AppliedW9

April 26, 2024

1 Question 1

You generate a random number $u = rand()$, which is uniformly distributed in $[0, 1)$. The sample is then given by:

$$X = \begin{cases} 1 & \text{if } u < p_1 \\ 2 & \text{if } p_1 \leq u < p_1 + p_2 \\ 3 & \text{if } p_1 + p_2 \leq u < p_1 + p_2 + p_3 \\ \vdots & \\ i & \text{if } \sum_{j=1}^{i-1} p_j \leq u < \sum_{j=1}^{i} p_j \\ \vdots & \\ n & \text{if } \sum_{j=1}^{n-1} p_j \leq u < 1 \end{cases}$$

2 Question 2

Let x be the amount of fish, and y denote sharks. In the original Lotka-Volterra we had:

$$\frac{dx}{dt} = rx - fx - \alpha xy$$
$$\frac{dy}{dt} = (s + \beta x)y - fy$$

where the constant f, corresponds to the fishing rate; r and s are the rate of growth for fish and shark respectively; α is the prop. constant of fish being eaten by shark, and β is the prop. constant of shark surviving by eating fish.

We can identify 5 events:

1. Fish born, with propensity rx; $\rightarrow x = x + 1$.
2. Fish out by means of fishing, with propensity fx ; $\rightarrow x = x - 1$.
3. Fish eaten by shark, with propensity αxy ; $\rightarrow x = x - 1$.
4. Shark die due to natural birth rate with propensity $|sy|$, note that since s is usually negative we will take it's absolute value to get propensity.

5. Shark surviving with propensity βxy; note that we have included the effect of increased survival through eating fish; $\rightarrow y = y + 1$.
6. Shark die due to fishing, with propensity fy ; $\rightarrow y = y - 1$.

The algorithm goes as follows:

- For each event $e_1, \ldots, e_5$ we determine the propensity of occurrence $p_1, \ldots, p_5$, based on the expressions above.
- We sample the time to occurrence for each event, $t_i = -\frac{\log u}{p_i}$, where $u = rand()$.
- Find the event that happens the earliest $t = \min_{1 \leq i \leq 5}(t_i)$
- $t_{curr} = t_{prev} + t$, and update variables x and y as dictated by the chosen event.
- Repeat.

[96]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy.linalg import norm
```

2.1 Gillespie's Algorithm Implementation for Lotka Volterra

[97]:
```
norm2 = lambda x: norm(x, ord=2)

def generate_gillespie_LV(r, s, f, alpha, beta):
    fish_born_propensity = lambda t, xy: np.abs(r*xy[0])
    fish_fished_propensity = lambda t, xy: np.abs(f*xy[0])
    fish_sharked_propensity = lambda t, xy: np.abs(alpha*xy[0]*xy[1])
    shark_death_propensity = lambda t, xy: np.abs(xy[1]*s)
    shark_survived_propensity = lambda t, xy: np.abs(beta*xy[0]*xy[1])
    shark_fished_propensity = lambda t, xy: np.abs(f*xy[1])

    fish_born = lambda t, xy: np.array([xy[0] + 1, xy[1]])
    fish_fished = lambda t, xy: np.array([xy[0] - 1, xy[1]])
    fish_sharked = lambda t, xy: np.array([xy[0] - 1, xy[1]])
    shark_death = lambda t, xy: np.array([xy[0], xy[1]-1])
    shark_survived = lambda t, xy: np.array([xy[0], xy[1]+1])
    shark_fished = lambda t, xy: np.array([xy[0], xy[1]-1])

    propensity_vector = lambda t, xy: np.array([
        fish_born_propensity(t, xy),
        fish_fished_propensity(t, xy),
        fish_sharked_propensity(t, xy),
        shark_death_propensity(t, xy),
        shark_survived_propensity(t, xy),
        shark_fished_propensity(t, xy),
    ])
```

```
    event_vector = lambda t, xy: np.array([
        fish_born(t, xy),
        fish_fished(t, xy),
        fish_sharked(t, xy),
        shark_death(t, xy),
        shark_survived(t, xy),
        shark_fished(t, xy),
    ])

    return propensity_vector, event_vector
```

[104]:
```
def solve_LV_stochastic(t0=0, xy0=np.array([5, 5]),
                        r=0.8, s=-0.8, f=0.1, alpha=0.045, beta=0.03,
                        max_iter=1000, verbose=False, random_state=None):
    np.random.seed(random_state)
    props, events = generate_gillespie_LV(r=r, s=s, f=f, alpha=alpha, beta=beta)
    T = np.zeros(max_iter+1)
    T[0] = t0
    XY = np.zeros(shape=(len(xy0), max_iter+1))
    XY[:, 0] = xy0

    tte = lambda p: -np.log(np.random.random(len(p)))/p # function for sampling␣
 ↪time to next event

    for i in range(max_iter):
        p = props(t=T[i], xy = XY[:, i]) # propensities
        if norm2(p) < 1e-10:
            print('Stopping early because everyone is dead')
            return T[:i], XY[:, :i]
        p = p/sum(p) # probabilities
        t = tte(p) # randomly sampled time to next event
        idx = np.argmin(t) # index of soonest event
        T[i+1] = T[i] + t[idx] # update time with soonest event
        XY[:, i+1] = events(T[i], XY[:, i])[idx] #update populations with␣
 ↪consequence of soonest event
        if verbose: print('t={0}, fish={1}, sharks={2}, event={3}'.format(T[i],␣
 ↪XY[0, i], XY[1, i], idx))
    return T, XY
```

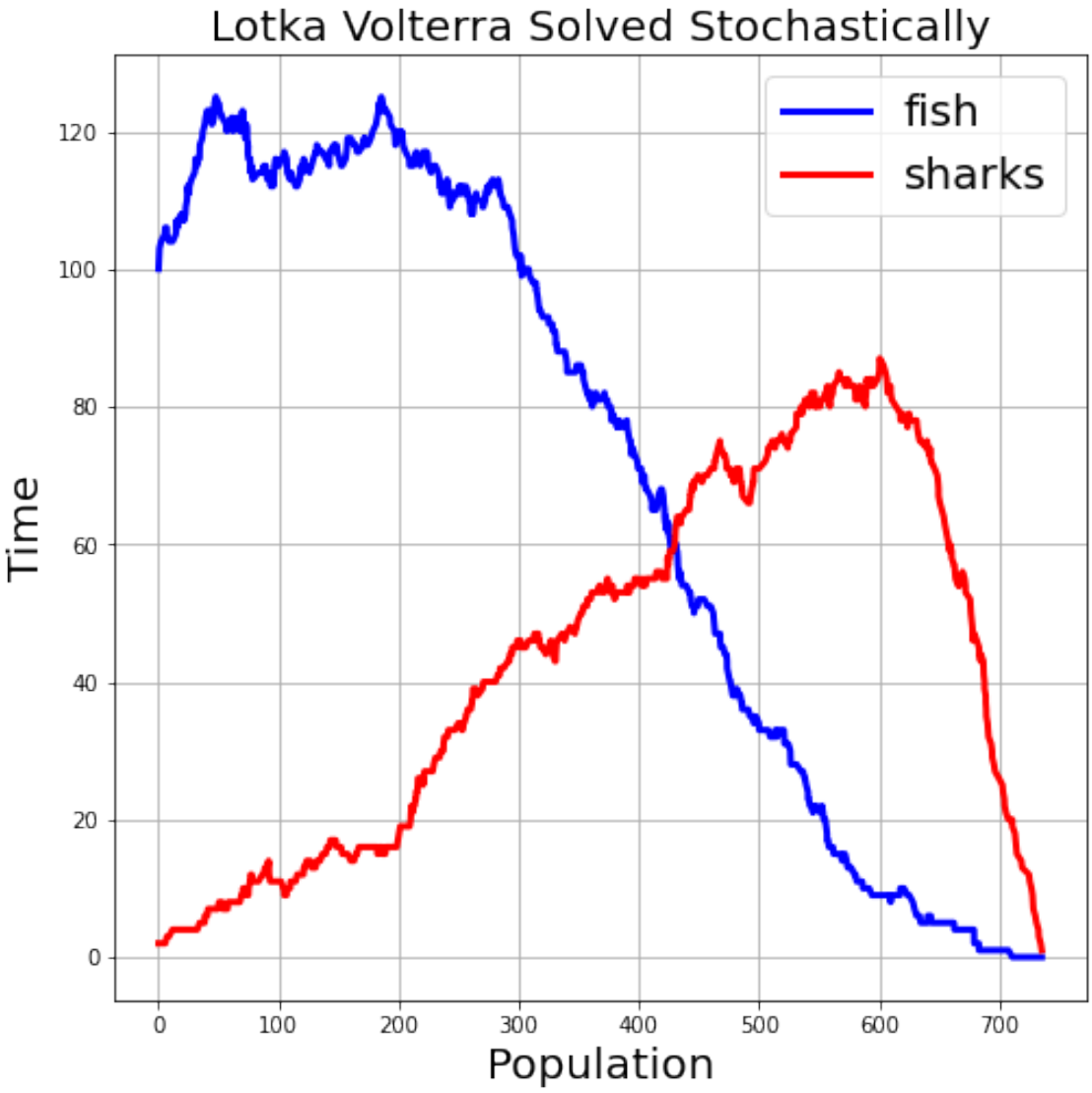
[107]:
```
xy0 = np.array([100, 2])
t, xy = solve_LV_stochastic(xy0=xy0, max_iter=1000, verbose=False,␣
 ↪random_state=None)

fig, axs = plt.subplots(1, 1, sharey=False, sharex=False)
fig.set_figwidth(8)
fig.set_figheight(8)
axs.plot(t, xy[0], label='fish', c='b', lw=3)
```

```
axs.plot(t, xy[1], label='sharks', c='r', lw=3)
axs.legend(fontsize=20)
axs.grid()
axs.set_xlabel('Population', size=20)
axs.set_ylabel('Time', size=20)
axs.set_title('Lotka Volterra Solved Stochastically', size=20)
```

```
Stopping early because everyone is dead
```

```
/tmp/ipykernel_103091/3749321015.py:11: RuntimeWarning: divide by zero
encountered in true_divide
  tte = lambda p: -np.log(np.random.random(len(p)))/p # function for sampling
time to next event
```

[107]: `Text(0.5, 1.0, 'Lotka Volterra Solved Stochastically')`

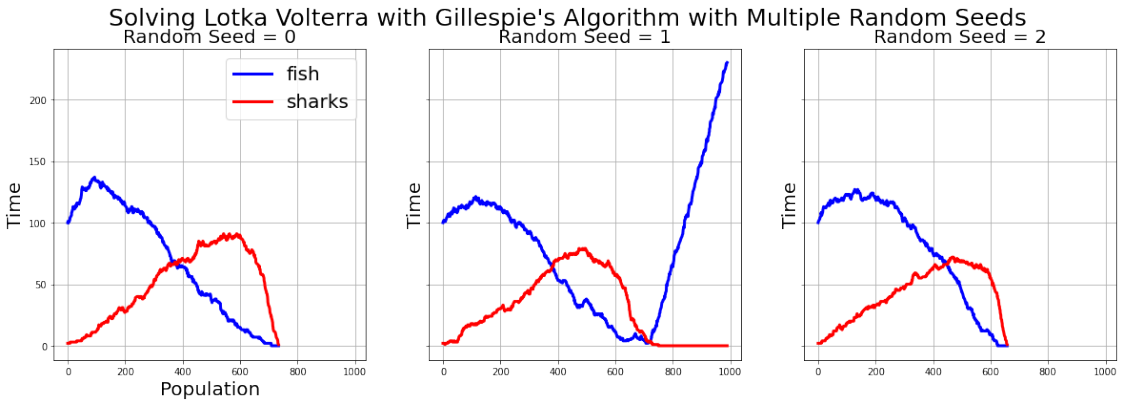


[116]:
```
xy0 = np.array([100, 2])
reps = 3
fig, axs = plt.subplots(1, reps, sharey=True, sharex=True)
fig.set_figwidth(6*reps+2)
fig.set_figheight(6)
for i in range(reps):
    t, xy = solve_LV_stochastic(xy0=xy0, max_iter=1000, verbose=False,␣
 ↪random_state=i)
    axs[i].plot(t, xy[0], label='fish', c='b', lw=3)
    axs[i].plot(t, xy[1], label='sharks', c='r', lw=3)
    axs[i].grid()
    axs[i].set_title('Random Seed = {0}'.format(i), size=20)
    axs[i].set_ylabel('Time', size=20)
axs[0].legend(fontsize=20)
axs[0].set_xlabel('Population', size=20)
fig.suptitle("Solving Lotka Volterra with Gillespie's Algorithm with Multiple␣
 ↪Random Seeds", size=25)
```

```
Stopping early because everyone is dead
Stopping early because everyone is dead
```

```
/tmp/ipykernel_103091/3749321015.py:11: RuntimeWarning: divide by zero
encountered in true_divide
  tte = lambda p: -np.log(np.random.random(len(p)))/p # function for sampling
time to next event
/tmp/ipykernel_103091/3749321015.py:11: RuntimeWarning: divide by zero
encountered in true_divide
  tte = lambda p: -np.log(np.random.random(len(p)))/p # function for sampling
time to next event
/tmp/ipykernel_103091/3749321015.py:11: RuntimeWarning: divide by zero
encountered in true_divide
  tte = lambda p: -np.log(np.random.random(len(p)))/p # function for sampling
time to next event
```

[116]:
```
Text(0.5, 0.98, "Solving Lotka Volterra with Gillespie's Algorithm with Multiple
Random Seeds")
```



3 Question 3

To obtain the canonical form, you reorder the states in such a way that transient states come first. The upper left submatrix constitutes the matrix Q, for transitions between transient states; and the matrix R (upper right), contains transition probabilities from transient into absorbing states.

A possible re-ordering yields:

$$Q = \begin{pmatrix} 0 & \frac{2}{3} \\ \frac{1}{3} & 0 \end{pmatrix}$$

and

$$R = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{2}{3} \end{pmatrix}$$

The first step is to compute the fundamental matrix:

$$N = (I - Q)^{-1}$$

.

$$N = \begin{pmatrix} 9/7 & 6/7 \\ 3/7 & 9/7 \end{pmatrix}$$

For absorption times, we compute:

$$N \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 15/7 \\ 12/7 \end{pmatrix}$$

We have three transient states, thus each component of the resulting vector is the expected number of steps required for fixation starting in each one of the states. For absorption probabilities we compute $B = NR$. This yields:

$$B = \begin{pmatrix} 3/7 & 4/7 \\ 1/7 & 6/7 \end{pmatrix}$$

Each component represents the probability of being absorbed into one of the two absorbing states, given I start in a particular transient state.

4 Question 4

First, we show that $N = (I - Q)^{-1}$ gives $N = I + Q + Q^2 + Q^3 + \cdots$. Start noting that:

$$(I-Q)(I+Q^1+Q^2+\cdots+Q^n) = (I+Q^1+Q^2+\cdots+Q^n) - (Q^1+Q^2+\cdots+Q^n+Q^{n+1}) = I - Q^{n+1}$$

Multiplying both sides by N yields:

$$I + Q^1 + Q^2 + \cdots + Q^n = N(I - Q^{n+1})$$

Remember that $\lim_{n\to\infty} Q^n = \mathbf{0}$. Thus, letting $n \to \infty$, we obtain:

$$I + Q^1 + Q^2 + Q^3 \cdots = N$$

Now, to prove the opposite direction, assume $N = I + Q + Q^2 + Q^3 + \cdots$. Multiplying both sides by Q, we have:

$$N \cdot Q = (I + Q + Q^2 + Q^3 + \cdots) \cdot Q = Q + Q^2 + Q^3 + \cdots$$

Now, by subtracting the above equation from N, we obtain:

$$N \cdot (I - Q) = I$$

which gives $N = (I - Q)^{-1}$.
Optional:

Let $X^{(k)}$ be a random variable which equals 1 if the chain is in state s_j after k steps starting from state s_i, and equals 0 otherwise. We have

$$P\left(X^{(k)} = 1\right) = q_{ij}^{(k)},$$

and

$$P\left(X^{(k)} = 0\right) = 1 - q_{ij}^{(k)},$$

where $q_{ij}^{(k)}$ is the ij-th entry of Q^k. Using this, the number of times we are expect to visit state j starting from state i in n steps can be given by:

$$\begin{aligned} E\left(X^{(0)} + X^{(1)} + \cdots + X^{(n)}\right) &= E\left(X^{(0)}\right) + E\left(X^{(1)}\right) + \cdots + E\left(X^{(n)}\right) \\ &= q_{ij}^{(0)} + q_{ij}^{(1)} + \cdots + q_{ij}^{(n)} \end{aligned}$$

Here we have made use of the fact that $E\left(X^{(k)}\right) = q_{ij}^{(k)}$ since $X^{(k)}$ is a 0-1 random variable. Taking n to infinity and setting this equal to n_{ij} gives that n_{ij} is the expected number of times the chain visits state j starting in state i before absorption.

5 Question 5

5.1 Snakes and Ladders

5.2 A: Transition Matrix and Canonical Form

[159]:
```
def transition_matrix(size, pairs, d=6):
    T = np.zeros(shape=(size,size))
    for i in range(size - d):
        T[i,(i+1):(i+1+d)] = 1/d
    for i in range(size-d, size-1):
        T[i,(i+1):-1] = 1/d
        T[i, -1] = ((2+d-(size-i)))/d
    T[-1, -1] = 1

    for (s, e) in pairs:
        T[s-d:s, s] = 0
        T[s-d:s, e] += 1/d

    return T
```

[160]:
```
T = transition_matrix(size=8, pairs=[(5, 2)], d=4)
T
```

[160]:
```
array([[0.  , 0.25, 0.25, 0.25, 0.25, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.5 , 0.25, 0.25, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.25, 0.25, 0.25, 0.  , 0.25, 0.  ],
       [0.  , 0.  , 0.25, 0.  , 0.25, 0.  , 0.25, 0.25],
       [0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.25, 0.5 ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.75],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  ]])
```

[161]:
```
pairs = [(2, 16),
         (19, 33),
         (27, 41),
         (39, 45),
         (46, 60),
         (24, 10),
         (31, 15),
         (47, 33),
         (57, 44)]
size=63
A = transition_matrix(size=size, pairs=pairs,d=6)
A[0, 0:17]
```

[161]:
```
array([0.        , 0.16666667, 0.16666667, 0.16666667, 0.16666667,
       0.16666667, 0.16666667, 0.        , 0.        , 0.        ,
```

```
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        ])
```

[162]:
```
def extract_canonical_components(M):
    D = np.diag(M)
    absorbing, transient = np.where(D==1)[0], np.where(D!=1)[0]
    I = M[absorbing, :][:, absorbing]
    Q = M[transient, :][:, transient]
    R = M[transient, :][:, absorbing]
    O = M[absorbing, :][:, transient]
    return I, Q, R, O, absorbing, transient
```

[163]:
```
M = np.array([
    [1, 0, 0, 0],
    [1/3, 0, 2/3, 0],
    [0, 1/3, 0, 2/3],
    [0, 0, 0, 1]
])
```

[164]:
```
I, Q, R, O, absorbing, transient = extract_canonical_components(M)
absorbing, transient
```

[164]:
```
(array([0, 3]), array([1, 2]))
```

[165]:
```
Q, R
```

[165]:
```
(array([[0.        , 0.66666667],
        [0.33333333, 0.        ]]),
 array([[0.33333333, 0.        ],
        [0.        , 0.66666667]]))
```

5.3 B: Fundamental Matrix

[166]:
```
def fundamental_matrix(M):
    I, Q, R, O, absorbing, transient = extract_canonical_components(M)
    N = np.linalg.inv(I-Q)
    return N, transient
```

[167]:
```
N, transient = fundamental_matrix(M)
expected_visits = np.sum(N, axis=0)
least_visited_idx = np.argmin(expected_visits)
least_visited  = transient[least_visited_idx]
least_visited
```

[167]:
```
1
```

5.4 C: Absorption Time

[168]:
```
absorbtion_times = np.dot(N, np.ones_like(transient))
total_time = np.sum(absorbtion_times)
absorbtion_times, total_time
```

[168]:
```
(array([2.14285714, 1.71428571]), 3.8571428571428563)
```

5.5 Absorbtion Probabilities

[169]:
```
absorbtion_probabilities = np.dot(N, R)
absorbtion_probabilities
```

[169]:
```
array([[0.42857143, 0.57142857],
       [0.14285714, 0.85714286]])
```

5.6 Simulation

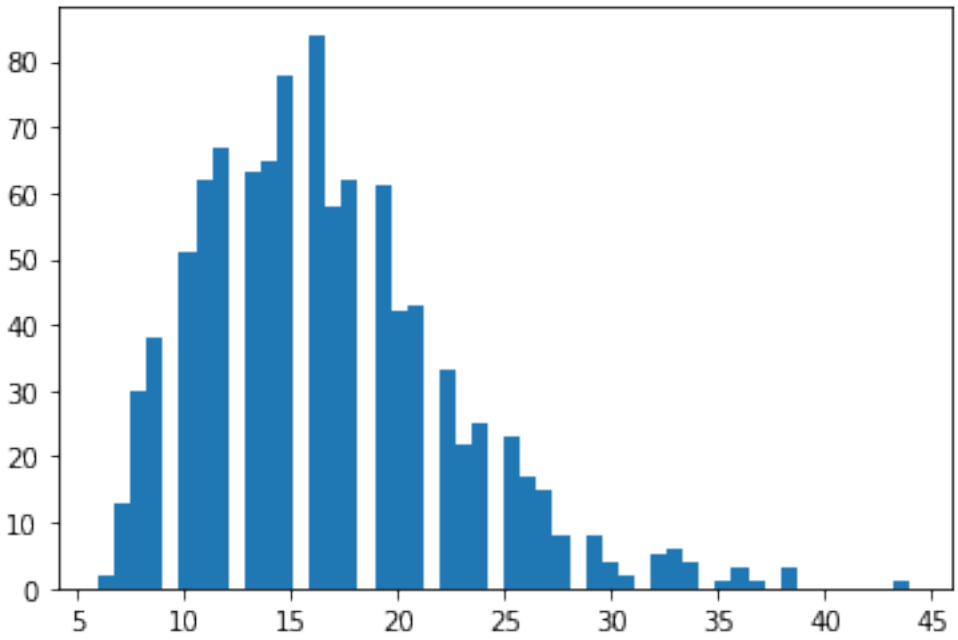
[170]:
```
def d6():
    return np.random.randint(1,6)+1

def game(T, max_iter=100):
    pos = 0
    spaces = list(range(T.shape[0]))
    visited = [pos]
    for i in range(max_iter):
        pos = np.random.choice(spaces, p=T[pos])
        visited += [pos]
        if pos == len(T)-1:
            return i, visited
    return max_iter, visited
```

[171]:
```
n_games = 1000
results = []
vis = []
for i in range(n_games):
    print('.', end='')
    rounds, visited = game(A)
    results += [rounds]
    vis += visited
```

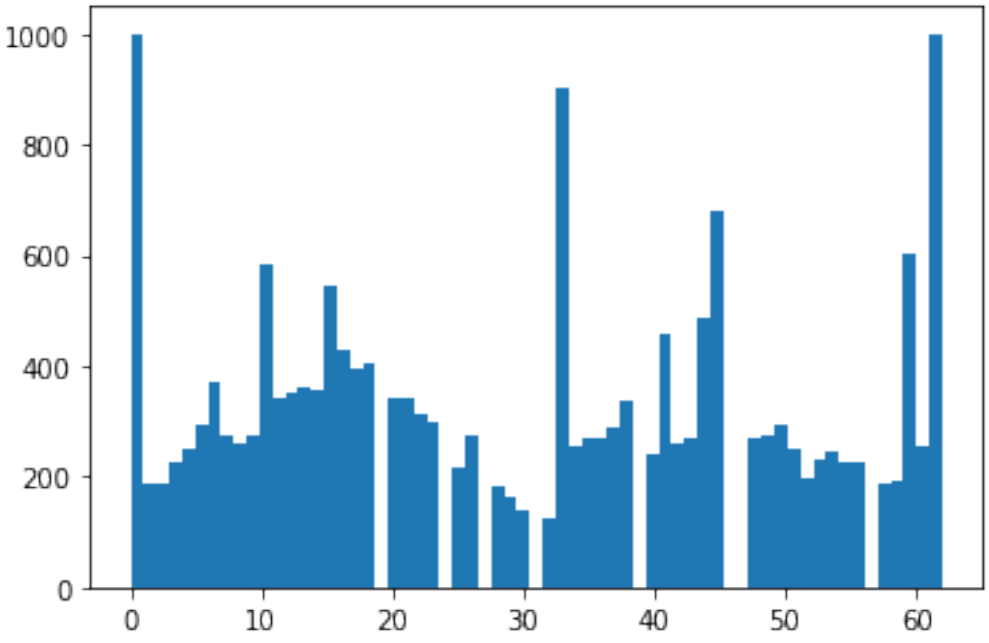
```
...
...
...
...
...
...
...
```

...
...
...
...
...
...

[172]:
```
hist = plt.hist(results, bins=50)
```



[173]:
```
hist2 = plt.hist(vis, bins=63)
```


1000
800
600
400
200
0
0
10
20
30
40
50
60