SI140 Project 3 Reverse Engineering the Mechanism of WeChat Red Envelope

January 16, 2026

1 Contributions of members

*Although the roles are different, all three members of the team have made **similar level of effort** to complete this project.*

- 2024533141 Boyang Zhou:
  - Design the mechanisms
  - Take part in the experiment and disposal data
  - Complete a small part of code work
  - Do the paperwork
- 2024533010 Zian Chen:
  - Optimize the mechanisms
  - Take part in the experiment
  - Complete most of the code work
  - Design and conduct simulation experiments
- 2024533002 Zidong Song:
  - Calculate and prove the expectation and variance of the mechanisms
  - Contact and organize the conduct of the experiment
  - Complete a small part of code work
  - Verify the results of the simulation experiments

2 Experiment, Data Processing and Visualization

We have conducted experiments together with other 4 teams (Mohan Li, Xinyi Lu, Zixiu Lin; Lufei Li, Zhan Jin, Sixian Shen; Junyang Wu, Zhengran Shao, Haojia Zhang; Lei Qian, Zihao Zhang, Jingqi Xu) and share the data set we get.

We conducted experiments on the following condition:

- Each WRE contains 60 RMB.

- Each WRE is allocated to 15 people.
- We send totally 150 WREs and collect the amount of money each participant received in each of the WREs.

The data is shown as follows:

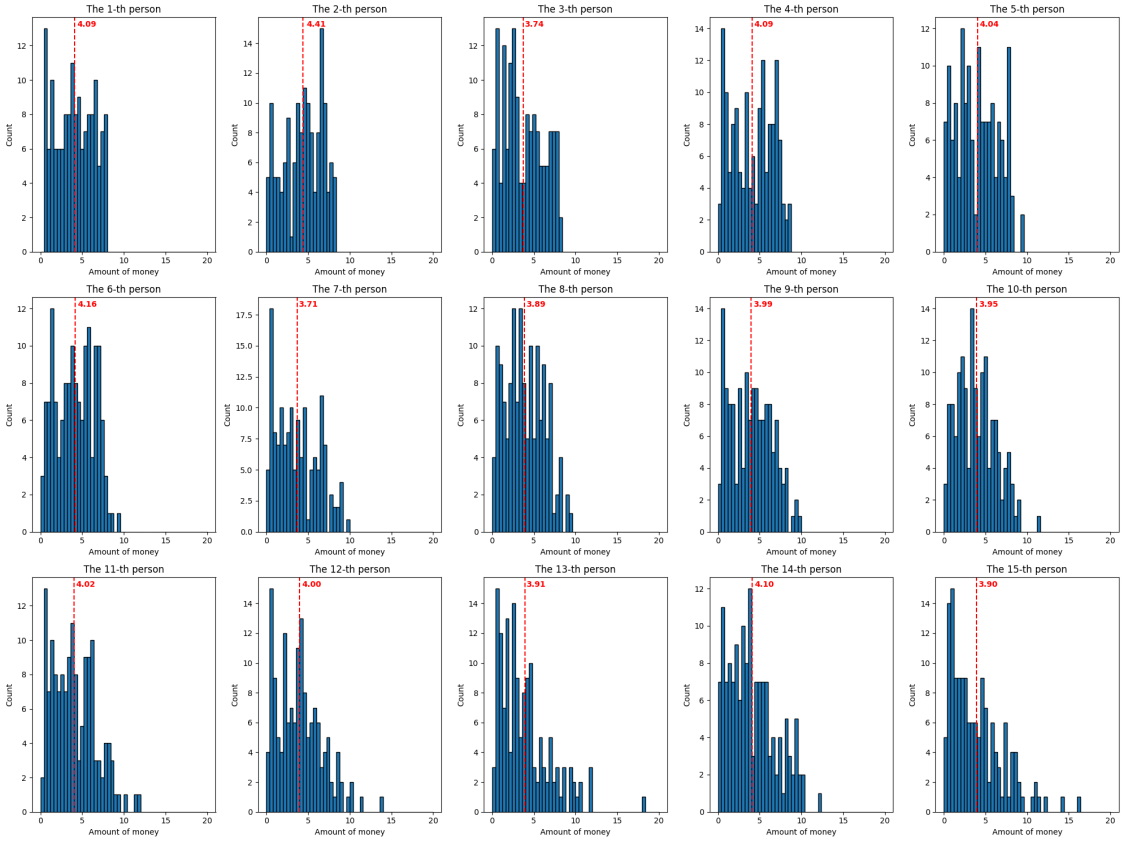
[1]:
```
import pandas as pd

df = pd.read_csv("Data.csv", header=None)
df.columns = df.iloc[0, :]
df = df.drop(0, axis=0)
df = df.set_index(df.columns[0])

df
```

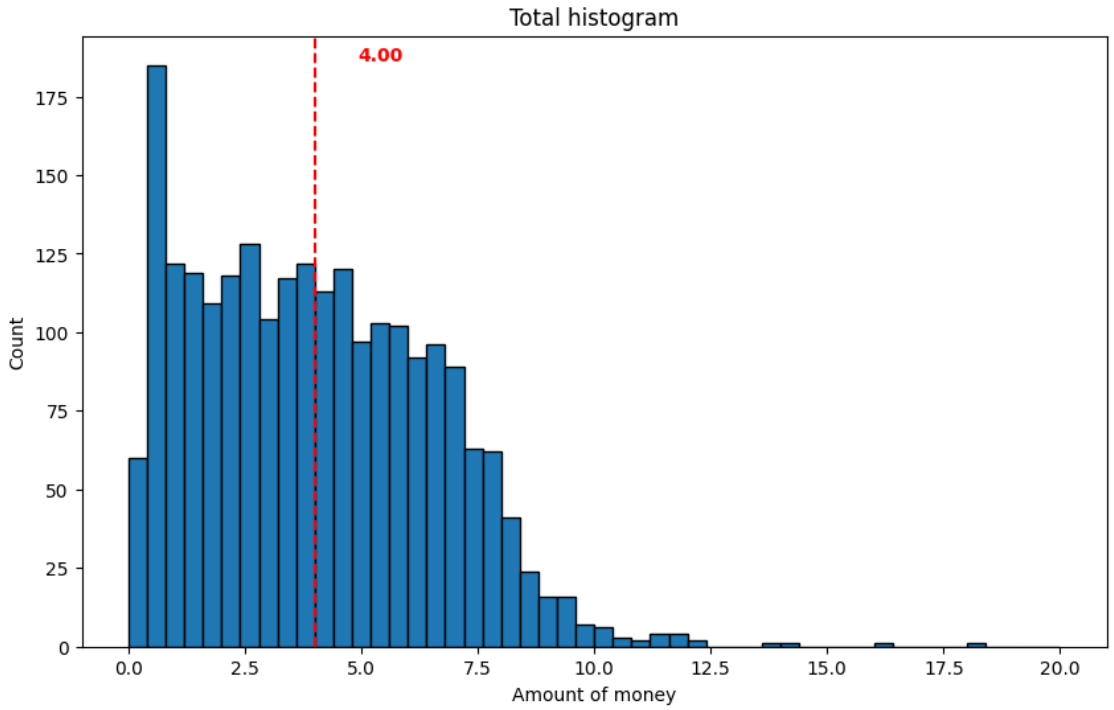
[1]:
```
0          1.0     2.0     3.0     4.0     5.0     6.0     7.0     8.0     9.0  \
Data
1         7.34    4.61    2.79    6.31    2.93    0.80    4.72    4.65    1.27
2         2.45    2.09    3.94    3.23    1.70    6.62    0.44    6.25    0.86
3         6.68    6.38    2.39    3.00    2.40    0.39    0.49    2.09    7.46
4         5.22    6.54    5.60    0.76    2.15    7.14    4.32    1.96    6.36
5         0.40    2.46    8.27    7.19    4.34    7.15    2.16    2.25    6.58
…          …       …       …       …       …       …       …       …       …
147       0.45    5.00    0.54    4.81    0.44    4.90    1.83    2.59    8.26
148       7.80    5.26    0.36    6.05    0.36    6.72    0.58    6.39    5.53
149       2.87    8.13    1.39    0.86    7.91    1.28    7.17    6.48    0.86
150       7.84    0.84    4.41    6.80    6.77    5.29    1.46    5.69    4.03
Total   612.97  661.55  561.73  612.88  605.54  624.36  557.02  582.93  598.73

0         10.0    11.0    12.0    13.0    14.0    15.0      Total
Data
1         6.06    3.64    7.14    1.34    2.29    4.11      60.00
2         2.32   11.40    2.16    6.14    1.42    8.98      60.00
3         7.04    2.14    0.61    8.66    2.86    7.41      60.00
4         4.19    4.56    3.29    0.85    3.05    4.01      60.00
5         1.06    2.64    7.34    4.12    3.61    0.43      60.00
…          …       …       …       …       …       …         …
147       2.43    9.23    5.68    1.83    3.93    8.08      60.00
148       5.77    3.98    5.17    2.67    2.24    1.12      60.00
149       3.50    5.17    6.20    0.69    1.59    5.90      60.00
150       4.90    3.13    0.69    1.91    3.13    3.11      60.00
Total   592.72  603.68  600.56  586.16  614.73  584.44  9,000.00

[151 rows x 16 columns]
```

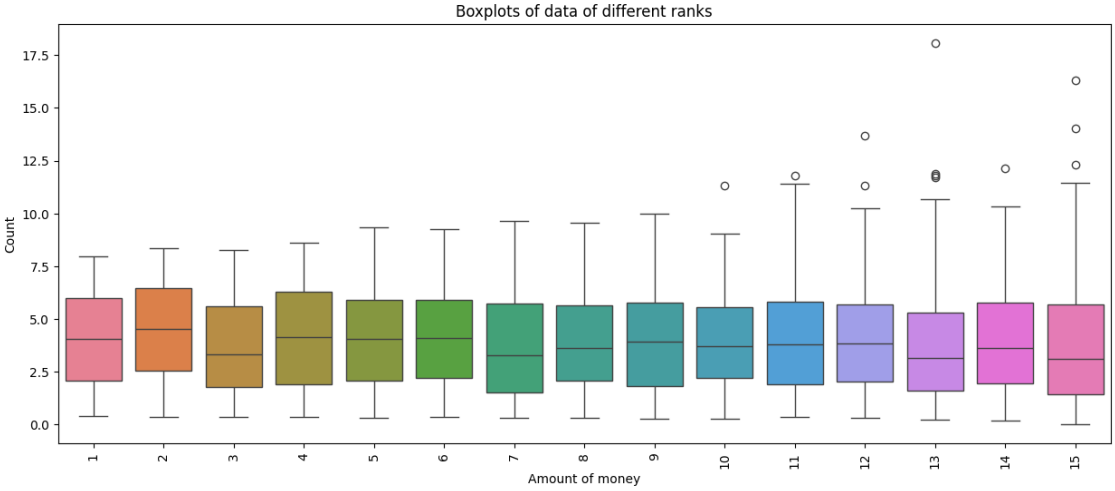
[2]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

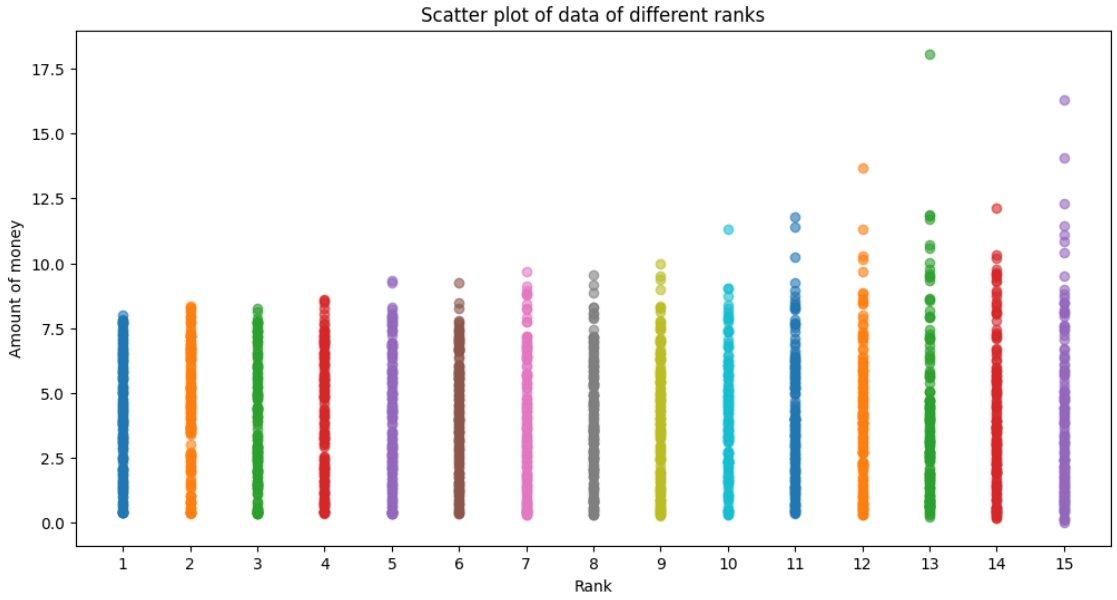
```
df = pd.read_csv("Data.csv", header=None)
df = df.iloc[1:-1, 1:-1]
df = df.apply(pd.to_numeric)

bins = np.linspace(0, 20, 51)
n_cols = min(15, df.shape[1])

plt.figure(figsize=(20, 15))
for i in range(n_cols):
    plt.subplot(3, 5, i+1)
    counts, _, _ = plt.hist(df.iloc[:, i], bins=bins, edgecolor='black')
    plt.axvline(df.iloc[:, i].mean(), color='red', linestyle='dashed',␣
 ↪linewidth=1.5)
    plt.text(df.iloc[:, i].mean() * 1.35, counts.max(), f'{df.iloc[:, i].mean():
 ↪.2f}',
             color='red', ha='center', va='bottom', fontweight='bold')
    plt.title(f'The {i+1}-th person')
    plt.xlabel('Amount of money')
    plt.ylabel('Count')
plt.tight_layout()
plt.show()

plt.figure(figsize=(10,6))
counts, _, _ = plt.hist(df.iloc[:, :n_cols].values.flatten(), bins=bins,␣
 ↪edgecolor='black')
plt.axvline(df.iloc[:, :n_cols].values.flatten().mean(), color='red',␣
 ↪linestyle='dashed', linewidth=1.5)
plt.text(df.iloc[:, :n_cols].values.flatten().mean() * 1.35, counts.max(),␣
 ↪f'{df.iloc[:, :n_cols].values.flatten().mean():.2f}',
        color='red', ha='center', va='bottom', fontweight='bold')
plt.title('Total histogram')
plt.xlabel('Amount of money')
plt.ylabel('Count')
plt.show()

plt.figure(figsize=(15,6))
sns.boxplot(data=df.iloc[:, :n_cols])
plt.title('Boxplots of data of different ranks')
plt.xticks(rotation=90)
plt.xlabel('Amount of money')
plt.ylabel('Count')
plt.show()

plt.figure(figsize=(12,6))
for i in range(n_cols):
    plt.scatter([i+1]*df.shape[0], df.iloc[:, i], alpha=0.6)
```

```
plt.xticks(range(1, n_cols+1))
plt.xlabel('Rank')
plt.ylabel('Amount of money')
plt.title('Scatter plot of data of different ranks')
plt.show()

all_data = df.iloc[:, :n_cols].values.flatten()
total_mean = all_data.mean()
total_var = ((all_data - total_mean) ** 2).mean()

print("Overall statistics:")
print(f"Mean: {total_mean:.2f}")
print(f"Variance: {total_var:.4f}")
```




Total histogram
4.00
Count
Amount of money
0
25
50
75
100
125
150
175
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Boxplots of data of different ranks
Count
Amount of money
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15




```
Overall statistics:
Mean: 4.00
Variance: 6.3688
```

3 Observations

From the data and the plots, we discover:

- The amounts of money that all ranks received are roughly the same;
- The latter one to grab the WRE is more likely to receive the amount of money with a bigger variance.

Based on the observations, we have tried 2 probability models to fit the data.

3.1 Method1: Gamma-Dirichlet Split

Since the Gamma distribution has an excellent additivity, it is easy to set the expected value and the variance. From the observation, we know all the ranks get roughly the same amount of money, so we set the expected values of all ranks the same.

According to the data from the experiment, we set the parameters as follows:

- $sum = 6000$
- $n = 15$
- $\alpha = 33.99$
- $E_i = 400,\ \forall i \in [1, 15]$

α is calculated by the variance of the data. The specific proof of the variance and mean is shown in the appendix: Tested Machanisms.

3.2 Method2: Twice as the Mean

Since the latter one is more likely to receive the amount of money with a bigger variance, we tried a new method: Twice as the Mean.

According to the data from the experiment, we set the parameters as follows:

- $sum = 6000$
- $n = 15$

4 Model Testing

We use Monte Carlo simulation to obtain the theoretical frequency within each money range by the above two methods. We do the Chi-Square test and Kolmogorov-Smirnov test in the following code:

[3]:
```
from scipy.stats import ks_2samp, chisquare
from utils import logger
from utils import np
from utils import json2dict

S = 6000
EXP = 5
PathExpr_2e = "results/TwiceAsTheMean/stats.json"
PathExpr_gd = "results/GammaDirichlet-fitted-Eequal/stats.json"
PathReal = "Data.json"

def dict2listT(d: dict, S: int) -> np.ndarray:
    res = np.zeros(S, dtype=int)
    for k, v in d.items():
        res[int(k)-1] = v
    return res

def dict2listA(d: dict) -> np.ndarray:
    return np.repeat(
        np.fromiter((int(k) for k in d.keys()), dtype=int),
        np.fromiter(d.values(), dtype=int)
    )

def mergeBins(real, expr, exp: int) -> tuple[np.ndarray, np.ndarray]:
    real = real.copy()
    expr = expr.copy()
    newReal = []
    newExpr = []
    sReal = 0
```

```
    sExpr = 0
    for r, e in zip(real[::-1], expr[::-1]):
        sReal += r
        sExpr += e
        if sExpr >= exp:
            newReal.append(sReal)
            newExpr.append(sExpr)
            sReal = 0
            sExpr = 0
    if sExpr > 0:
        newReal.append(sReal)
        newExpr.append(sExpr)
    return (np.array(newReal[::-1]), np.array(newExpr[::-1]))

def subsample(freq: np.ndarray, N: int) -> np.ndarray:
    return np.random.choice(freq, size=N, replace=False)

def ksTest(expr: dict, real: dict, S: int) -> tuple[float, float]:
    expr = dict2listA(expr)
    real = dict2listA(real)
    expr = subsample(expr, min(len(expr), len(real)))
    real = subsample(real, min(len(expr), len(real)))
    # logger.info(f"KS testing...")
    stat, pvalue = ks_2samp(expr, real)
    return stat, pvalue

def chiSquareTest(expr: dict, real: dict, S: int, exp: int) -> tuple[float,␣
 ↪float]:
    expr = dict2listT(expr, S)
    real = dict2listT(real, S)
    # logger.info(f"Chi-square testing...")
    expr = expr/expr.sum()*real.sum()
    newReal, newExpr = mergeBins(real, expr, exp)
    stat, pvalue = chisquare(f_obs=newReal, f_exp=newExpr)
    return stat, pvalue


if __name__ == "__main__":
    for PathExpr, Machanism in [(PathExpr_gd, "Method1: Gamma-Dirichlet␣
 ↪Split"), (PathExpr_2e, "Method2: Twice as the mean")]:
        expr = json2dict(PathExpr)['freq']
        real = json2dict(PathReal)['freq']
        KSstat, KSpvalue = ksTest(expr, real, S)
        CHIstat, CHIpvalue = chiSquareTest(expr, real, S, EXP)
        print(Machanism + ":")
        print(f"KS test:")
        print(f"  statistic={KSstat}, pvalue={KSpvalue}")
```

```
        print(f"Chi-square test:")
        print(f"  statistic={CHIstat}, pvalue={CHIpvalue}")
        print("-"*80)
```

```
Method1: Gamma-Dirichlet Split:
KS test:
  statistic=0.07555555555555556, pvalue=5.2321728698241065e-06
Chi-square test:
  statistic=1016.3797734262617, pvalue=2.747082920630171e-69
--------------------------------------------------------------------------------
Method2: Twice as the mean:
KS test:
  statistic=0.037333333333333336, pvalue=0.08690273448520303
Chi-square test:
  statistic=588.1426501249248, pvalue=8.748675778849906e-11
--------------------------------------------------------------------------------
```

From the data, we can conclude that:

- Method 1: **Gamma-Dirichlet Split**
  - **KS test**: statistic $= 0.071$, p-value $2.62 \cdot 10^{-5}$, which means the distribution differs from the reference.
  - **Chi-square test**: statistic 1016.38, p-value $2.75 \cdot 10^{-69}$, which means significant difference from the expected frequencies.

  So method 1 produces a distribution that deviates considerably from the distribution measured by the experiment.
- Method 2: **Twice as the Mean**
  - **KS test**: statistic $= 0.037$, p-value 0.087 , which indicates no significant difference at the 5% level.
  - **Chi-square test**: statistic 588.14, p-value $8.75 \cdot 10^{-11}$ which indicates some deviation in categorical frequencies, but smaller than Method 1.

  So method 2 aligns much better with the distribution measured by the experiment even if some minor deviations remain.

Above all, **method 2: Twice as the Mean** is closer to the intended distribution.

5 From Modeling to Decision-Making Policy

From the Twice as the Mean Model, we know that the later one grabs the WRE, the greater the variance in the amount won will be, and the more exciting the result will be. (*The specific proof is shown in the appendix 1: Tested Machanisms*)

So in practice, we suggest:

- If the number of people who can get the WRE is far smaller than the number of people in the group, grab as quickly as you can to ensure you grab the WRE successfully;

- If you aim to get a more steady amount of money, grab earlier;
- If you aim to get more than two times of expectation, grab later.

6 Further explorations

6.1 A "user-specific" WRE machanism

- We have invented a new machanism entitled DIE (Divide Integers Exponentially) Machanism. This is a machanism that supports different expectations specified by each person (with a sum of the total amount of money, of course) and also a controllable variance.
  - The variance of the machanism is usually larger than other machanisms, so it can be applied to more thrilling scenarios of lucky WREs, which may be attractive to the youth.
  - *The specific implementation and proof is shown in the appendix 1: Tested Machanisms.*
- As mentioned above, Gamma-Dirichlet Split Machanism has a even better variance adjustability, which can also be used in the user-specific condition.
  - We can manually adjust the variance to meet the needs of different scenarios. For example, we can set a larger α to get a smaller variance, which may leave every young children who receives the WRE happiness.
  - *The specific implementation and proof is shown in the appendix 1: Tested Machanisms.*
- Also, we have found a new machanism entitled Mean-Determined Split. This is also a machanism easy to adjust the variance.
  - *The specific implementation and proof is shown in the appendix 1: Tested Machanisms.*

6.2 A "fairness-ware" WRE machanism

- The DIE (Divide Integers Exponentially) Machanism, the Gamma-Dirichlet Split Machanism and the Mean-Determined Split Machanism all support the adjustment of expectation respectively for each person. We can dynamically adjust the expectation to compensate unlucky users in a group for fairness.
  - For example, maintain a variable to record the difference between the current amount and the expected amount for everyone, and multiply the expectation by 1.3 when someone is unlucky enough, and normalize the expectation of everyone to 1. Then, she/he will have a larger probability to receive a bigger WRE.
- Also, since the allocations are determined at the moment of sending out the WRE, we can set a higher probability to those who keeps receiving money less than the expectation. That can also compensate unlucky users in a group for fairness.

7 Some Comments on Some Requirements of the Project

- It is of limited methodological value to perform generative modeling on the experimental data solely for the purpose of generating new samples to verify the underlying distribution. Since the observed data may already contain sampling biases, any generative model trained on such data will inevitably inherit and potentially amplify these biases. As a result, the generated

samples cannot provide a more faithful representation of the true latent distribution than the original observations.

- Whether the analysis is conducted on WRE data inside or outside China, on workdays or holidays, or on QQ Red Packet data, the fundamental research paradigm remains the same. These scenarios differ only in context rather than in methodology, and our work has already fully established this unified analytical framework.

8 Acknowledgments

- Chatgpt and Gemini have helped us a lot with the repeated parts of code work.
- The Twice as the Mean Machanism is inspired by https://juejin.cn/post/7517116929376305203.
- The other four groups made contributions limited in data collection.

9 Appendix 1: Tested Machanisms

9.1 Uniform Random Permutation

9.1.1 Problem Description

Distribute a total sum among n recipients such that every ordered combination summing to the total is equally likely to be chosen. This method results in a Uniform random permutation of the distributed amounts.

9.1.2 Calculation of Expectation

Let the random vector $(X_1, \dots, X_n)$ represent a solution to the equation $\sum_{i=1}^{n} X_i = S$ where $X_i \geq 1$, with all solutions being equally likely. Let $Y_i = X_i - 1$. Then $Y_i \geq 0$ and the equation becomes:

$$\sum_{i=1}^{n} Y_i = S - n = M$$

By symmetry, the expected values of all variables are equal:

$$E\left[\sum_{i=1}^{n} Y_i\right] = \sum_{i=1}^{n} E[Y_i] = nE[Y_i] = M$$

$$\therefore E[Y_i] = \frac{M}{n}$$

9.1.3 Calculation of Variance

We calculate the expectation of the binomial coefficient $E[\binom{Y_i}{2}]$. The probability that Y_1 takes the value k is:

$$P(Y_1 = k) = \frac{\binom{(M-k)+(n-1)-1}{(n-1)-1}}{\binom{M+n-1}{n-1}} = \frac{\binom{M-k+n-2}{n-2}}{\binom{M+n-1}{n-1}}$$

The expectation is given by:

$$E\left[\binom{Y_1}{2}\right] = \sum_{k=0}^{M} \binom{k}{2} P(Y_1 = k) = \frac{1}{\binom{M+n-1}{n-1}} \sum_{k=0}^{M} \binom{k}{2} \binom{M+n-2-k}{n-2}$$

Using the Parallel Summation Identity: $\sum_{i=r}^{n} \binom{i}{r}\binom{n-i}{s} = \binom{n+1}{r+s+1}$, where $r = 2$ and $s = n-2$, we get:

$$\sum_{k=0}^{M} \binom{k}{2} \binom{M+n-2-k}{n-2} = \binom{M+n-1}{n+1}$$

Substituting this back into the expectation formula:

$$E\left[\binom{Y_1}{2}\right] = \frac{\binom{M+n-1}{n+1}}{\binom{M+n-1}{n-1}} = \frac{\frac{(M+n-1)!}{(n+1)!(M-2)!}}{\frac{(M+n-1)!}{(n-1)!M!}} = \frac{M(M-1)}{n(n+1)}$$

$$E[Y_1(Y_1-1)] = 2E\left[\binom{Y_1}{2}\right] = \frac{2M(M-1)}{n(n+1)}$$

Using the formula $Var(Y) = E[Y(Y-1)] + E[Y] - (E[Y])^2$:

$$\begin{aligned}
Var(Y_i) &= \frac{2M(M-1)}{n(n+1)} + \frac{M}{n} - \left(\frac{M}{n}\right)^2 \\
&= \frac{M}{n}\left[\frac{2(M-1)}{n+1} + 1 - \frac{M}{n}\right] \\
&= \frac{M}{n}\left[\frac{2n(M-1) + n(n+1) - M(n+1)}{n(n+1)}\right] \\
&= \frac{M}{n^2(n+1)}\left[2nM - 2n + n^2 + n - Mn - M\right] \\
&= \frac{M}{n^2(n+1)}\left[Mn - M + n^2 - n\right] \\
&= \frac{M}{n^2(n+1)}\left[M(n-1) + n(n-1)\right] \\
&= \frac{M(M+n)(n-1)}{n^2(n+1)}
\end{aligned}$$

Substitute $M = S - n$ (noting that $M + n = S$):

$$Var(X_i) = \frac{(S-n)S(n-1)}{n^2(n+1)}$$

9.1.4 Code

[4]:
```python
from utils import logger
from utils import rnd

def UniformRandom(S: int, n: int) -> list[int]:
    if n<=0:
        raise ValueError(f"n must be positive, got {n}")
    if S<n:
        raise ValueError(f"S must be at least n, got S={S}, n={n}")

    # logger.info(f"run UniformRandom with S={S}, n={n}")

    cuts = rnd.sample(range(1, S), n-1)
    cuts.sort()
    res = []
    lst = 0
    for i in cuts:
        res.append(i-lst)
        lst=i
    res.append(S-lst)
    return res
```

9.2 Mean-determined Split

9.2.1 Problem Description

Distribute a total sum S among n recipients such that the expected value of the i-th recipient equals a target value E_i. The algorithm allocates a base amount of 1 to each recipient and distributes the remaining amount $M = S - n$ in discrete steps of size m (and a final remainder step).

9.2.2 Calculation of Expectation

Let X_i be the random variable representing the final amount allocated to recipient i. The algorithm initializes $X_i = 1$. Let $M = S-n$ be the remaining sum to be distributed. We define the probability weight for each recipient i as:

$$p_i = \frac{E_i - 1}{S - n} = \frac{E_i - 1}{M}$$

Note that $\sum p_i = \frac{\sum E_i - n}{S-n} = \frac{S-n}{S-n} = 1$, so this is a valid probability distribution. The distribution process consists of $K = \lfloor \frac{M}{m} \rfloor$ steps of size m, and one final step of size $r = M \mod m$. Let $I_{i,k}$ be an indicator variable for the k-th step of size m, and J_i be the indicator variable for the remainder step of size r.

$$X_i = 1 + \sum_{k=1}^{K} (m \cdot I_{i,k}) + (r \cdot J_i)$$

Since each step chooses a recipient independently based on weights p_i:

$$E[I_{i,k}] = p_i \quad \text{and} \quad E[J_i] = p_i$$

By linearity of expectation:

$$\begin{aligned} E[X_i] &= 1 + \sum_{k=1}^{K} mE[I_{i,k}] + rE[J_i] \\ &= 1 + \sum_{k=1}^{K} mp_i + rp_i \\ &= 1 + p_i(K \cdot m + r) \end{aligned}$$

Since $K \cdot m + r = M$:

$$E[X_i] = 1 + p_i M = 1 + \frac{E_i - 1}{M} M = E_i$$

Thus, the expected value matches the target configuration.

9.2.3 Calculation of Variance

Since the selection in each step is independent using `rnd.choices` function, the variance of the sum is the sum of the variances. The variance of a single Bernoulli trial scaled by constant c with probability p is $c^2 p(1-p)$. The variance of the amount added in the main loop (Steps 1 to K) for recipient i is:

$$Var_{main} = \sum_{k=1}^{K} Var(m \cdot I_{i,k}) = K \cdot m^2 \cdot p_i(1-p_i)$$

The variance of the amount added in the remainder step is:

$$Var_{rem} = Var(r \cdot J_i) = r^2 \cdot p_i(1-p_i)$$

The total variance for recipient i is:

$$Var(X_i) = Var_{main} + Var_{rem} = p_i(1-p_i)(Km^2 + r^2)$$

By substituting the following parameters, we obtain: $p_i = \frac{E_i - 1}{M}$ $M = S - n$ $K = \lfloor \frac{S-n}{m} \rfloor$ $r = (S-n) \mod m$

$$Var(X_i) = \frac{(E_i - 1)(S - n - E_i + 1)}{(S-n)^2} (\lfloor \frac{S-n}{m} \rfloor m^2 + ((S-n) \mod m)^2)$$

9.2.4 Code

[5]:
```python
from utils import logger
from utils import rnd

def MeanDetermined(S: int, n: int, m: int, E: list[float]) -> list[int]:
    if n<=0:
        raise ValueError(f"n must be positive, got {n}")
    if S<n:
        raise ValueError(f"S must be at least n, got S={S}, n={n}")
    if E:
        if len(E)!=n:
            raise ValueError(f"Length of E must be n={n}, got {len(E)}")
        if any(e<1 for e in E):
```

```
            raise ValueError("All expected values must be at least 1")
        if abs(sum(E)-S) > 1e-6:
            raise ValueError(f"Sum of expected values must equal S={S}, got␣
↪sum(E)={sum(E)}")
    else:
        E = [S/n]*n

    # logger.info(f"run MeanDetermined with S={S}, n={n}, E={E}")

    res = [1]*n
    S -= n
    w = [(e-1)/S for e in E]
    indicies = list(range(n))
    for _ in range(S//m):
        i = rnd.choices(indicies, w)[0]
        res[i] += m
    i = rnd.choices(indicies, w)[0]
    res[i] += S % m
    return res
```

9.3 DIE (Divide Integers Exponentially) Machanism

9.3.1 Correctness of DIE Algorithm with dynamic step

Problem Description Distribute total sum S among n recipients with target expected values $E = [\mu_1, \dots, \mu_n]$. Let X_i be the random variable for the amount allocated to recipient i. Constraint: $\sum X_i = S$, $X_i \geq 1$, and $E[X_i] = \mu_i$.

Calculation of Expectation Initialize $X_i = 1$. Let remaining sum $M = S - n$. The algorithm distributes M via a sequence of independent atomic additions. Let Y_i be the accumulated value for recipient i from M, so $X_i = 1 + Y_i$. In every atomic step, recipient i is selected with fixed probability p_i:

$$p_i = \frac{\mu_i - 1}{\sum_{j=1}^{n}(\mu_j - 1)} = \frac{\mu_i - 1}{S - n}$$

Since expectation is linear:

$$E[Y_i] = M \cdot p_i = (S - n) \cdot \frac{\mu_i - 1}{S - n} = \mu_i - 1$$
$$E[X_i] = 1 + E[Y_i] = \mu_i$$

Calculation of Variance The algorithm decomposes M into a set of atomic operations Ω. For each operation $k \in \Omega$, a value v_k is added to a recipient chosen independently with probability p_i. Let $I_k^{(i)}$ be the indicator variable that recipient i is chosen in step k:

$$I_k^{(i)} \sim \text{Bernoulli}(p_i) \implies Var(I_k^{(i)}) = p_i(1 - p_i)$$

The total allocation is $Y_i = \sum_{k \in \Omega} v_k I_k^{(i)}$. Due to independence:

$$Var(X_i) = Var(Y_i) = \sum_{k \in \Omega} v_k^2 Var(I_k^{(i)}) = p_i(1 - p_i) \sum_{k \in \Omega} v_k^2$$

Define the Sum of Squared Increments $\mathcal{K}$ as follows:

$$\mathcal{K} = \sum_{\text{blocks}} \sum_{\text{steps } j} \text{count}_j \cdot (\text{val}_j)^2$$

Substitute p_i:

$$Var(X_i) = \frac{\mu_i - 1}{S - n} \left(1 - \frac{\mu_i - 1}{S - n}\right) \mathcal{K}$$

$$Var(X_i) = \frac{(\mu_i - 1)(S - n - \mu_i + 1)}{(S - n)^2} \mathcal{K}$$

9.3.2 Correctness of DIE Algorithm with Constant Step

Problem Description Distribute a total sum S among n recipients into m blocks using a base-q decomposition method. The algorithm ensures the final sum is exactly S, and the expected allocation for each recipient i matches a target E_i. The adjustable amount $M = S - n$ is distributed via independent weighted random trials based on the digit representation of M. #### Calculation of Expectation Let X_i be the random variable for the amount allocated to recipient i. Since every recipient starts with a base value of 1, let $Y_i = X_i - 1$. The allocation probability for recipient i is normalized as $p_i = \frac{E_i - 1}{S - n} = \frac{E_i - 1}{M}$. The total adjustable sum M is split into m blocks: $M = \sum_{j=1}^{m} B_j$. Each block B_j is decomposed in base q such that $B_j = \sum_k d_{j,k} q^k$, where $d_{j,k}$ is the digit at position k. The algorithm performs $d_{j,k}$ independent trials for each position, adding q^k upon success. By linearity of expectation:

$$E[Y_i] = \sum_{j=1}^{m} \sum_k d_{j,k} \cdot q^k \cdot p_i = p_i \sum_{j=1}^{m} \left(\sum_k d_{j,k} q^k \right) = p_i \sum_{j=1}^{m} B_j = p_i M$$

Substituting p_i:

$$E[X_i] = 1 + \frac{E_i - 1}{M} \cdot M = E_i$$

Calculation of Variance Allocations are independent across blocks and digit positions. For a single trial at digit position k with value q^k, the variance is $(q^k)^2 Var(\text{Bernoulli}(p_i)) = q^{2k} p_i (1 - p_i)$. The total variance is the sum of variances of all independent trials across all m blocks. Define the Structural Constant Λ as the sum of squared-base weights:

$$\Lambda = \sum_{j=1}^{m} \sum_k d_{j,k} q^{2k}$$

The variance for recipient i is:

$$Var(X_i) = Var(Y_i) = p_i (1 - p_i) \Lambda$$

Substitute $p_i = \frac{E_i - 1}{S - n}$:

$$Var(X_i) = \frac{E_i - 1}{S - n} \left(1 - \frac{E_i - 1}{S - n}\right) \Lambda$$

Simplifying the term:

$$Var(X_i) = \frac{(E_i - 1)(S - n - E_i + 1)}{(S-n)^2}\Lambda = \frac{(E_i - 1)(S - E_i)}{(S-n)^2}\Lambda$$

[ ]:
```python
from utils import logger
from utils import rnd

def DyStep(S):
    q = max(2,int(np.log2(S)))
    return S%q, S//q, q

def step3(S):
    q = 3
    return S%q, S//q, q

def DIE_core(S: int, n: int, E: list[float], myStep: callable) -> list[int]:
    if n<=0:
        raise ValueError(f"n must be positive, got {n}")
    if E:
        if len(E)!=n:
            raise ValueError(f"Length of E must be n={n}, got {len(E)}")
        if any(e<0 for e in E):
            raise ValueError("All expected values must be non-negative")
        if abs(sum(E)-S) > 1e-6:
            raise ValueError(f"Sum of expected values must equal S={S}, got␣
 ↪sum(E)={sum(E)}")
    else:
        E = [S/n]*n

    # logger.debug(f"run DIE(core) with S={S}, n={n}, E={E}")

    res = [0]*n
    w = [e/S for e in E]
    val = 1
    indices = list(range(n))
    while S>0:
        x, S, q = myStep(S)
        for _ in range(x):
            i = rnd.choices(indices, w)[0]
            res[i] += val
        val *= q
    return res

def DIE(S: int, n: int, m: int, E: list[float], myStep: callable) -> list[int]:
    if m<=0:
        raise ValueError(f"m must be positive, got m={m}")
```

```
    if S<n:
        raise ValueError(f"S must be at least n, got S={S}, n={n}")

    # logger.info(f"run DIE with S={S}, n={n}, m={m}, E={E}")

    res = [1]*n
    S -= n
    E = [e - 1 for e in E] if E else None
    blk = S//m
    for t in range(m):
        if t==m-1:
            blk += S%m
        par = DIE_core(blk, n, [e*blk/S for e in E] if E else None, myStep)
        for i in range(n):
            res[i] += par[i]
    return res
```

9.4 Gamma-Dirichlet Split Machanism

9.4.1 Problem Description

Distribute a sum S among n recipients given target expectations $\mathbf{E} = (E_1, \dots, E_n)$. The algorithm uses a Gamma-Dirichlet split controlled by parameter α followed by randomized rounding. Constraints: $X_i \in \mathbb{Z}^+$, $\sum X_i = S$. Transformation: Let $Y_i = X_i - 1$ and $M = S - n$. Target shifted means are $\mu_i = E_i - 1$.

9.4.2 Calculation of Expectation

Let $\mathbf{C} = (C_1, \dots, C_n)$ be the continuous allocation before rounding, where $C_i = M \cdot P_i$. The proportions $\mathbf{P} \sim Dir(\alpha_1, \dots, \alpha_n)$ with parameters $\alpha_i = \alpha \frac{\mu_i}{M}$. Note $\sum \alpha_i = \alpha$.

$$E[C_i] = M \cdot E[P_i] = M \cdot \frac{\alpha_i}{\sum \alpha_j} = M \cdot \frac{\alpha \mu_i / M}{\alpha} = \mu_i$$

The integer allocation is $Y_i = \lfloor C_i \rfloor + \Delta_i$. The rounding step distributes the residual $K = M - \sum \lfloor C_j \rfloor$ using Multinomial sampling with weights equal to fractional parts $f_i = C_i - \lfloor C_i \rfloor$. The conditional expectation of the rounding increment Δ_i given the continuous realization is:

$$E[\Delta_i | \mathbf{C}] = K \cdot \frac{f_i}{\sum f_j} = K \cdot \frac{f_i}{K} = f_i$$

Using the Law of Total Expectation:

$$E[Y_i] = E[\lfloor C_i \rfloor + E[\Delta_i | \mathbf{C}]] = E[\lfloor C_i \rfloor + f_i] = E[C_i] = \mu_i$$

$$\therefore E[X_i] = E[Y_i] + 1 = \mu_i + 1 = E_i$$

9.4.3 Calculation of Variance

We decompose variance using the Law of Total Variance: $Var(Y_i) = Var(E[Y_i|\mathbf{C}]) + E[Var(Y_i|\mathbf{C})]$. Since $E[Y_i|\mathbf{C}] = C_i$, the first term is the variance of the Dirichlet split, and the second is the

rounding noise. 1. Variance of Continuous Split (Var_{Dir}) For $P_i \sim Beta(\alpha_i, \alpha - \alpha_i)$, the variance is:

$$Var(P_i) = \frac{\frac{\alpha_i}{\alpha}(1 - \frac{\alpha_i}{\alpha})}{\alpha + 1} = \frac{\frac{\mu_i}{M}(1 - \frac{\mu_i}{M})}{\alpha + 1}$$

Scaling by M:

$$Var_{Dir} = Var(C_i) = M^2 Var(P_i) = \frac{\mu_i(M - \mu_i)}{\alpha + 1}$$

2. Expectation of Rounding Noise (Var_{Rnd}) Given $\mathbf{C}$, Δ_i follows a Binomial distribution $B(K, p_i)$ with probability $p_i = f_i/K$:

$$Var(Y_i|\mathbf{C}) = Var(\Delta_i|\mathbf{C}) = Kp_i(1 - p_i) = f_i(1 - \frac{f_i}{K})$$

Taking the expectation over the continuous space:

$$Var_{Rnd} = E\left[f_i\left(1 - \frac{f_i}{K}\right)\right]$$

Note that $0 \leq f_i < 1$, so this term is strictly bounded by 0.25 and typically negligible for large S. 3. Total Variance Substituting $\mu_i = E_i - 1$ and $M = S - n$:

$$Var(X_i) = \frac{(E_i - 1)(S - n - E_i + 1)}{\alpha + 1} + \epsilon_{round}$$

$$Var(X_i) \approx \frac{(E_i - 1)(S - E_i - n + 1)}{\alpha + 1}$$

9.4.4 Code

[7]:
```
from utils import logger
from utils import rnd
import math

def GammaDirichlet(S: int, n: int, alpha: float, E: list[float]) -> list[int]:
    if n <= 0:
        raise ValueError(f"n must be positive, got {n}")
    if S < n:
        raise ValueError(f"S must be at least n, got S={S}, n={n}")
    if E:
        if len(E)!=n:
            raise ValueError(f"Length of E must be n={n}, got {len(E)}")
        if any(e<1 for e in E):
            raise ValueError("All expected values must be at least 1")
        if abs(sum(E)-S) > 1e-6:
            raise ValueError(f"Sum of expected values must equal S={S}, got␣
 ↪sum(E)={sum(E)}")
    else:
        E = [S/n]*n
    if alpha <= 0:
        raise ValueError(f"alpha must be positive, got alpha={alpha}")
```

```
    # logger.info(f"run GammaDirichlet with S={S}, n={n}, alpha={alpha}, E={E}")

    base = [1]*n
    S = S-n
    E = [e-1 for e in E]
    samples = [rnd.gammavariate(alpha * e / S, 1.0) if e > 0 else 0.0 for e in␣
↪E]
    gamma = sum(samples)
    prop = [x/gamma for x in samples]
    scaled = [p*S for p in prop]
    res = [math.floor(x) for x in scaled]
    frac = [x-r for x, r in zip(scaled, res)]
    K = S-sum(res)
    indices = list(range(n))
    if K>0:
        I = rnd.choices(indices, frac, k=K)
        for i in I:
            res[i] += 1
    return [b+a for b, a in zip(base, res)]
```

9.5 Twice as the Mean Machanism

9.5.1 Problem Description

Distribute a total sum S among n recipients. Initialize each recipient with a base amount of 1. The remaining sum to distribute is $M = S - n$. For each recipient i (where $i = 1, \ldots, n$), let $k = n - (i-1)$ be the number of remaining recipients, and R_i be the remaining undistributed sum ($R_1 = M$). The algorithm assigns an additional amount Y_i drawn from a uniform distribution:

$$Y_i \sim U\left[0, 2 \cdot \frac{R_i}{k}\right]$$

The remaining sum updates as $R_{i+1} = R_i - Y_i$. The final amount for recipient i is $X_i = 1 + Y_i$.

9.5.2 Calculation of Expectation

We prove that the expected amount for every recipient is equal. Let E_i denote the expectation with respect to the i-th step, conditional on the previous state R_i. The expected value of the random allotment Y_i given the current remaining sum R_i is the mean of the uniform distribution:

$$E[Y_i|R_i] = \frac{1}{2}\left(0 + \frac{2R_i}{n-i+1}\right) = \frac{R_i}{n-i+1}$$

The expected remaining sum for the next step is:

$$E[R_{i+1}|R_i] = R_i - E[Y_i|R_i] = R_i - \frac{R_i}{n-i+1} = R_i \cdot \frac{n-i}{n-i+1}$$

By the Law of Iterated Expectations, the unconditional expected remaining sum at step i is:

$$E[R_i] = M \cdot \prod_{j=1}^{i-1} \frac{n-j}{n-j+1} = M \cdot \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdots \frac{n-i+1}{n-i+2} = M \cdot \frac{n-i+1}{n}$$

Substituting $E[R_i]$ back into the expectation of Y_i:

$$E[Y_i] = E\left[\frac{R_i}{n-i+1}\right] = \frac{1}{n-i+1}E[R_i] = \frac{1}{n-i+1} \cdot \frac{M(n-i+1)}{n} = \frac{M}{n}$$

Thus, the total expected amount for each recipient is:

$$E[X_i] = 1 + E[Y_i] = 1 + \frac{S-n}{n} = \frac{S}{n}$$

9.5.3 Calculation of Variance

Unlike the Uniform Random method, the variance here depends on the order of distribution. We calculate $Var(Y_i)$. First, determine the second moment $E[Y_i^2]$. For a uniform variable $U \sim [0, 2\mu]$, $E[U^2] = \frac{4}{3}\mu^2$.

$$E[Y_i^2|R_i] = \frac{4}{3}\left(\frac{R_i}{n-i+1}\right)^2$$

Taking the total expectation:

$$E[Y_i^2] = \frac{4}{3(n-i+1)^2}E[R_i^2]$$

We need the recurrence relation for the second moment of the remaining sum $E[R_i^2]$.

$$R_{i+1}^2 = (R_i - Y_i)^2 = R_i^2 - 2R_iY_i + Y_i^2$$

Conditioning on R_i:

$$\begin{aligned} E[R_{i+1}^2|R_i] &= R_i^2 - 2R_iE[Y_i|R_i] + E[Y_i^2|R_i] \\ &= R_i^2 - \frac{2R_i^2}{n-i+1} + \frac{4R_i^2}{3(n-i+1)^2} \\ &= R_i^2\left[1 - \frac{2}{n-i+1} + \frac{4}{3(n-i+1)^2}\right] \end{aligned}$$

Let $k = n-i+1$. The recurrence factor is $\rho(k_j) = \frac{3k^2-6k+4}{3k^2}$.

$$E[R_{j+1}^2] = \rho(k_j) \cdot E[R_j^2]$$

The term R_i is the result of $i-1$ updates starting from $R_1 = M$. We expand the recurrence:

$$E[R_i^2] = M^2 \cdot \prod_{j=1}^{i-1}\rho(k_j) = M^2 \cdot \prod_{j=1}^{i-1}\frac{3(n-j+1)^2 - 6(n-j+1) + 4}{3(n-j+1)^2} = M^2 \cdot \prod_{m=n-i+2}^{n}\frac{3m^2-6m+4}{3m^2}$$

Using the relation $X_i = Y_i + 1$ and

$$Var(X_i) = Var(Y_i) = E[Y_i^2] - (E[Y_i])^2 = \left[\frac{4M^2}{3(n-i+1)^2}\prod_{m=n-i+2}^{n}\left(\frac{3m^2-6m+4}{3m^2}\right)\right] - \frac{M^2}{n^2}$$

Substitute $M = S-n$ to get the final complete expression:

$$Var(X_i) = (S-n)^2\left[\frac{4}{3(n-i+1)^2}\left(\prod_{m=n-i+2}^{n}\frac{3m^2-6m+4}{3m^2}\right) - \frac{1}{n^2}\right]$$

9.5.4 Code

[1]:
```python
from utils import logger
from utils import rnd

def TwiceAsTheMean(S: int, n: int) -> list[int]:
    if n<=0:
        raise ValueError(f"n must be positive, got {n}")
    if S<n:
        raise ValueError(f"S must be at least n, got S={S}, n={n}")

    # logger.info(f"run TwiceAsTheMean with S={S}, n={n}")

    res = [1]*n
    S -= n
    for i in range(n):
        mean = S/(n-i)
        add = rnd.randint(0, int(2*mean))
        res[i] += add
        S -= add
    return res
```

10 Appendix 2: Test Cases and Results

We have tested totally 16 cases and it's too long to show them all in the main part.

All these test cases include:

- Uniform Random Permutation:
  - 6000 cents for 15 people with equal expectation
- Mean-determined Split with step $m = 1$:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of $100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700$ cents respectively
- Mean-determined Split with step $m = 10$:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of $100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700$ cents respectively
- DIE (Divide Integers Exponentially) with ternary step:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of $100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700$ cents respectively
- DIE (Divide Integers Exponentially) with dynamic step with a $min\{2, log_2 S\}$ number system, where S denotes the remaining cents:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of $100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700$ cents respectively
- Gamma-Dirichlet Split with $\alpha = 10$:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of

100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700 cents respectively

- Gamma-Dirichlet Split with $\alpha = 100$:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of
    100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700 cents respectively
- Gamma-Dirichlet Split with $\alpha = 1000$:
  - 6000 cents for 15 people with equal expectation
  - 6000 cents for 15 people with expectation of
    100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600, 600, 700 cents respectively
- Twice as the Mean:
  - 6000 cents for 15 people with equal expectation

Here is the code used to test our machanisms:

[9]:
```
import matplotlib.pyplot as plt

from utils import logger
from utils import np, os
from utils import saveStats, analyze, json2dict

from StatsTester import ksTest, chiSquareTest

BASE_DIR = "results"
DATA_PATH = "Data.json"
EXP = 5
REPEAT = 10000
FIXED_E = [100, 200, 200, 300, 300, 300, 400, 400, 400, 500, 500, 500, 600,␣
 ↪600, 700]
ALLOCATORS = {
    "DIE": lambda **kw: DIE(
        kw["S"],
        kw["n"],
        kw["m"],
        kw.get("E"),
        kw["step"]
    ),
    "UniformRandom": lambda **kw: UniformRandom(
        kw["S"],
        kw["n"]
    ),
    "MeanDetermined": lambda **kw: MeanDetermined(
        kw["S"],
        kw["n"],
        kw["m"],
        kw.get("E")
    ),
    "TwiceAsTheMean": lambda **kw: TwiceAsTheMean(
        kw["S"],
```

```
        kw["n"]
    ),
    "GammaDirichlet": lambda **kw: GammaDirichlet(
        kw["S"],
        kw["n"],
        kw["alpha"],
        kw.get("E")
    ),
}

# ========== saving results ==========

def saveFig(fig, path, show=True):
    # os.makedirs(os.path.dirname(path), exist_ok=True)
    # fig.savefig(path, dpi=150, bbox_inches="tight")
    if show:
        plt.show()
    plt.close(fig)

def plotBox(samples, exp_name, E=None, show=True):
    data = list(zip(*samples))
    n = len(data)

    fig, ax = plt.subplots(figsize=(10, 6))

    bp = ax.boxplot(
        data,
        patch_artist=True,
        whis=2,
        flierprops=dict(
            marker='x',
            markersize=4,
            alpha=0.5
        ),
        boxprops=dict(facecolor="#cce5ff"),
        medianprops=dict(color="black", linewidth=1)
    )
    offset = 0.01*max(map(max, data))
    fontSize = max(2,int(30/np.log2(n)))
    for i, col in enumerate(data, start=1):
        e = E[i-1]
        mean = np.mean(col)
        ax.hlines(
            mean,
            i - 0.35,
            i + 0.35,
            colors="blue",
```

```
            linewidth=2
        )
        ax.text(
            i,
            mean+offset,
            f" ={mean:.2f}",
            color="blue",
            fontsize=fontSize,
            ha="center",
            va="bottom"
        )
        ax.hlines(
            e,
            i - 0.35,
            i + 0.35,
            colors="red",
            linestyles="--",
            linewidth=2
        )
        ax.text(
            i,
            e-offset*3,
            f"E={e:.2f}",
            color="red",
            fontsize=fontSize,
            ha="center",
            va="bottom"
        )
    ax.set_xlabel("person")
    ax.set_ylabel("value")
    ax.set_title("Allocation distribution boxplot")
    path = os.path.join(BASE_DIR, exp_name, "boxplot.png")
    saveFig(fig, path, show)


def plotHist(samples, exp_name, i, show=True):
    fig, ax = plt.subplots()
    data = [s[i] for s in samples]
    ax.hist(data, bins=50, density=True)
    ax.set_title(f"Person {i+1} distribution")

    path = os.path.join(BASE_DIR, exp_name, f"hist_p{i+1}.png")
    saveFig(fig, path, show)

def plotHist_ax(ax, samples, i):
    data = [s[i] for s in samples]
    ax.hist(data, bins=50, density=True)
```

```
    ax.set_title(f"P{i+1}", fontsize=10)


# ========== experiment runner ==========

def run(exp):
    allocator = ALLOCATORS[exp["allocator"]]
    params = exp["params"]
    repeat = exp["repeat"]

    samples = []
    for _ in range(1, repeat+1):
        # if _%100==0:
        #     logger.debug(f"Running experiment {exp['name']}: {_}/{repeat}")
        samples.append(allocator(**params))

    stats = analyze(samples)
    return samples, stats

# ========== main ==========

def myStep(S):
    q = max(2,int(np.log2(S)))
    return S%q, S//q, q

def step3(S):
    q = 3
    return S%q, S//q, q

exp_DIE_dyStep_Eequal = {
    "name": "DIE-dyStep-Eequal",
    "allocator": "DIE",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 20,
        "E": None,
        "step": myStep,
    },
    "repeat": REPEAT,
    "test": False
}
exp_DIE_dyStep_Efixed = {
    "name": "DIE-dyStep-Efixed",
    "allocator": "DIE",
    "params": {
        "S": 6000,
```

```
        "n": 15,
        "m": 20,
        "E": FIXED_E,
        "step": myStep,
    },
    "repeat": REPEAT,
    "test": False
}
exp_DIE_3Step_Eequal = {
    "name": "DIE-3Step-Eequal",
    "allocator": "DIE",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 20,
        "E": None,
        "step": step3,
    },
    "repeat": REPEAT,
    "test": False
}
exp_DIE_3Step_Efixed = {
    "name": "DIE-3Step-Efixed",
    "allocator": "DIE",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 20,
        "E": FIXED_E,
        "step": step3,
    },
    "repeat": REPEAT,
    "test": False
}
exp_UniformRandom = {
    "name": "UniformRandom",
    "allocator": "UniformRandom",
    "params": {
        "S": 6000,
        "n": 15,
    },
    "repeat": REPEAT,
    "test": False
}
exp_MeanDetermined_1_Eequal = {
    "name": "MeanDetermined-1-Eequal",
    "allocator": "MeanDetermined",
```

```
    "params": {
        "S": 6000,
        "n": 15,
        "m": 1,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
exp_MeanDetermined_1_Efixed = {
    "name": "MeanDetermined-1-Efixed",
    "allocator": "MeanDetermined",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 1,
        "E": FIXED_E,
    },
    "repeat": REPEAT,
    "test": False
}
exp_MeanDetermined_10_Eequal = {
    "name": "MeanDetermined-10-Eequal",
    "allocator": "MeanDetermined",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 10,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
exp_MeanDetermined_10_Efixed = {
    "name": "MeanDetermined-10-Efixed",
    "allocator": "MeanDetermined",
    "params": {
        "S": 6000,
        "n": 15,
        "m": 10,
        "E": FIXED_E,
    },
    "repeat": REPEAT,
    "test": False
}
exp_TwiceAsTheMean = {
    "name": "TwiceAsTheMean",
```

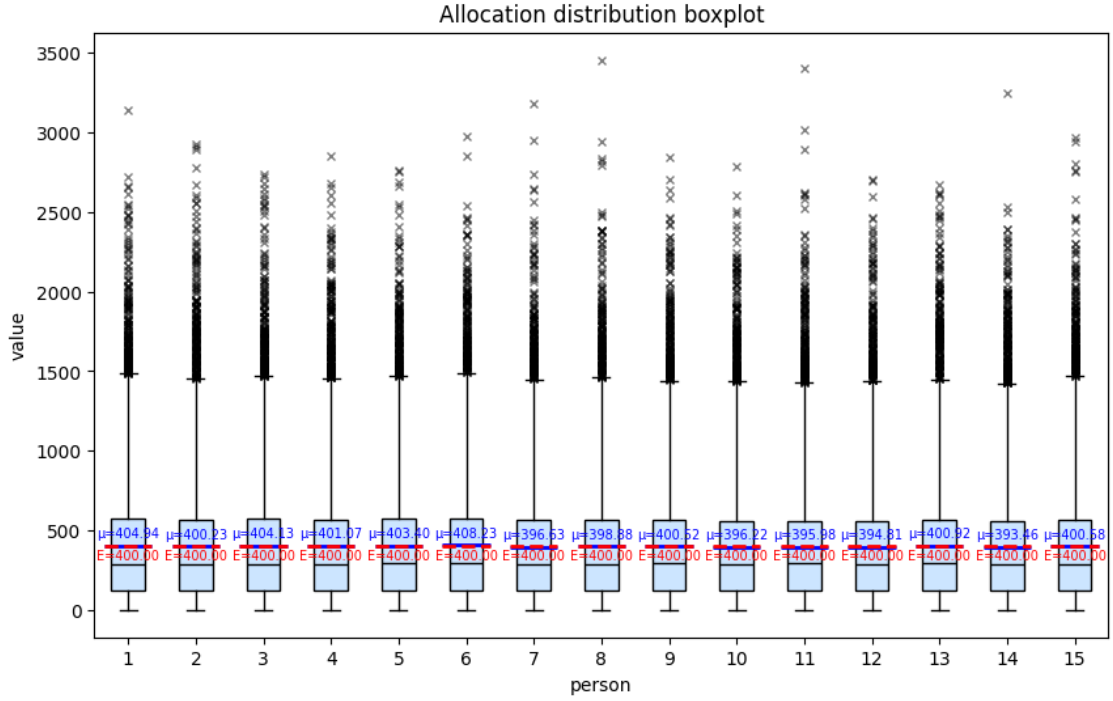
```
    "allocator": "TwiceAsTheMean",
    "params": {
        "S": 6000,
        "n": 15,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_10_Eequal = {
    "name": "GammaDirichlet-10-Eequal",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 10.0,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_10_Efixed = {
    "name": "GammaDirichlet-10-Efixed",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 10.0,
        "E": FIXED_E,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_100_Eequal = {
    "name": "GammaDirichlet-100-Eequal",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 100.0,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_100_Efixed = {
    "name": "GammaDirichlet-100-Efixed",
    "allocator": "GammaDirichlet",
```

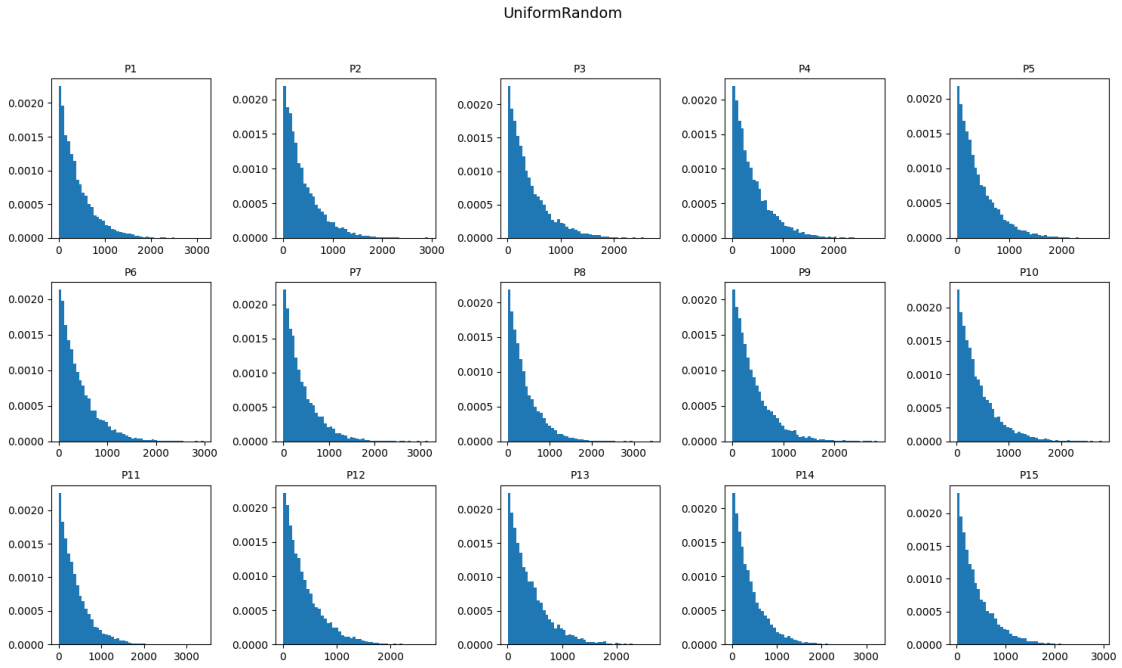
```
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 100.0,
        "E": FIXED_E,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_1000_Eequal = {
    "name": "GammaDirichlet-1000-Eequal",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 1000.0,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_1000_Efixed = {
    "name": "GammaDirichlet-1000-Efixed",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 1000.0,
        "E": FIXED_E,
    },
    "repeat": REPEAT,
    "test": False
}
exp_GammaDirichlet_fitted_Eequal = {
    "name": "GammaDirichlet-fitted-Eequal",
    "allocator": "GammaDirichlet",
    "params": {
        "S": 6000,
        "n": 15,
        "alpha": 33.99,
        "E": None,
    },
    "repeat": REPEAT,
    "test": False
}
```

```
if __name__ == "__main__":

    for experiment in [exp_UniformRandom, exp_MeanDetermined_1_Eequal,␣
 ↪exp_MeanDetermined_1_Efixed, exp_MeanDetermined_10_Eequal,␣
 ↪exp_MeanDetermined_10_Efixed, exp_DIE_3Step_Eequal, exp_DIE_3Step_Efixed,␣
 ↪exp_DIE_dyStep_Eequal, exp_DIE_dyStep_Efixed, exp_GammaDirichlet_10_Eequal,␣
 ↪exp_GammaDirichlet_10_Efixed, exp_GammaDirichlet_100_Eequal,␣
 ↪exp_GammaDirichlet_100_Efixed, exp_GammaDirichlet_1000_Eequal,␣
 ↪exp_GammaDirichlet_1000_Efixed, exp_TwiceAsTheMean]:
        try:
            samples, stats = run(experiment)
        except ValueError as e:
            logger.error(f"Error in experiment {experiment['name']}: {e}")
            exit(1)

        # logger.success(f"Successfully ran experiment {experiment['name']}")

        print("-" * 80)
        print(experiment["name"] + ":")

        print("Mean:", stats["mean"], "|",stats["global"]["mean"])
        print("Var :", stats["var"], "|",stats["global"]["var"])
        print("Std :", stats["std"], "|",stats["global"]["std"])

        E = experiment["params"]["E"] if "E" in experiment["params"] and␣
 ↪experiment["params"]["E"] is not None else [experiment["params"]["S"]/
 ↪experiment["params"]["n"]]*experiment["params"]["n"]
        plotBox(samples, experiment["name"], E, True)
        # for i in range(experiment["params"]["n"]):
        #     plotHist(samples, experiment["name"], i, True)
        n = experiment["params"]["n"]
        fig, axes = plt.subplots(3, 5, figsize=(15, 9))
        axes = axes.flatten()

        for i in range(n):
            plotHist_ax(axes[i], samples, i)

        fig.suptitle(experiment["name"], fontsize=14)
        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.show()
        plt.close(fig)


        if experiment["test"]:
            S = experiment["params"]["S"]
            expr = json2dict(f"{BASE_DIR}/{experiment['name']}/stats.
 ↪json")['freq']
```

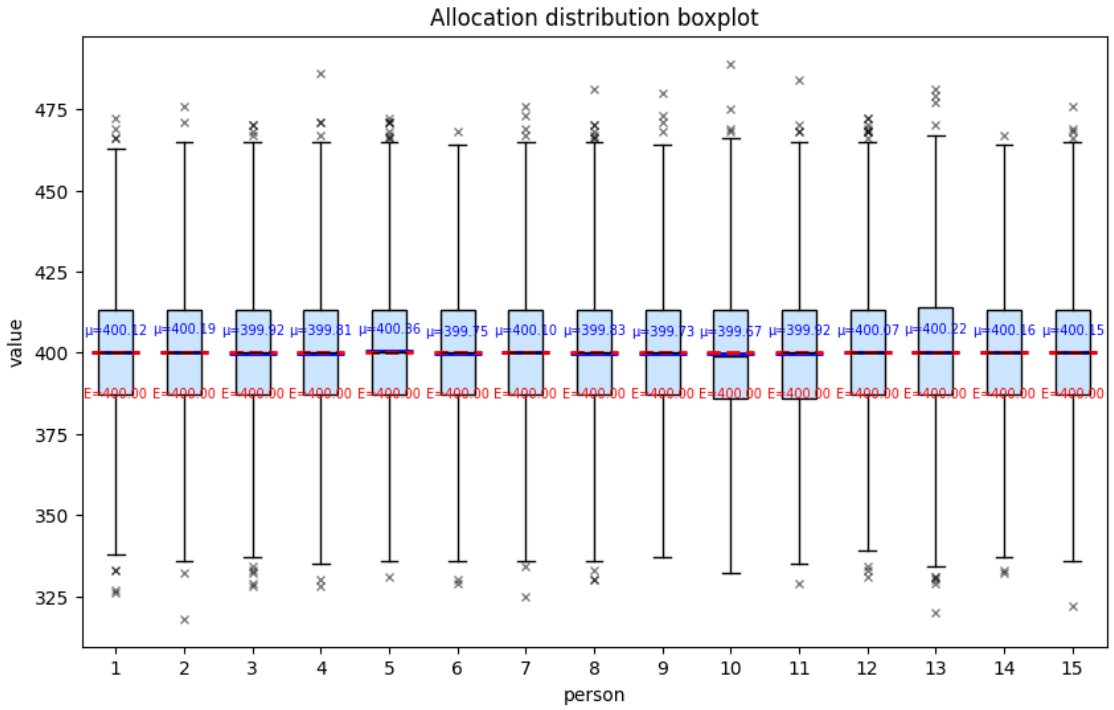
```
            real = json2dict(DATA_PATH)['freq']
            KSstat, KSpvalue = ksTest(expr, real, S)
            CHIstat, CHIpvalue = chiSquareTest(expr, real, S, EXP)
            print(f"KS test:")
            print(f"  statistic={KSstat}, pvalue={KSpvalue}")
            print(f"Chi-square test:")
            print(f"  statistic={CHIstat}, pvalue={CHIpvalue}")
```

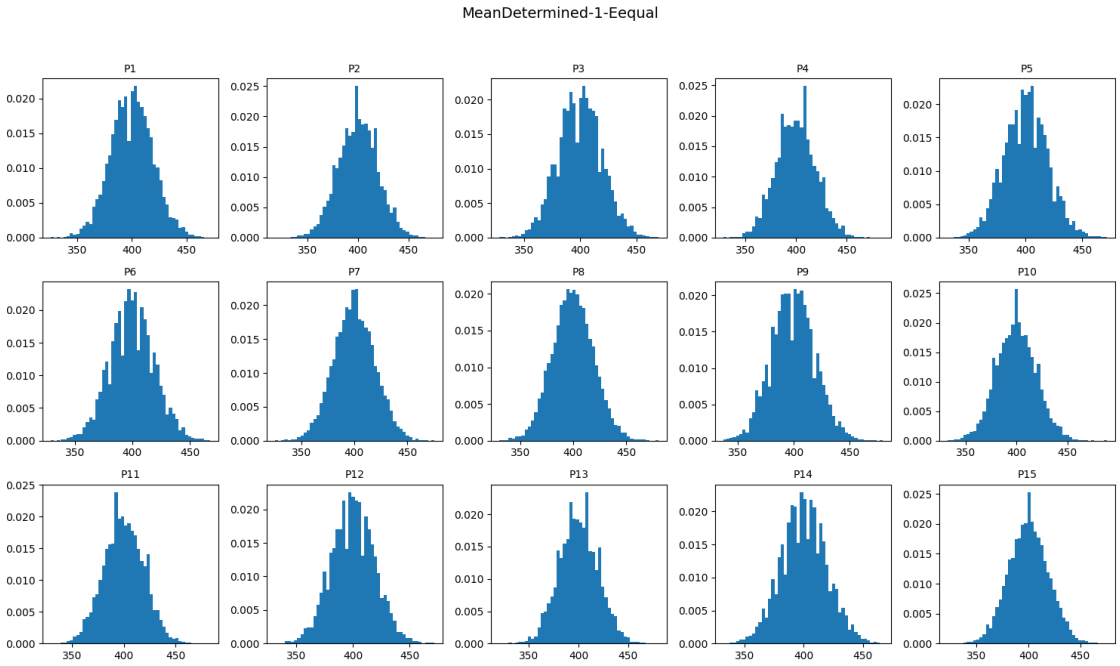
```
UniformRandom:
Mean: [404.9402 400.226  404.1278 401.0721 403.4008 408.2259 396.6253 398.884
 400.5198 396.2234 395.983  394.8145 400.9178 393.4623 400.5771] | 400.0
Var : [147270.57602396 143657.510924   146833.79986716 141349.50930159
 138858.61955936 144107.81846919 137745.53789991 138935.681344
 137147.90940796 137606.55369244 137254.378511   134919.51568975
 139990.21364316 136333.45937871 143757.62085559] | 140400.31265333333
Std : [383.75848658 379.0217816  383.18898714 375.96477136 372.63738347
 379.61535594 371.14085992 372.74076963 370.33486118 370.9535735
 370.47858037 367.31391981 374.15266088 369.23361085 379.15382216] |
374.70029710868033
```





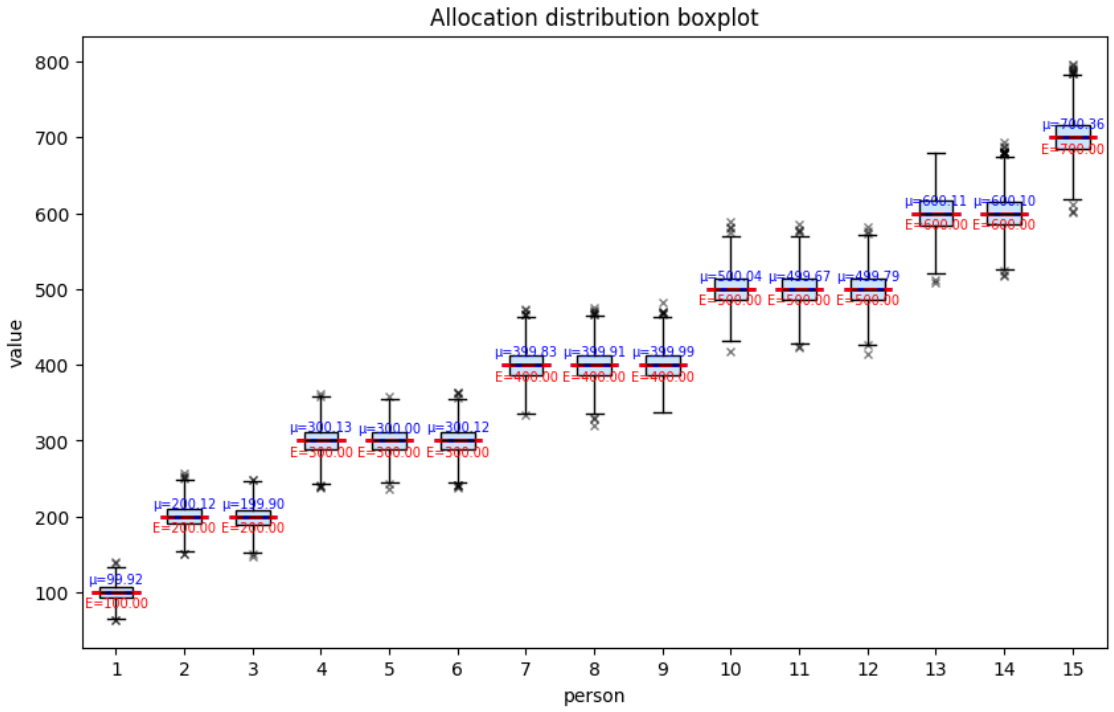
```
MeanDetermined-1-Eequal:
Mean: [400.1229 400.1901 399.9168 399.8134 400.359  399.7488 400.1021 399.8308
 399.7277 399.6726 399.9168 400.0702 400.2193 400.1634 400.1461] | 400.0
Var : [370.90219559 377.40056199 376.25187776 374.88378044 375.785119
 371.06389856 372.58767559 373.69217136 378.65015271 376.29940924
 376.07727776 368.40547196 386.08820751 373.01570044 368.02595479] | 374.64952
Std : [19.25882124 19.4268001  19.39721314 19.36191572 19.38517782 19.26301894
 19.30253029 19.33111925 19.45893504 19.39843832 19.39271198 19.19389153
 19.6491274  19.31361438 19.18400257] | 19.355865260948683
```

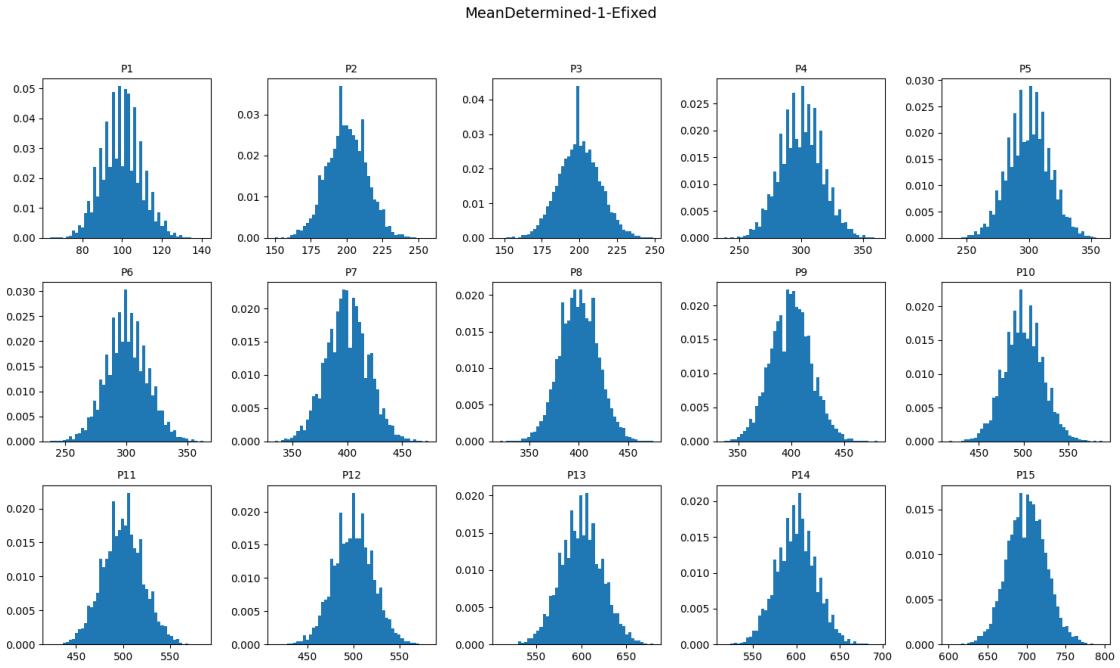




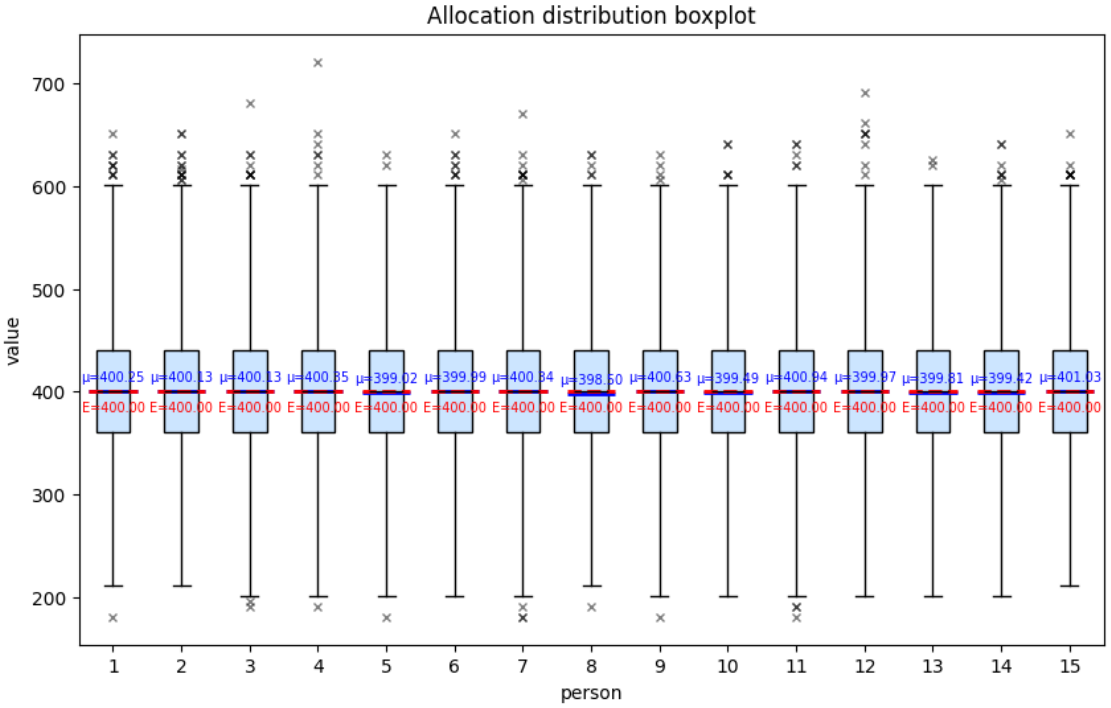
```
MeanDetermined-1-Efixed:
Mean: [ 99.9227 200.1172 199.9036 300.1335 300.0014 300.1203 399.831  399.9056
```

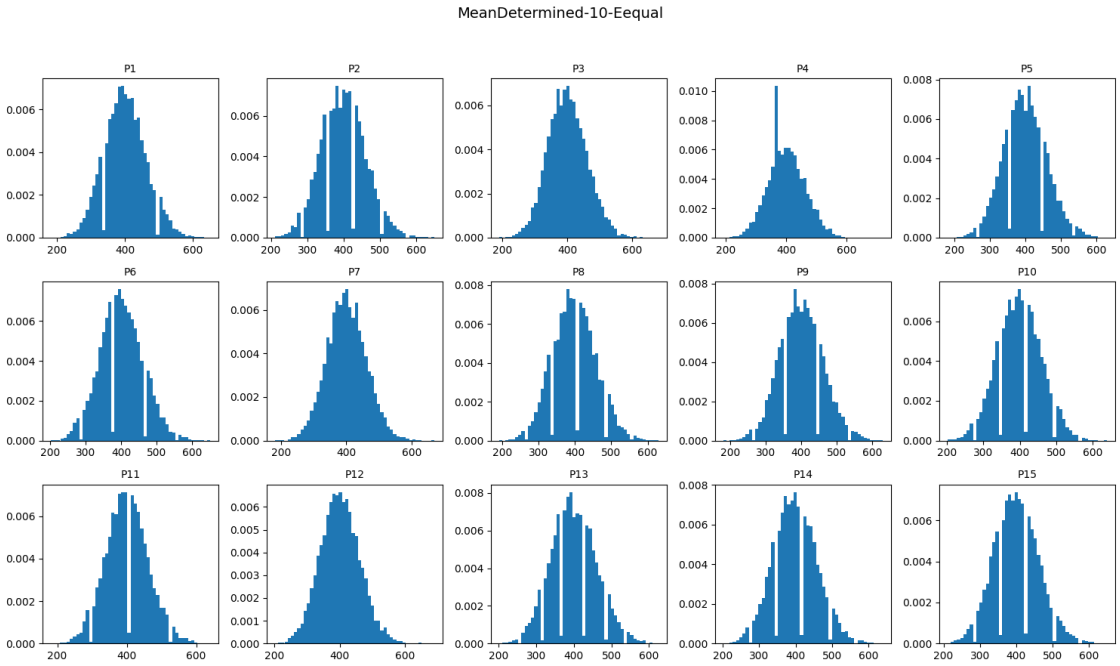
```
 399.9883 500.041  499.6676 499.7942 600.1073 600.1043 700.362 ] | 400.0
Var : [ 97.44172471 191.50146416 190.15150704 283.71567775 278.51819804
 284.21082791 366.629239   373.18228864 365.07856311 453.276519
 457.68131024 461.23144636 544.31038671 527.15042151 616.262156  ] |
27045.344573333332
Std : [ 9.8712575  13.83840541 13.78954339 16.84386172 16.68886449 16.85855355
 19.14756483 19.31792661 19.10702915 21.29029166 21.39348757 21.47629964
 23.33046049 22.95975656 24.82462801] | 164.45468851125324
```





```
MeanDetermined-10-Eequal:
Mean: [400.247  400.1265 400.1265 400.3535 399.0225 399.9885 400.3375 398.505
 400.6345 399.4855 400.942  399.967  399.8085 399.4225 401.033 ] | 400.0
Var : [3752.182991   3753.93949775 3649.09449775 3700.09453775 3642.24199375
 3701.91436775 3698.48859375 3592.649975   3687.62890975 3684.42378975
 3755.196636   3758.242911   3728.34282775 3674.24399375 3778.728911  ] |
3704.262
Std : [61.25506502 61.26940099 60.40773541 60.82840239 60.35099    60.84335927
 60.81520035 59.9387185  60.72585042 60.69945461 61.27965924 61.30450971
 61.06015745 60.61554251 61.47136659] | 60.862648644304
```

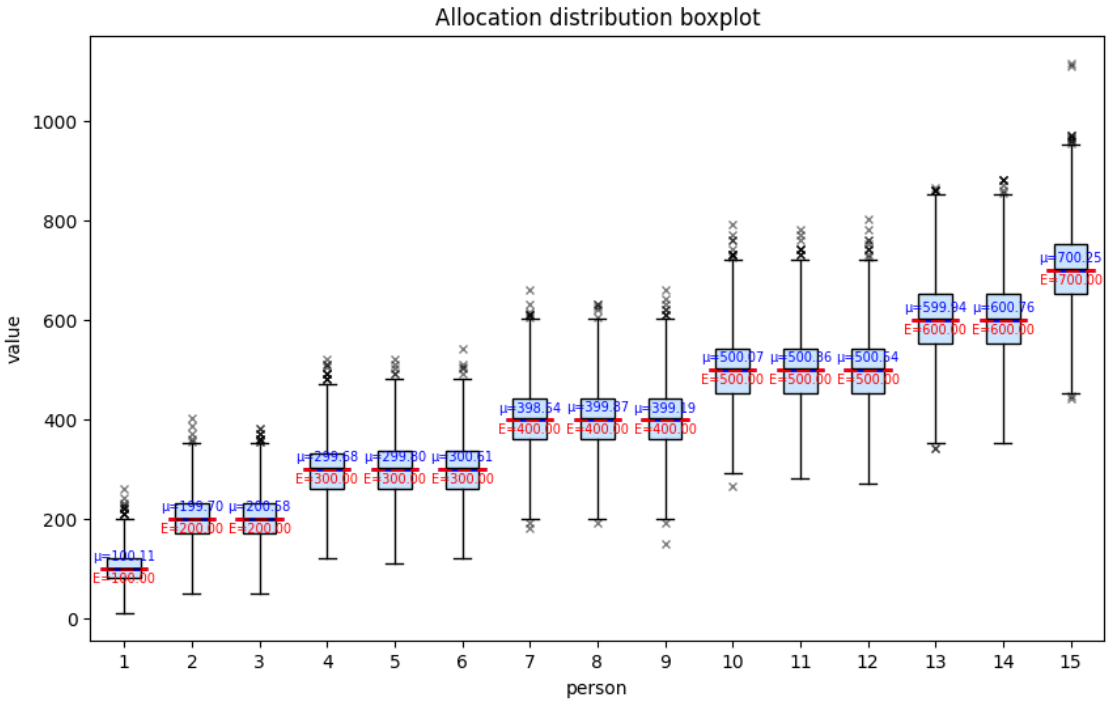


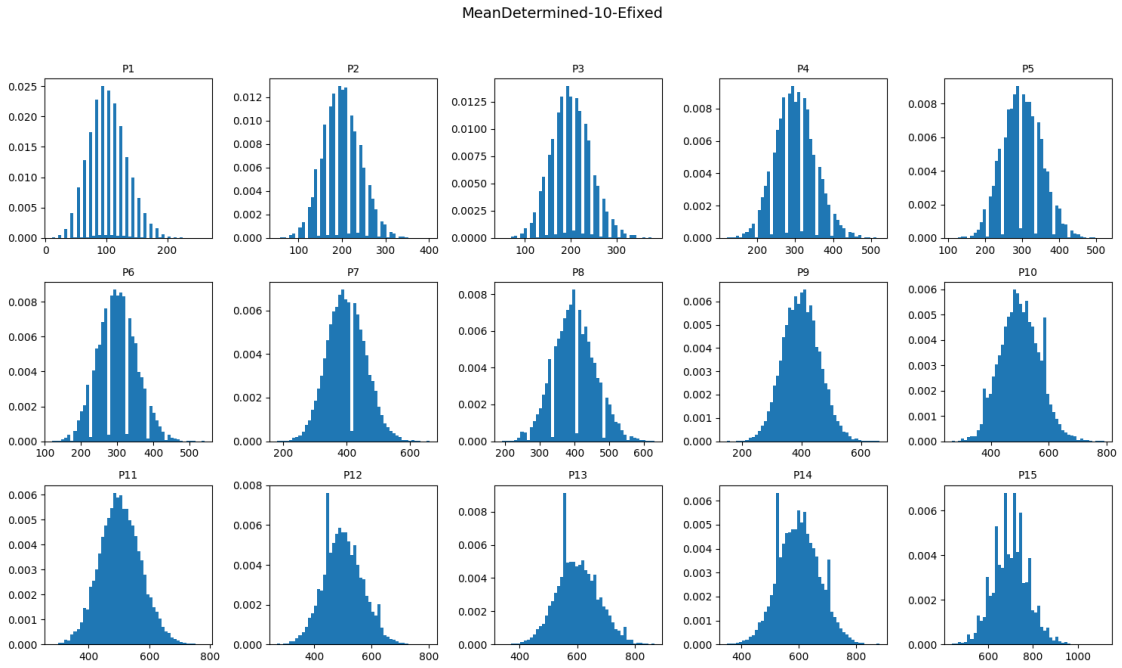


MeanDetermined-10-Efixed:

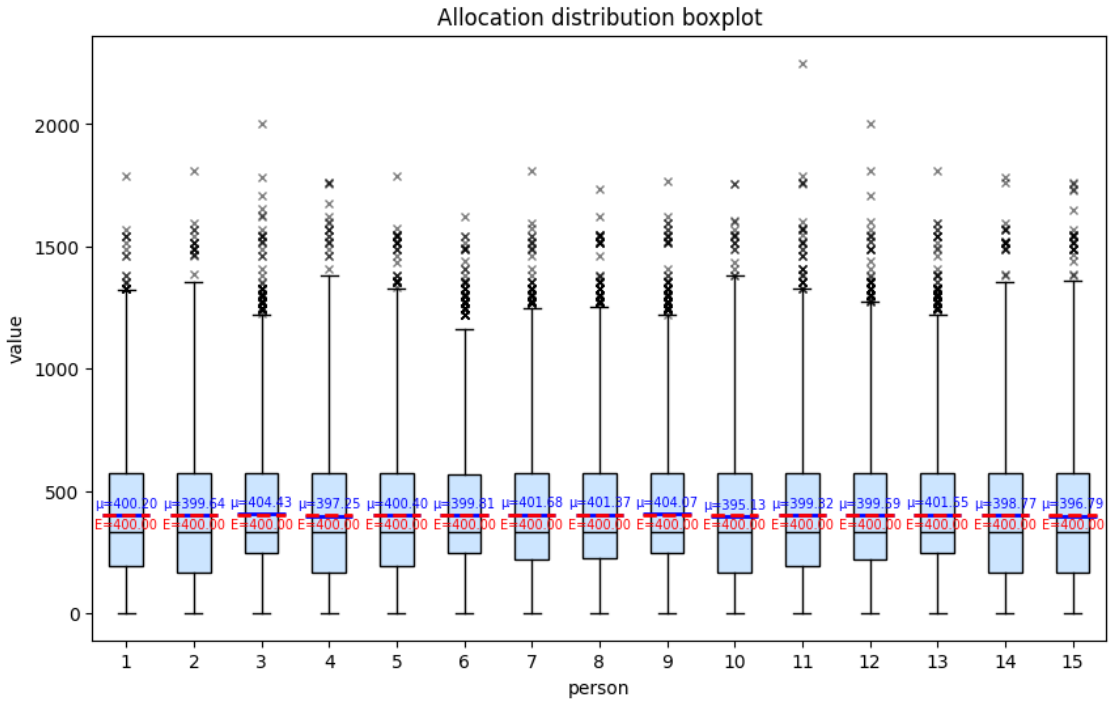
Mean: [100.113 199.6985 200.578 299.6795 299.8015 300.6145 398.5435 399.874

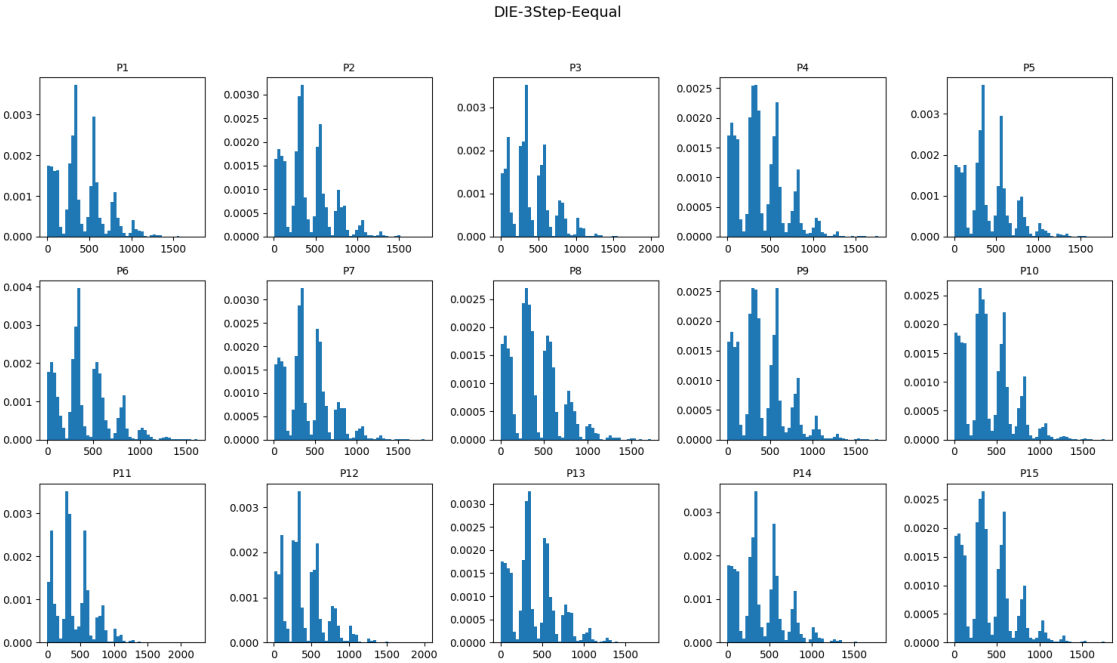
```
 399.1855 500.0735 500.3565 500.536  599.9425 600.756  700.2475] | 400.0
Var : [ 999.263231   1901.03359775 1924.476916   2886.39877975 2808.25109775
 2938.48888975 3682.17810775 3791.577124   3723.45008975 4623.52409775
 4640.11340775 4666.494704   5422.97419375 5336.290464   6156.86124375] |
30395.319333333333
Std : [31.61112511 43.600844   43.86886044 53.72521549 52.99293441 54.20783052
 60.68095342 61.57578358 61.0200794  67.99650063 68.1183779  68.31174646
 73.64084596 73.04991762 78.4656692 ] | 174.3425344926858
```





```
DIE-3Step-Eequal:
Mean: [400.2014 399.6374 404.4308 397.2472 400.4039 399.813  401.6773 401.3717
 404.0704 395.1281 399.316  399.5945 401.5513 398.7681 396.7889] | 400.0
Var : [74646.44863804 77195.31512124 76687.54741136 75927.30209216
 76761.20336479 74206.265631   74892.83996471 75459.21973911
 75863.77864384 75324.21229039 76427.210544   75263.79866975
 76318.90876831 76287.63892239 78048.01833679] | 75959.81028
Std : [273.21502272 277.84044904 276.92516572 275.54909198 277.05812272
 272.40827012 273.66556226 274.69841597 275.43380084 274.45256838
 276.45471699 274.34248426 276.25877139 276.20217038 279.37075426] |
275.6080736843535
```

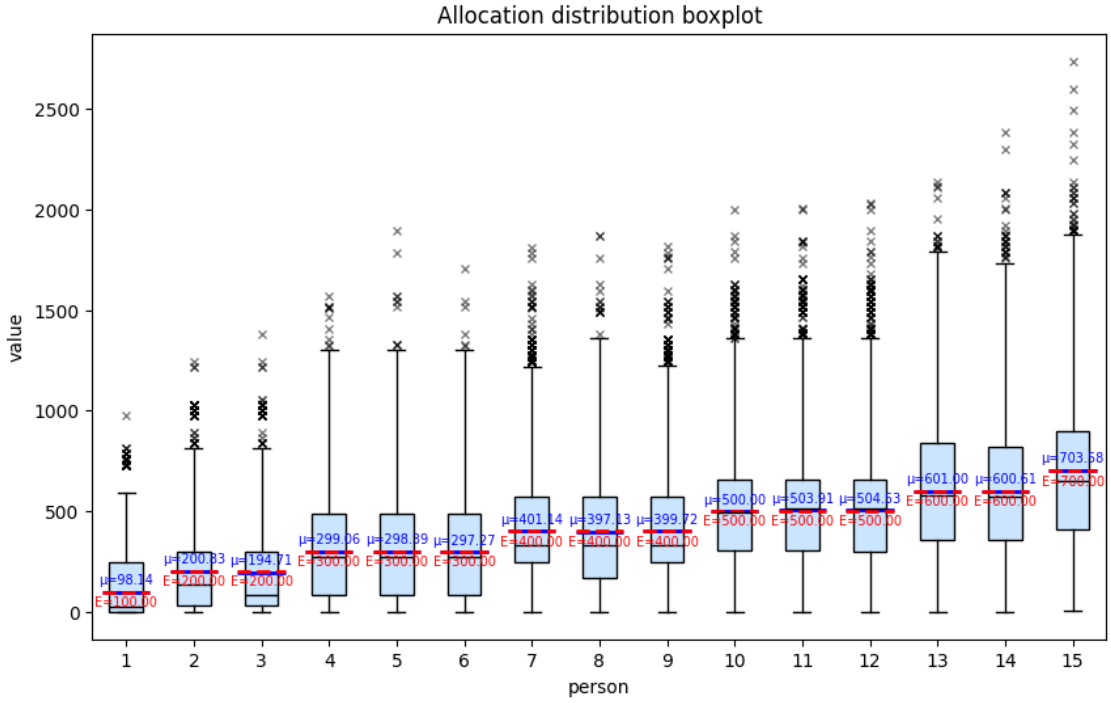


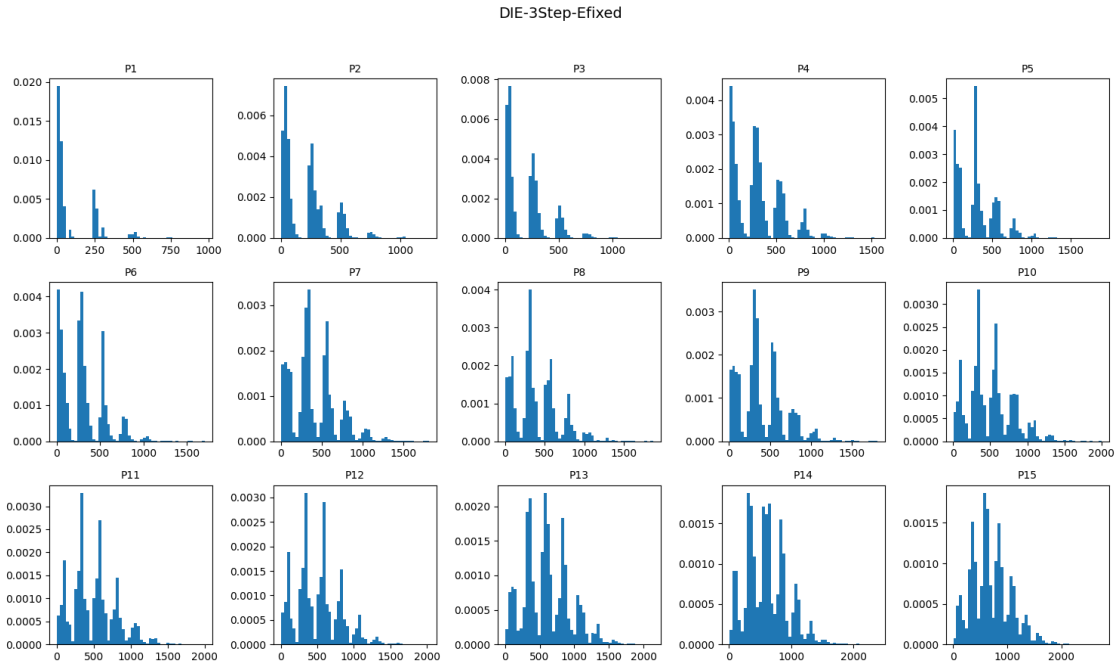


DIE-3Step-Efixed:

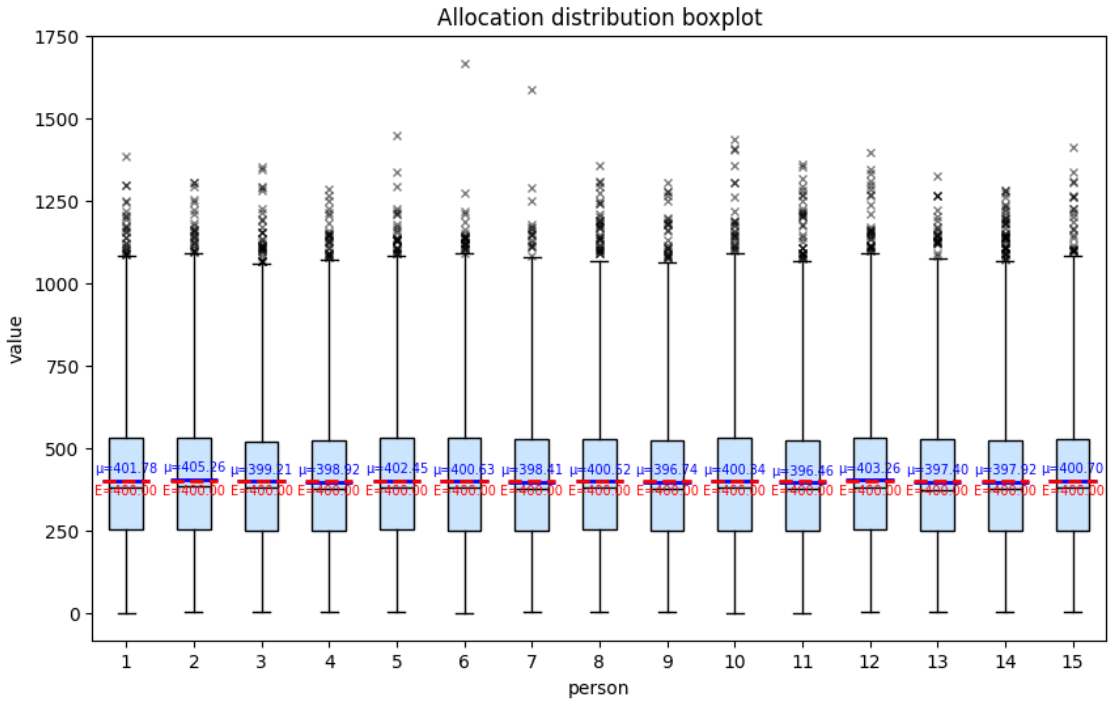
Mean: [98.14 200.8316 194.706 299.0572 298.3853 297.2677 401.1443 397.1272

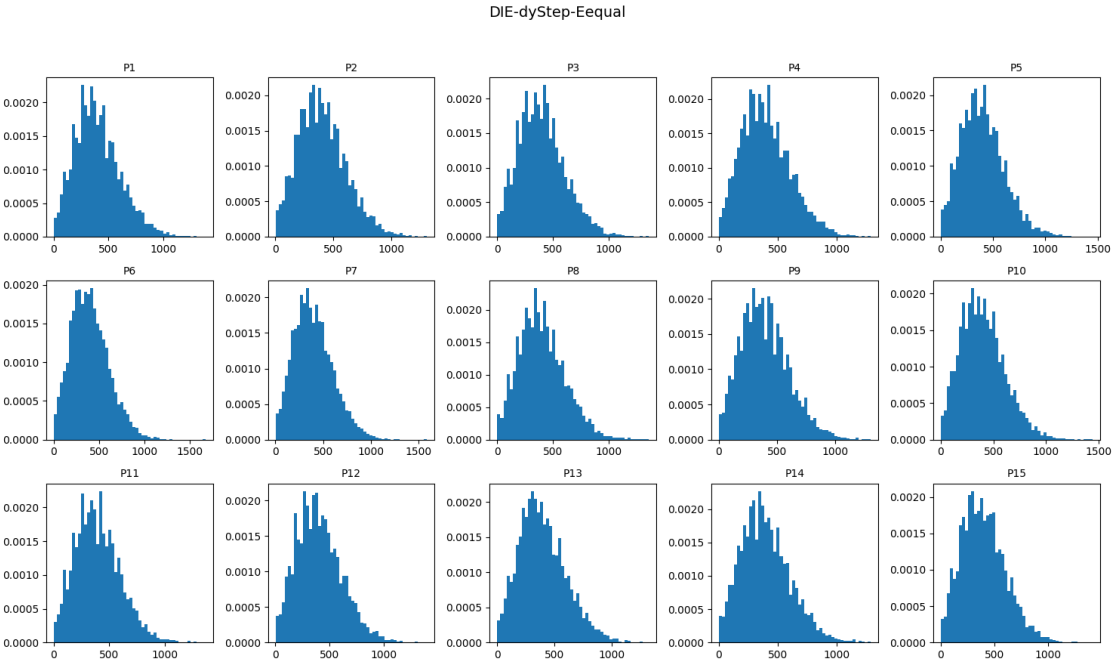
```
 399.7198 499.996  503.9134 504.5283 600.995  600.6099 703.5783] | 400.0
Var : [ 19636.4532       38482.15884144  38343.351764    57929.12352816
  57099.81764391  56864.34923671  75657.57127751  75324.20362016
  75200.35968796  90887.094784    94230.99870044  94542.22119911
 109953.598775   108429.28972199 129244.49206911] | 102024.21670666667
Std : [140.13012952 196.16869995 195.81458517 240.68469733 238.95568134
 238.46246924 275.05921413 274.45255258 274.22683984 301.47486592
 306.97068052 307.4771881  331.59251918 329.28603026 359.50589991] |
319.41229892830785
```





```
DIE-dyStep-Eequal:
Mean: [401.7818 405.258  399.2053 398.9188 402.4497 400.6317 398.4145 400.5233
 396.7396 400.3399 396.457  403.2585 397.4043 397.9193 400.6983] | 400.0
Var : [42386.20458876 41770.401636   41097.06915191 41355.78940656
 42354.85486991 41904.93925511 41225.78168975 41407.04085711
 41438.70479184 42843.15956799 40574.134951   42999.02107775
 42010.36844151 42030.92918751 41335.40287711] | 41788.0722
Std : [205.87910187 204.37808502 202.72412079 203.36122887 205.80295156
 204.70695947 203.04133    203.48720072 203.56498911 206.9858922
 201.43022353 207.36205313 204.96431017 205.01446092 203.31109876] |
204.42131053292854
```

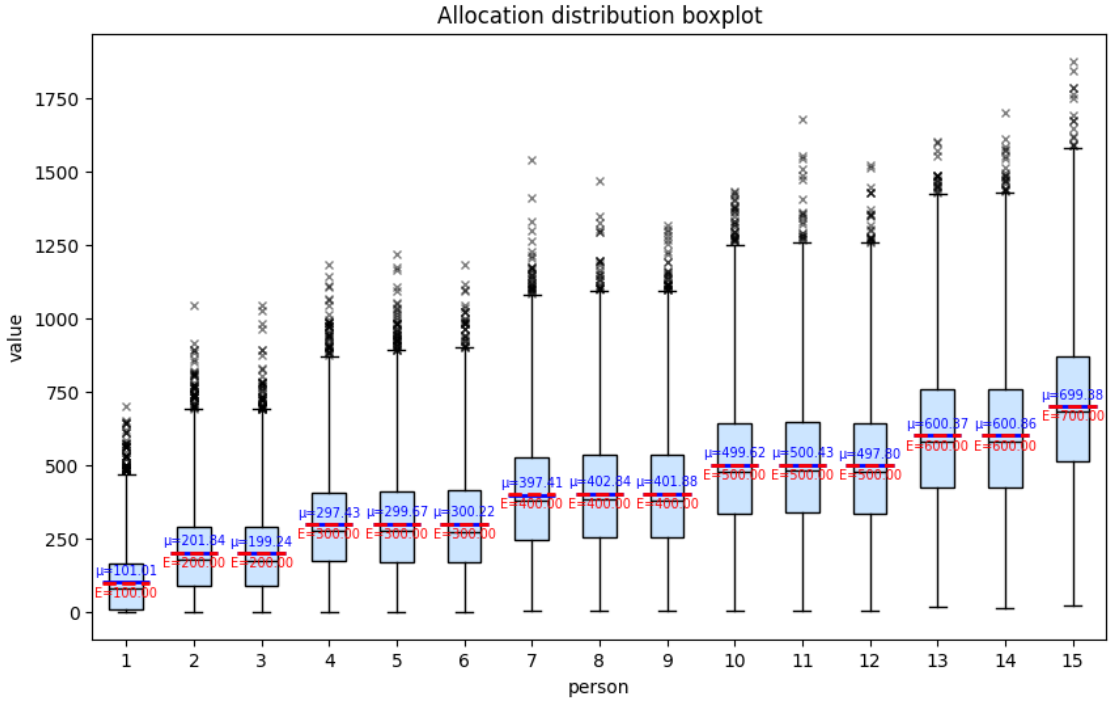


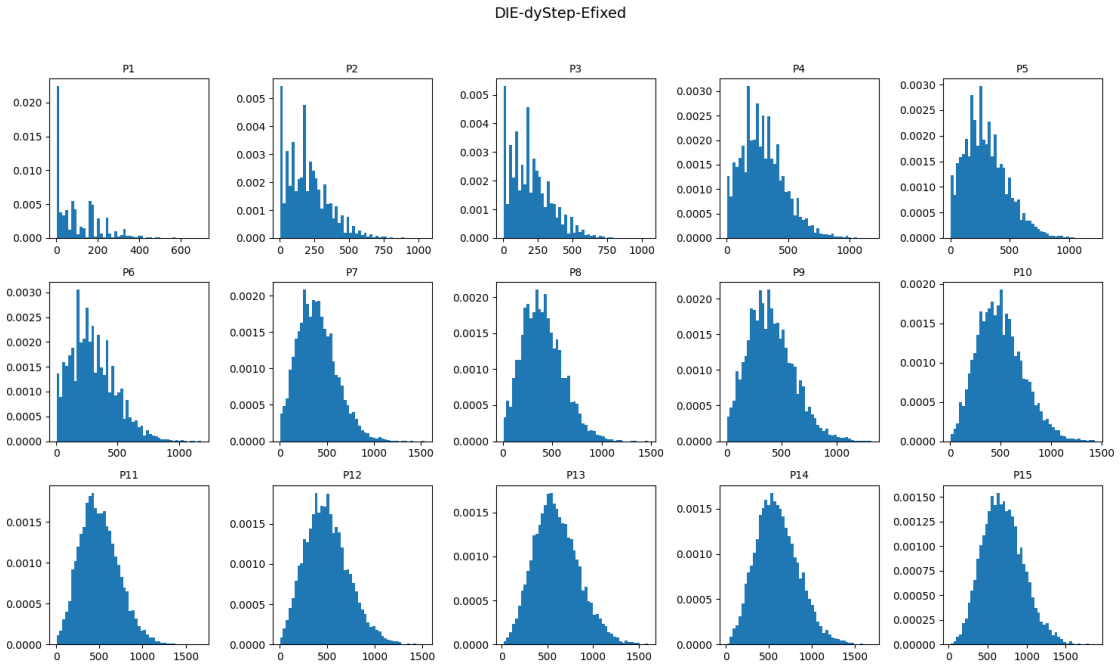


DIE-dyStep-Efixed:

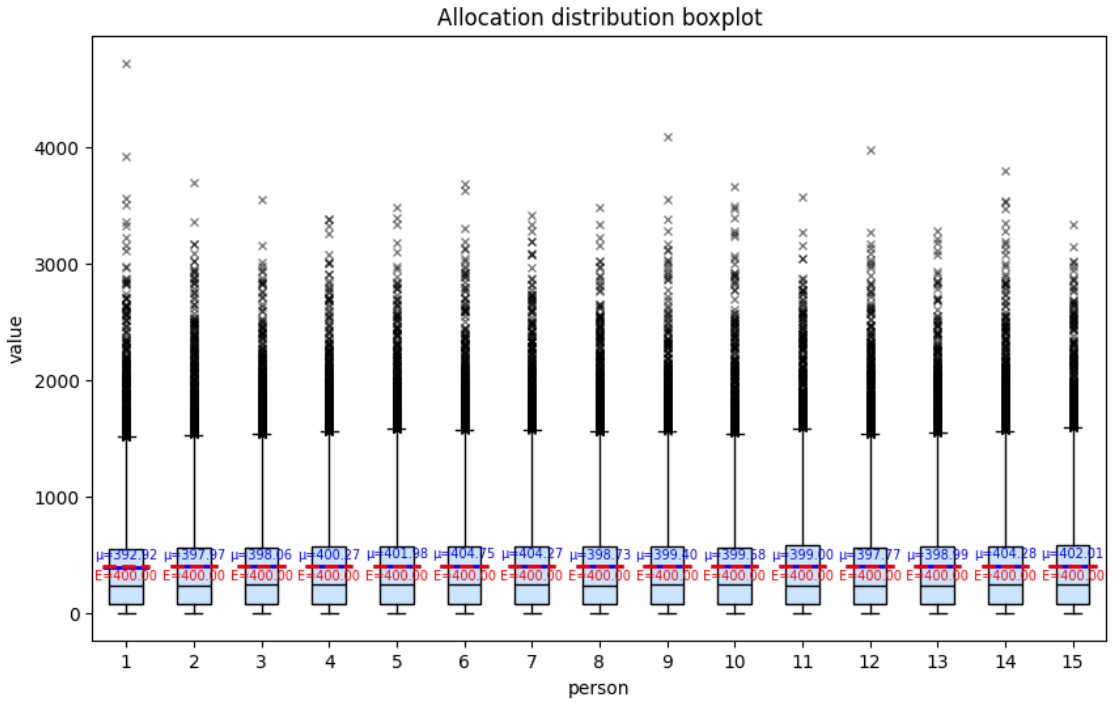
Mean: [101.0096 201.8354 199.2427 297.4269 299.6686 300.2207 397.4127 402.8448

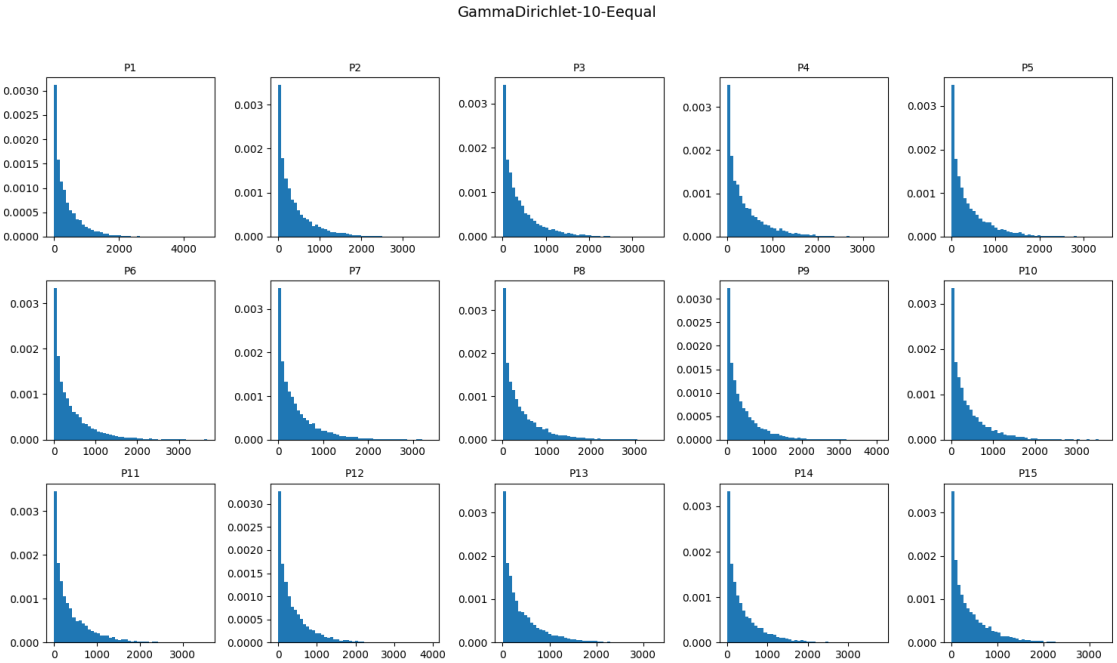
```
 401.8766 499.6187 500.4311 497.7991 600.3717 600.8649 699.3765] | 400.0
Var : [11126.77530784 21896.71330684 21575.49279671 30886.84145639
 31914.18957404 32900.62879151 41952.70257871 41695.08151296
 43140.08277244 51967.26511031 50461.61045279 52046.21973919
 60418.11633911 61525.38104799 69242.24754775] | 68131.72853333333
Std : [105.48353098 147.97538075 146.88598571 175.74652616 178.64542976
 181.38530478 204.82358892 204.19373524 207.70190845 227.96329773
 224.63661868 228.13640599 245.80096895 248.04310321 263.13921705] |
261.02055193668815
```





```
GammaDirichlet-10-Eequal:
Mean: [392.915  397.9747 398.0586 400.2715 401.9818 404.7542 404.2654 398.734
 399.4038 399.5809 398.9953 397.7747 398.9924 404.2834 402.0143] | 400.0
Var : [202550.153375   206674.73065991 197644.46436604 201591.60138775
 203425.59166876 208574.54398236 207527.14516284 202887.991044
 198661.99554556 200600.83865519 202549.89467791 200304.76513991
 198346.72634224 211187.03408444 198397.72789551] | 202737.30565333334
Std : [450.05572252 454.61492569 444.57222626 448.98953372 451.02726267
 456.69962118 455.55147367 450.43089486 445.71515068 447.88484977
 450.05543512 447.55420358 445.36134357 459.55090478 445.41859851] |
450.26359574512946
```

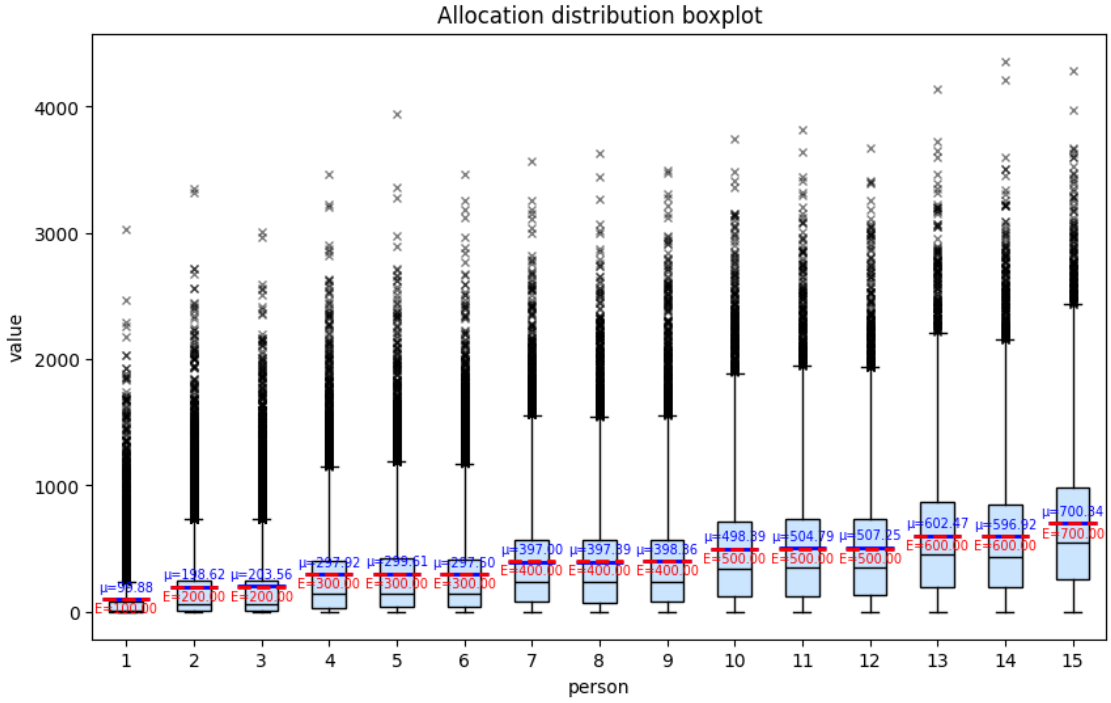


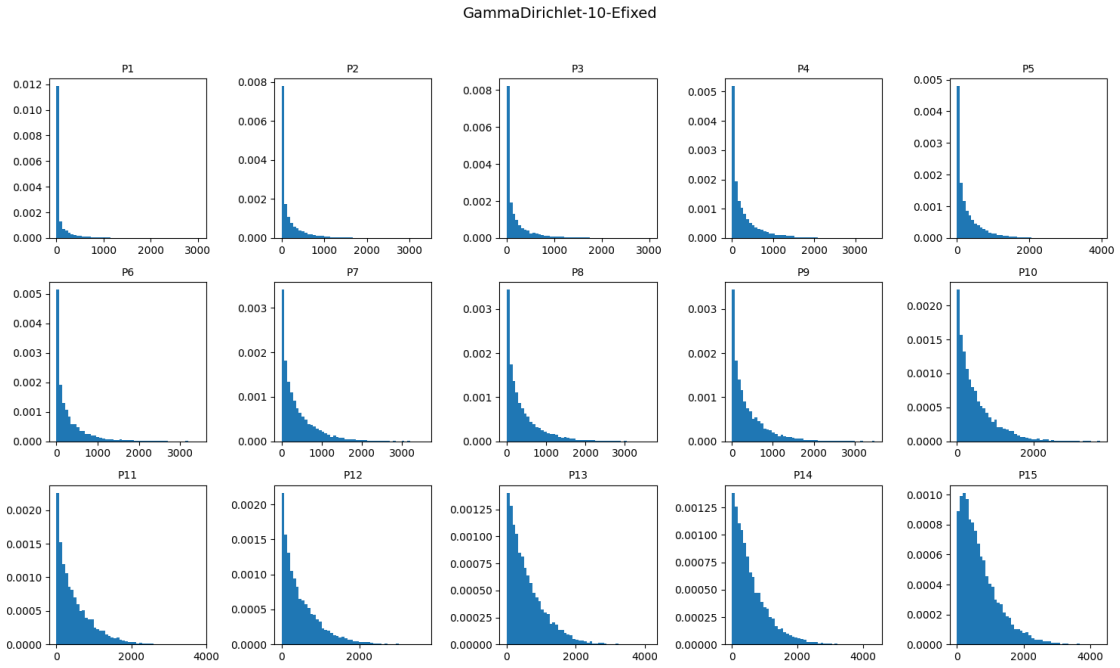


GammaDirichlet-10-Efixed:

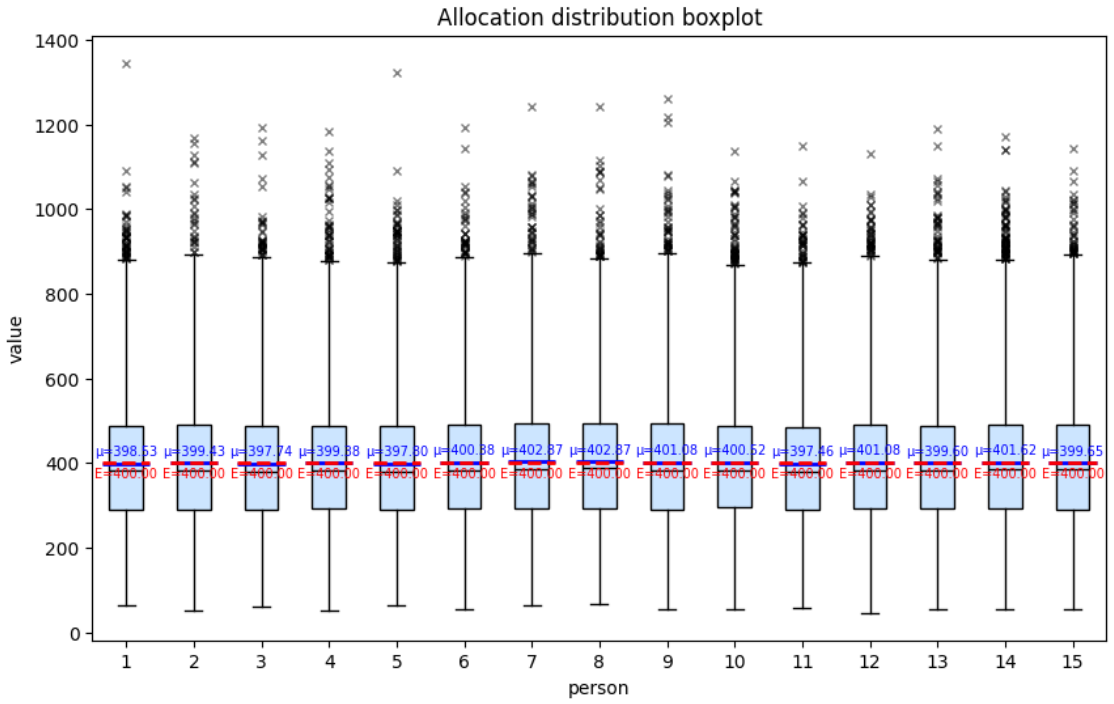
Mean: [99.8782 198.6231 203.562 297.9223 299.6086 297.5041 397.0012 397.3873

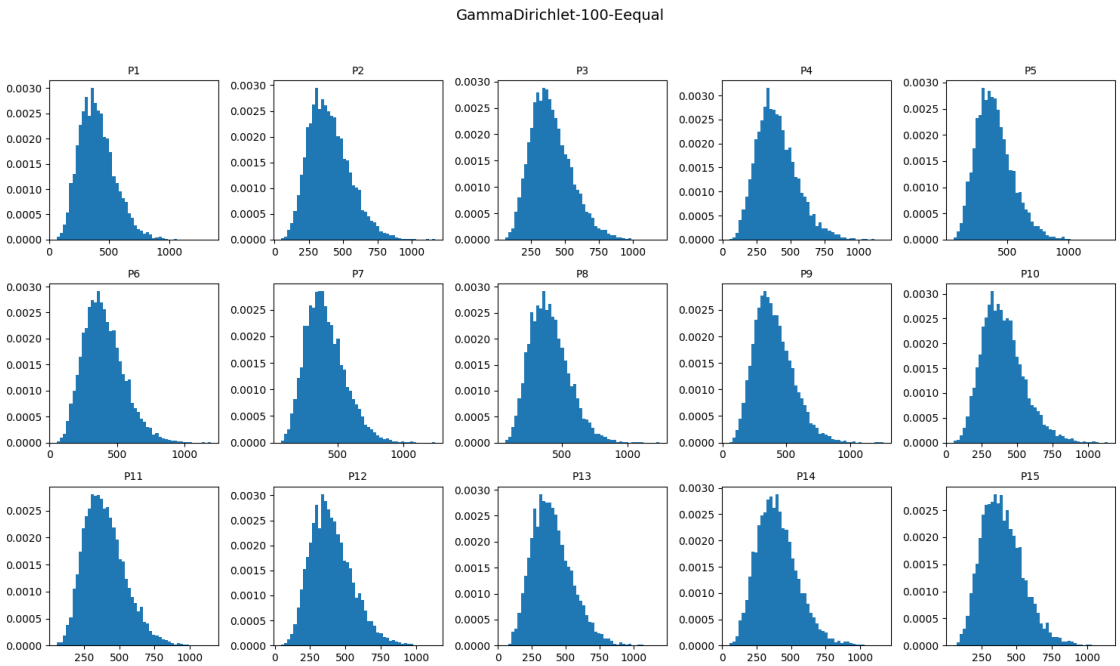
```
 398.3626 498.3886 504.7879 507.2527 602.467  596.9172 700.3372] | 400.0
Var : [ 50896.81596476 104274.45824639 106540.645956   150022.60226271
 147786.51460604 151281.06538319 193781.01179856 198690.78429871
 206037.96572124 249724.47359004 251111.04831359 251284.46104271
 290183.647111   293820.77754416 337145.88029616] | 225663.57706666668
Std : [225.60322685 322.915559   326.40564633 387.32751292 384.43011667
 388.94866677 440.20564717 445.74744452 453.91405103 499.72439763
 501.10981662 501.28281543 538.68696579 542.05237528 580.64264423] |
475.0406057029932
```





```
GammaDirichlet-100-Eequal:
Mean: [398.5262 399.4309 397.7356 399.3829 397.8019 400.3805 402.8705 402.8656
 401.0805 400.5152 397.4609 401.0803 399.6003 401.6158 399.6529] | 400.0
Var : [21622.39351356 22283.77902519 22247.24589264 22115.03028759
 22077.15885639 22086.42411975 22545.76572975 21868.32553664
 22723.41181975 21654.29036896 21371.63327119 21946.38685191
 21868.85493991 22381.55879036 21785.35542159] | 22041.26564
Std : [147.04554911 149.27752351 149.15510683 148.71123121 148.58384453
 148.61501983 150.15247494 147.87942905 150.74286656 147.15396824
 146.19040075 148.14312961 147.88121902 149.60467503 147.59862947] |
148.4630110162124
```

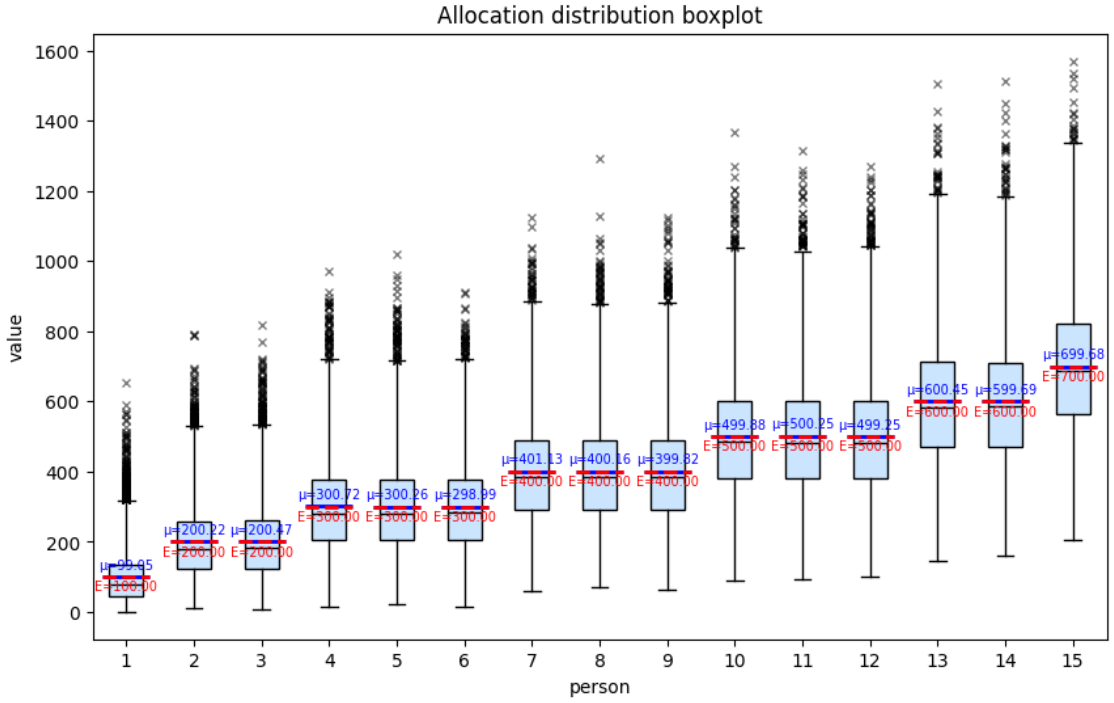


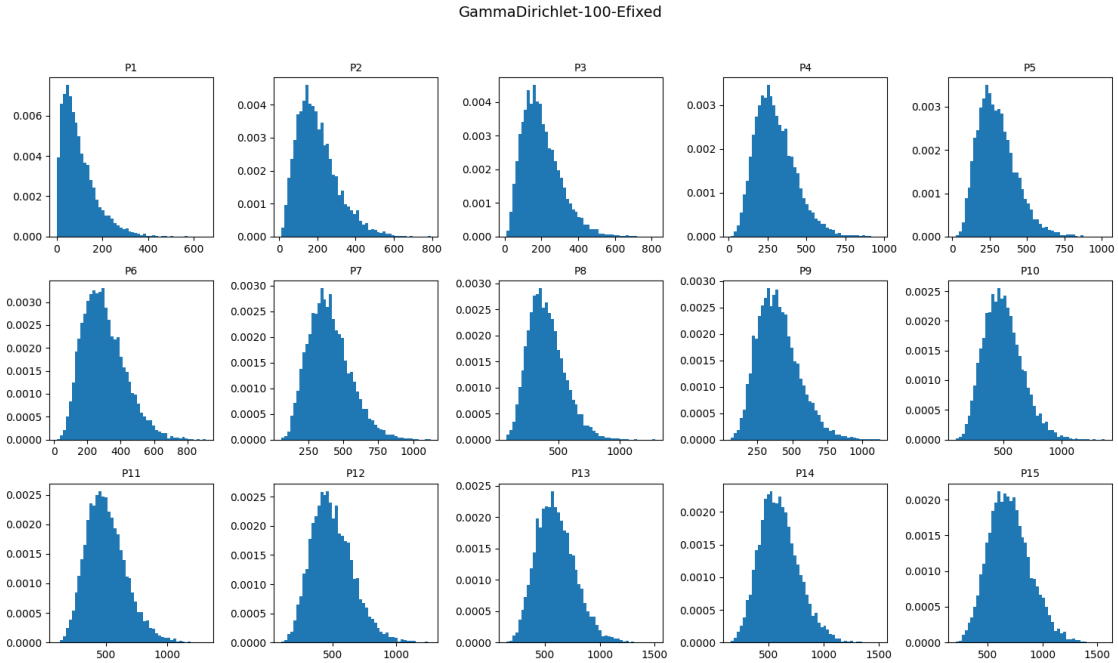


GammaDirichlet-100-Efixed:

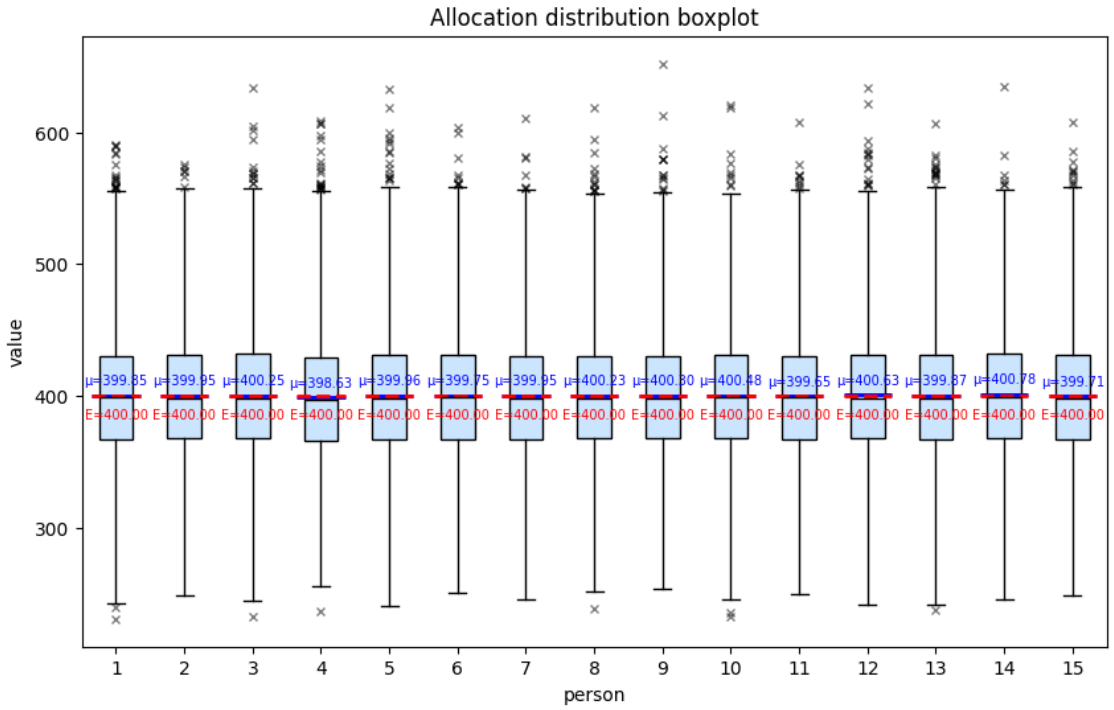
Mean: [99.053 200.2195 200.4718 300.7241 300.2579 298.9893 401.1272 400.1551

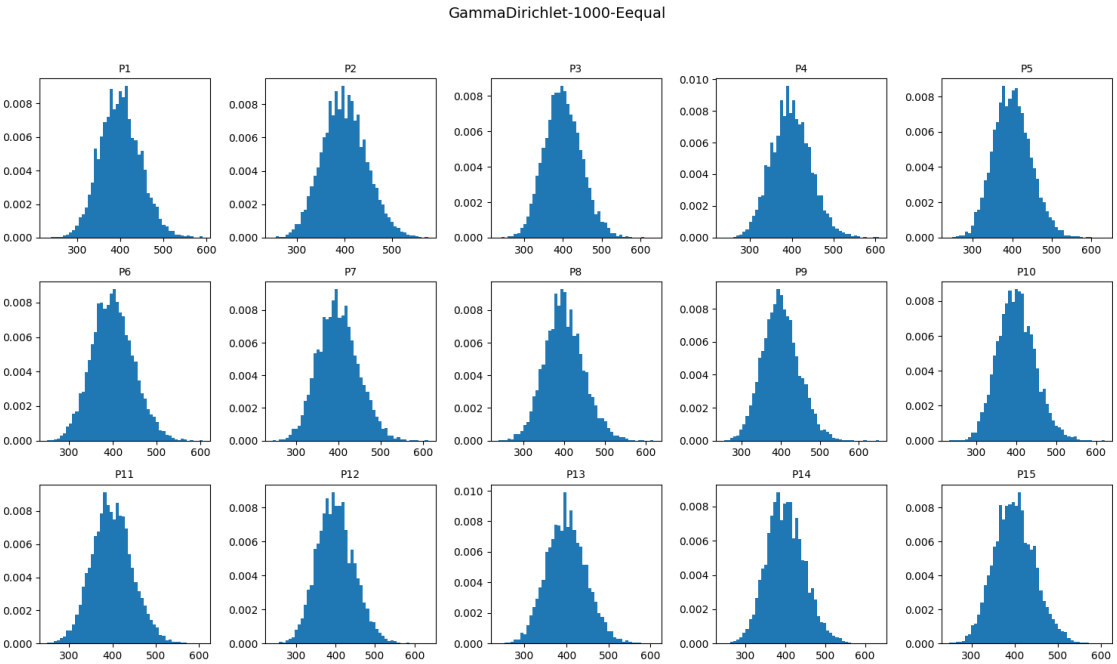
```
 399.816  499.8759 500.2473 499.2485 600.4462 599.6866 699.6816] | 400.0
Var : [ 5853.407191   11348.70971975 11205.26500476 16930.08397919
 16832.70118759 16213.91538551 22368.62862016 22260.26044399
 22207.167944   26943.23929919 27070.24654271 26957.42774775
 31750.06790556 32080.08378044 36459.69662144] | 48434.65932
Std : [ 76.50756297 106.53032301 105.85492433 130.11565616 129.74090021
 127.33387368 149.56145433 149.19872802 149.02069636 164.14395907
 164.53038182 164.18717291 178.18548736 179.1091393  190.94422385] |
220.0787570848218
```





```
GammaDirichlet-1000-Eequal:
Mean: [399.8502 399.9492 400.2472 398.6298 399.9636 399.7526 399.9535 400.2325
 400.2977 400.4827 399.6505 400.6341 399.8666 400.7804 399.7094] | 400.0
Var : [2185.07435996 2188.21881936 2257.72269216 2249.09355196 2252.60627504
 2214.53719324 2222.01853775 2234.10384375 2200.18547471 2189.70690071
 2195.26214975 2239.82081719 2241.24900444 2244.70757584 2227.89055164] |
2223.0558133333334
Std : [46.74477896 46.77840121 47.51549949 47.42460914 47.4616295  47.05886944
 47.13829163 47.2663077  46.90613472 46.79430415 46.85362472 47.32674526
 47.34183144 47.37834501 47.2005355 ] | 47.14929281901621
```

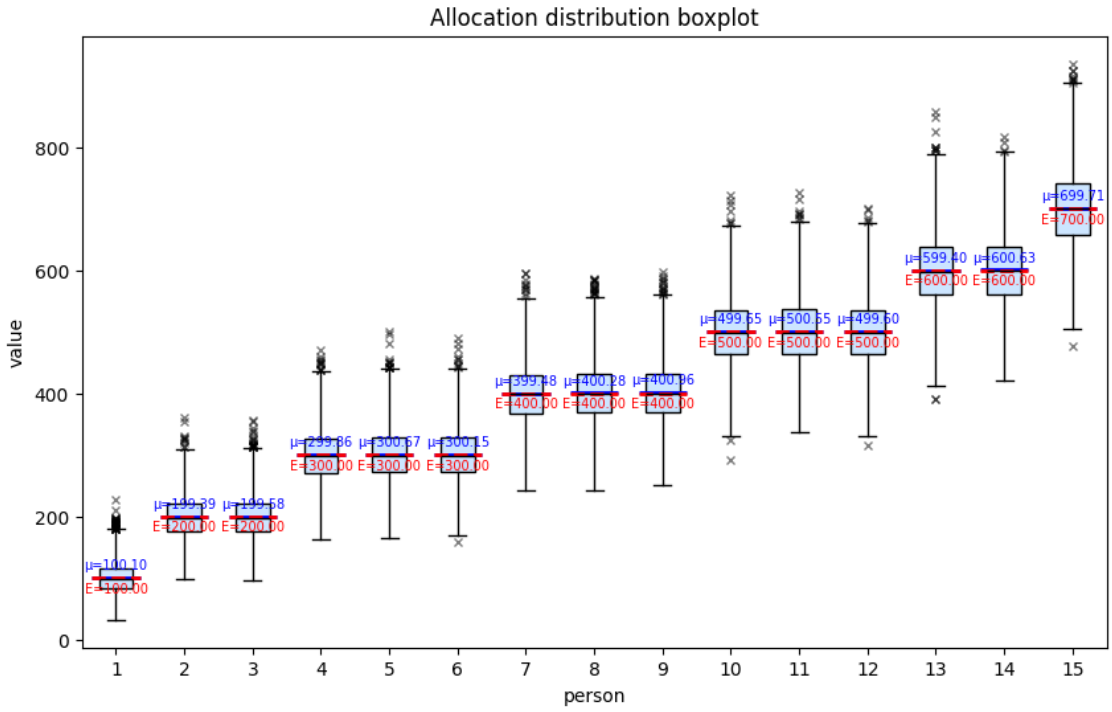


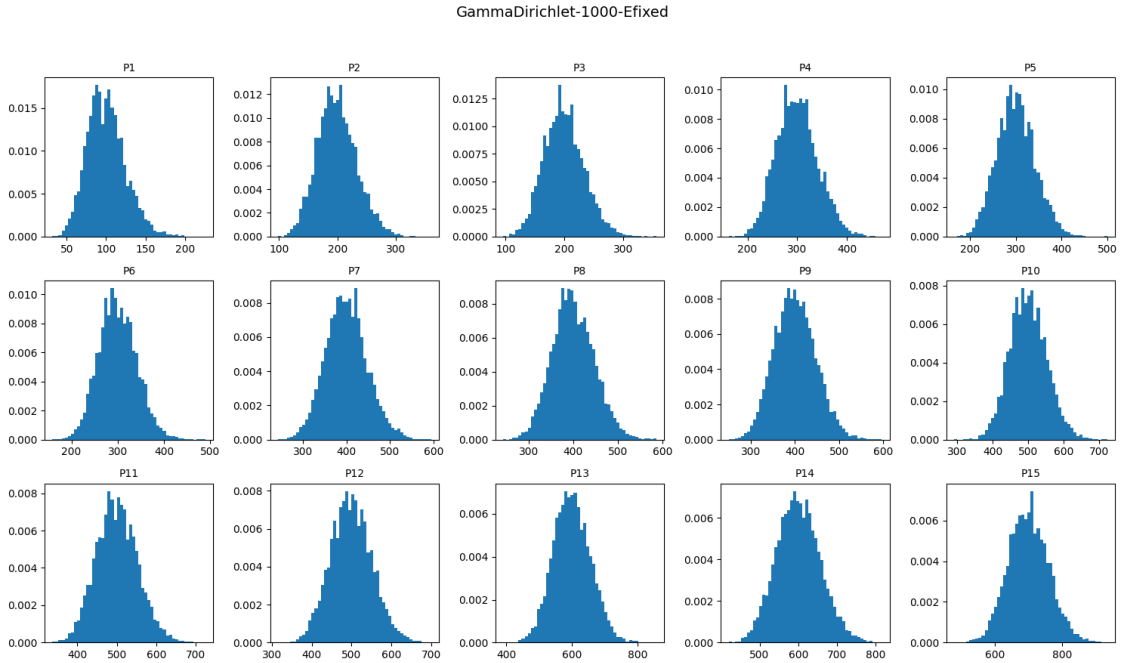


GammaDirichlet-1000-Efixed:

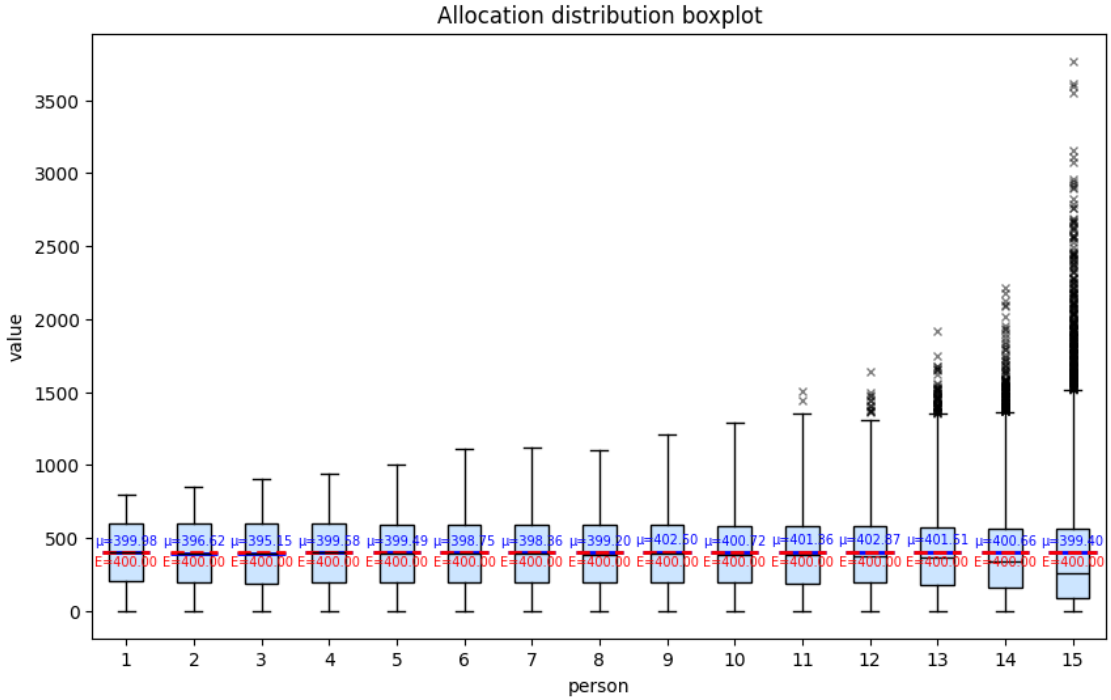
Mean: [100.1028 199.3867 199.5836 299.8558 300.6737 300.1529 399.4847 400.2756

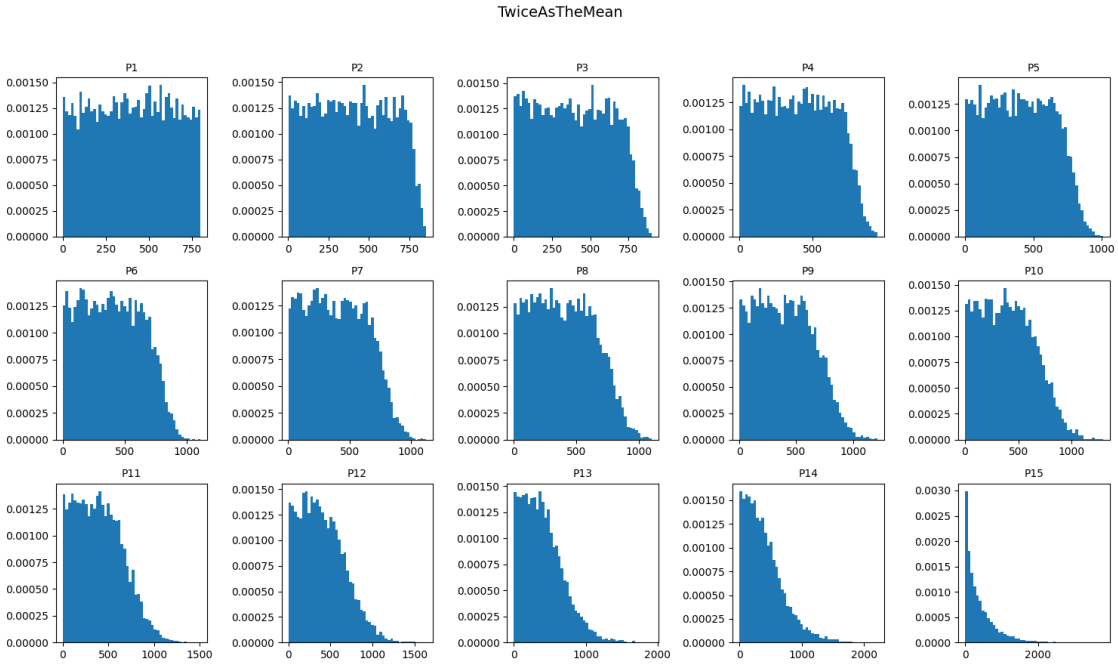
```
 400.9557 499.6504 500.5453 499.5965 599.4001 600.6278 699.7084] | 400.0
Var : [ 587.09843216 1120.08896311 1153.85401104 1698.31140636 1722.08022831
 1681.77732159 2240.94636591 2225.37964464 2198.21153751 2682.74697984
 2739.17674791 2699.20668775 3198.72241999 3196.43366716 3667.63036944] |
28854.91728
Std : [24.23011416 33.46773018 33.96842668 41.21057396 41.49795451 41.00947844
 47.33863502 47.17392971 46.88508865 51.7952409  52.33714501 51.95389002
 56.55724905 56.53701148 60.56096407] | 169.86735201326945
```





```
TwiceAsTheMean:
Mean: [399.9788 396.6215 395.1504 399.5819 399.4882 398.7524 398.3576 399.2031
 402.5036 400.7166 401.36   402.8695 401.5071 400.662  399.3979] |
399.74337333333335
Var : [ 52283.95115056  53310.94423775  54690.46737984  54708.12989239
  54615.67826076  55757.68509424  56981.15292224  57722.93085039
  59986.34738704  61157.86788444  64718.6686      69022.85526975
  79127.68434959  96659.283956   188230.20677559] | 70602.20416942063
Std : [228.65684147 230.89162877 233.85993111 233.89769108 233.69997488
 236.13065259 238.7072536  240.25596944 244.92110441 247.30116838
 254.39864111 262.72201139 281.29643501 310.90076223 433.85505273] |
265.7107528298782
```


Allocation distribution boxplot
value
person
μ=399.88 E=400.00
μ=396.62 E=400.00
μ=395.15 E=400.00
μ=399.68 E=400.00
μ=399.49 E=400.00
μ=398.75 E=400.00
μ=398.86 E=400.00
μ=399.20 E=400.00
μ=402.50 E=400.00
μ=400.72 E=400.00
μ=401.86 E=400.00
μ=402.87 E=400.00
μ=401.61 E=400.00
μ=400.66 E=400.00
μ=399.40 E=400.00


TwiceAsTheMean
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15