpha=0.5, label='4d6', color='orange', bins=21)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=21)
plt.legend()
plt.show()
```



C

In [22]:
```
from numpy.random import choice
from matplotlib import pyplot as plt
from numpy.random import uniform, randn, poisson, randint, triangular
from numpy import clip, round

result = []
N=100_000

def dice_throw():
    sequence = [4,5,6,1,1,1,1,2,2,2,3,3]
    return choice(sequence)

for i in range(N):
    die_sum=sum([dice_throw() for _ in range(10)])
    result.append(die_sum)

tri = triangular(4, 35, 60, N)
gauss=randn(N)*5.4+35
gauss = clip(round(gauss), 10, 60)

plt.hist(tri, alpha=0.5, label='Triangular', color='blue', bins =51)
plt.hist(result, alpha=0.5, label='10d6', color='orange', bins=51)
plt.hist(gauss, alpha=0.5, label='Gaussian', color='green', bins=51)
plt.legend()
plt.show()
```


Triangular
10d6
Gaussian
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60

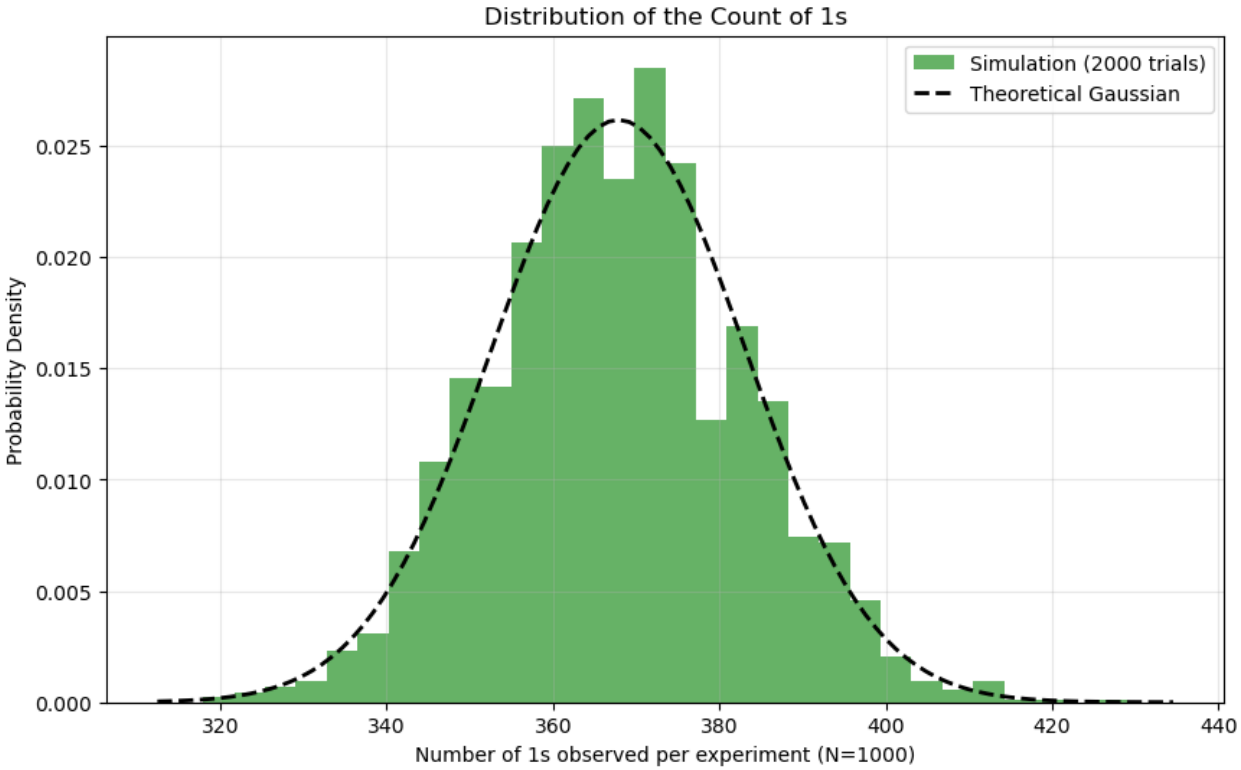
5

In [23]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# --- 1. Configuration & Data Generation ---
mu = 1.0           # Average count rate
N = 1000           # Number of measurements per experiment
np.random.seed(42) # Optional: ensures reproducible results

print(f"--- Simulation Parameters: N={N}, mu={mu} ---")

# Generate the main dataset for parts (a), (b), (c), and first half of (d)
measurements = np.random.poisson(lam=mu, size=N)


# --- 2. Part (a): How many results are 0? ---
print("\n--- Part (a): Zero Results ---")

# Experimental Count
zero_counts = np.count_nonzero(measurements == 0)

# Theoretical Expectation: E = N * P(0)
# P(0) for Poisson(1) is e^-1
prob_zero = np.exp(-mu)
theoretical_zeros = N * prob_zero

print(f"Observed zeros: {zero_counts}")
print(f"Expected zeros: {theoretical_zeros:.2f}")


# --- 3. Part (b): Probability of 5 consecutive zeros ---
print("\n--- Part (b): Probability of 5 Consecutive Zeros ---")

# We look for the pattern [0, 0, 0, 0, 0] anywhere in the array
# This counts overlapping patterns (e.g., 6 zeros = 2 patterns of 5)
sequence_length = 5
num_patterns_found = 0
possible_positions = N - sequence_length + 1

for i in range(possible_positions):
    window = measurements[i : i + sequence_length]
    if np.all(window == 0):
        num_patterns_found += 1

# Experimental Probability
experimental_prob_b = num_patterns_found / possible_positions

# Theoretical Probability: (e^-1)^5 = e^-5
theoretical_prob_b = np.exp(-5)

print(f"Patterns found: {num_patterns_found}")
print(f"Experimental Prob: {experimental_prob_b:.5f}")
print(f"Theoretical Prob:  {theoretical_prob_b:.5f}")


# --- 4. Part (c): Expected sequences of *at least* 5 consecutive zeros ---
print("\n--- Part (c): Distinct Sequences of >= 5 Zeros ---")

# Here we count distinct runs. A run of 6 zeros counts as 1 sequence, not 2.
distinct_sequences = 0
current_run_length = 0

for x in measurements:
    if x == 0:
        current_run_length += 1
    else:
        # The run of zeros has ended. Check if it was long enough.
        if current_run_length >= 5:
            distinct_sequences += 1
        current_run_length = 0

# Check if the array ends with a valid run
if current_run_length >= 5:
    distinct_sequences += 1

# Theoretical Expectation for runs of length >= k
# Formula: E = N * q * p^k (where p is prob of zero, q is prob of non-zero)
p = prob_zero
```

```
q = 1 - p
theoretical_sequences = N * q * (p**5)

print(f"Distinct sequences found: {distinct_sequences}")
print(f"Theoretical expectation:  {theoretical_sequences:.2f}")


# --- 5. Part (d): Expected 1s & Gaussian Distribution ---
print("\n--- Part (d): Analysis of 1s ---")

# Q1: How many 1s are expected?
ones_count = np.count_nonzero(measurements == 1)
# For Poisson with mu=1, P(1) = 1^1 * e^-1 / 1! = e^-1.
# So P(1) is exactly the same as P(0).
theoretical_ones = N * np.exp(-mu)

print(f"Observed 1s: {ones_count}")
print(f"Expected 1s: {theoretical_ones:.2f}")

# Q2: Do the number of 1s follow a Gaussian distribution?
print("\nRunning meta-simulation (2000 trials) to verify Gaussian distribution...")

# We repeat the entire 1000-measurement experiment 2000 times.
# We record the number of 1s found in each experiment.
meta_trials = 2000
counts_of_ones = []

for _ in range(meta_trials):
    trial_data = np.random.poisson(lam=mu, size=N)
    c = np.count_nonzero(trial_data == 1)
    counts_of_ones.append(c)

# Plotting
plt.figure(figsize=(10, 6))

# Histogram of experimental data
plt.hist(counts_of_ones, bins=30, density=True, alpha=0.6, color='green', label='Simulation (2000 trials)')

# Overplot Theoretical Gaussian (Normal) Curve
# The count of successes in N trials follows Binomial(N, p).
# For large N, this approximates Normal(mean=Np, var=Np(1-p)).
p_one = np.exp(-1)
theo_mean = N * p_one
theo_std = np.sqrt(N * p_one * (1 - p_one))

# Generate x-values for the curve
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)

# Normal Distribution PDF formula
pdf = (1 / (theo_std * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - theo_mean) / theo_std)**2)

plt.plot(x, pdf, 'k--', linewidth=2, label='Theoretical Gaussian')

plt.title('Distribution of the Count of 1s')
plt.xlabel('Number of 1s observed per experiment (N=1000)')
plt.ylabel('Probability Density')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("The close fit to the curve confirms the Gaussian distribution.")
```

```
--- Simulation Parameters: N=1000, mu=1.0 ---

--- Part (a): Zero Results ---
Observed zeros: 392
Expected zeros: 367.88

--- Part (b): Probability of 5 Consecutive Zeros ---
Patterns found: 13
Experimental Prob: 0.01305
Theoretical Prob:  0.00674

--- Part (c): Distinct Sequences of >= 5 Zeros ---
Distinct sequences found: 5
Theoretical expectation:  4.26

--- Part (d): Analysis of 1s ---
```

```
Observed 1s: 348
Expected 1s: 367.88

Running meta-simulation (2000 trials) to verify Gaussian distribution...
```

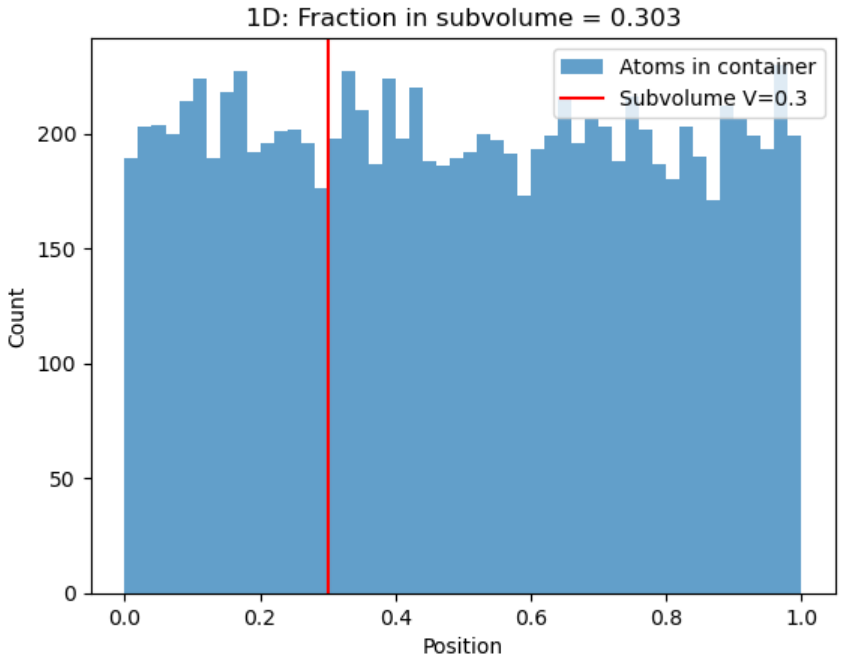



```
The close fit to the curve confirms the Gaussian distribution.
```

6

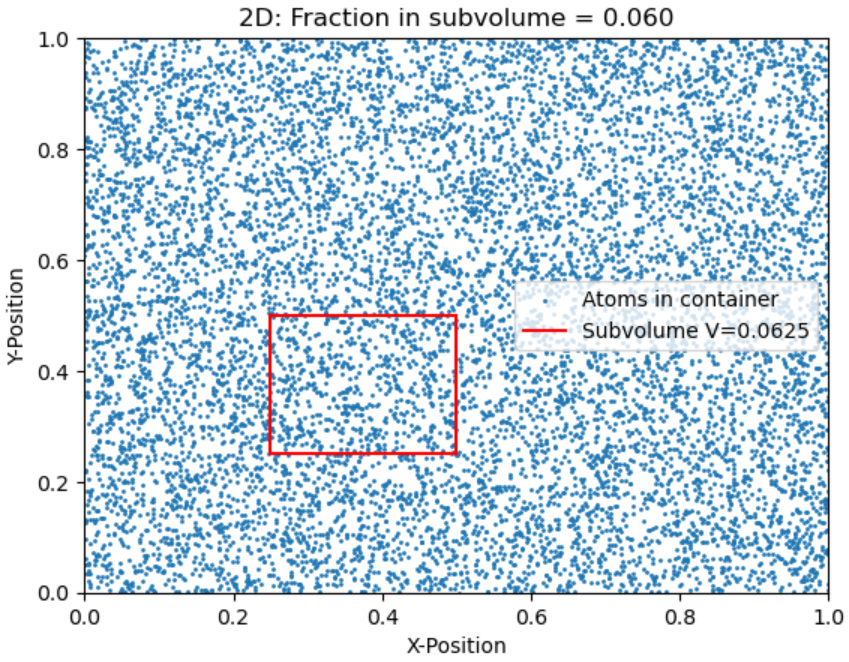
a

In [33]:
```python
from numpy.random import rand
from numpy import linspace
from matplotlib import pyplot as plt

N = 10_000
V = 0.3  # Choose a subvolume

X = rand(N)
count = sum(X < V)
fraction = count / N

plt.hist(X, bins=50, alpha=0.7, label=f'Atoms in container')
plt.axvline(V, color='red', label=f'Subvolume V={V}')
plt.xlabel('Position')
plt.ylabel('Count')
plt.title(f'1D: Fraction in subvolume = {fraction:.3f}')
plt.legend()
plt.show()

print(f"Fraction of atoms in subvolume: {fraction:.4f}")
print(f"Expected fraction: {V:.4f}")
```



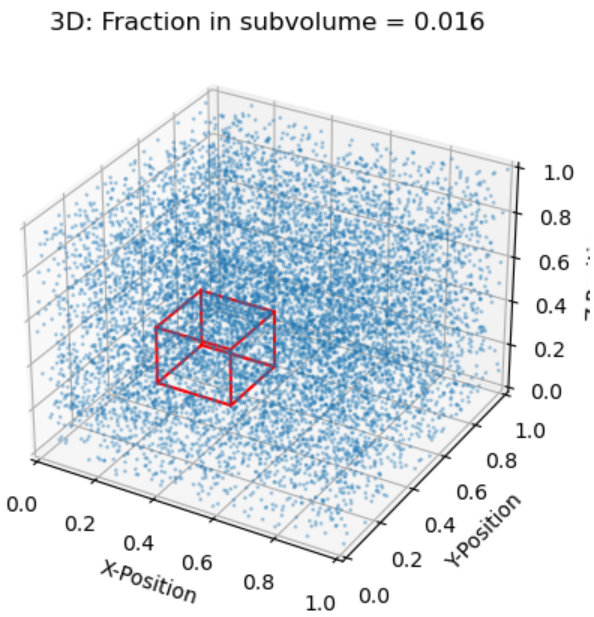

```
Fraction of atoms in subvolume: 0.3031
Expected fraction: 0.3000
```

In [43]:
```python
from numpy.random import rand
from numpy import linspace
from matplotlib import pyplot as plt
N = 10_000
x_min, x_max = 0.25, 0.5
y_min, y_max = 0.25, 0.5
V = (x_max - x_min) * (y_max - y_min)

X = rand(N)
Y = rand(N)

count = sum((X > x_min) & (X < x_max) & (Y > y_min) & (Y < y_max))
fraction = count / N

plt.scatter(X, Y, s=1, label=f'Atoms in container')
# Draw rectangle outline
plt.axvline(x_min, color='red', ymin=y_min, ymax=y_max)
plt.axvline(x_max, color='red', ymin=y_min, ymax=y_max)
plt.axhline(y_min, color='red', xmin=x_min, xmax=x_max)
plt.axhline(y_max, color='red', xmin=x_min, xmax=x_max, label=f'Subvolume V={V}')
plt.xlabel('X-Position')
plt.ylabel('Y-Position')
plt.title(f'2D: Fraction in subvolume = {fraction:.3f}')
plt.legend()
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.show()

print(f"Fraction of atoms in subvolume: {fraction:.4f}")
print(f"Expected fraction: {V:.4f}")
```




```
Fraction of atoms in subvolume: 0.0605
Expected fraction: 0.0625
```

In [45]:
```
from numpy.random import rand
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

N = 10_000 # How many particles do you want
x_min, x_max = 0.25, 0.5 # Set the x dimenstions of the box we are considering
y_min, y_max = 0.25, 0.5 # Set y
z_min, z_max = 0.25, 0.5 # Set z

V = (x_max - x_min) * (y_max - y_min) * (z_max - z_min)

X = rand(N)
Y = rand(N)
Z = rand(N)

count = sum((X > x_min) & (X < x_max) & (Y > y_min) & (Y < y_max) & (Z > z_min) & (Z < z_max))
fraction = count / N

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X, Y, Z, s=1, alpha=0.3, label='Atoms in container')

# Draw box outline for subvolume
corners = [[x_min, x_max], [y_min, y_max], [z_min, z_max]]

for s, e in [(0,1), (2,3), (4,5), (6,7), (0,2), (1,3), (4,6), (5,7), (0,4), (1,5), (2,6), (3,7)]:
    xs = [x_min if s & 1 == 0 else x_max, x_min if e & 1 == 0 else x_max]
    ys = [y_min if (s >> 1) & 1 == 0 else y_max, y_min if (e >> 1) & 1 == 0 else y_max]
    zs = [z_min if (s >> 2) & 1 == 0 else z_max, z_min if (e >> 2) & 1 == 0 else z_max]
    ax.plot3D(xs, ys, zs, 'r-')

ax.set_xlabel('X-Position')
ax.set_ylabel('Y-Position')
ax.set_zlabel('Z-Position')
ax.set_title(f'3D: Fraction in subvolume = {fraction:.3f}')
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_zlim(0, 1)
plt.show()

print(f"Fraction of atoms in subvolume: {fraction:.4f}")
print(f"Expected fraction: {V:.4f}")
```




```
Fraction of atoms in subvolume: 0.0161
Expected fraction: 0.0156
```

d

Comments: This is cool? I like how I got a red box to visualize what is going on without too much trouble

7

For a interesting simulation, I will calculate the probability of someone phasing through the door.

In [87]:
```python
import numpy as np

# Constants
hbar = 1.055e-34  # J·s
m_proton = 1.67e-27  # kg (proton mass)
v = 1  # m/s
L = 0.043  # m (door thickness)

# Energy and barrier (per particle)
E = 0.5 * m_proton * v**2
V0 = 2 * E

# Tunneling probability per particle
kappa = np.sqrt(2 * m_proton * (V0 - E)) / hbar
exponent = -2 * kappa * L

# For 1 particle to tunnel: N_particles * T = 1
# N_particles = 1 / T = exp(-exponent)
N_particles_needed = np.exp(-exponent)

# Mass needed
m_person_needed = N_particles_needed * m_proton

print(f"Tunneling probability per particle: T = exp({exponent:.2e})")
print(f"Particles needed for 1 to tunnel: {N_particles_needed:.2e}")
print(f"Mass needed: {m_person_needed:.2e} kg")
print(f"In solar masses: {m_person_needed / 2e30:.2e}")
print(f"In universe masses: {m_person_needed / 1e53:.2e}")
```

```
Tunneling probability per particle: T = exp(-1.36e+06)
Particles needed for 1 to tunnel: inf
Mass needed: inf kg
In solar masses: inf
In universe masses: inf
C:\Users\miles\AppData\Local\Temp\ipykernel_12328\578615797.py:19: RuntimeWarning: overflow encountered in exp
  N_particles_needed = np.exp(-exponent)
```

So it will basically never happen, even if you weighed as much as the sun

In [ ]: