1- Grouping

Consider tossing 2 independent 6-sided dice. We consider the value on the dice faces as the elementary outcomes: $i=1,2,\ldots 6$. The probability of the elementary outcomes sums to unity and the dice are fair (equally probable elementary outcomes). The elementary outcomes can be grouped into disjoint sets (the k events). Let k enumerate the sum of two faces, $k=2,\ldots 12$. Thus there are $k=11$ events.

a) Find $p_{k,}$ the probability distribution of the events k. Moreover, find the entropy of the elementary outcomes and the entropy of the events k.

b) Repeat the above but increase the number of fair dice to 3, and 5. Demonstrate changes in the probability distribution as the number of dice increase.

c) Repeat the above, but consider the case that the dice may be unfair, where the elementary outcomes have unknown and unequal probabilities.

- Consider two dice. One has a mean of 3 and the other a mean of 4.
- Consider two monotonically decreasing dice – one with a mean of 1.9 and the other with a mean of 2.3.

Constrained Optimization in Python:

We use SciPy optimize library to perform constrained optimization. This library provides a minimization method. We use this method and negate our objective to solve the maximization problem. To define this function we need to specify:

1. An objective function
2. A set of constraints
3. A starting position
4. Bounds for the output

Defining the objective function to be used in numpy minimize:

The objective function is simply defined as any other function in Python. Probs which is passed to this function as an argument is one of the solutions to this function. This function in our case is defined as:

```
def objective_func(probs, sign):
  #entropy = -sum([prob * np.log(prob) for prob in probs])
  return sign * entropy(probs)
```

Computing p*:

We wrap the SciPy maximization method call in a function that we call compute_probabilities which is defined as:

```
def compute_probabilities(average_value):
        # body of the computation
```

Given an average value for a die, this function returns the probabilities of each outcome. To do so, this function tries to maximize the entropy which is defined as:

```
entropy = -sum([prob * np.log(prob) for prob in probs])
```

In other words, we are going to minimize the negation of this function.

The constraints for this computations are as follows:

1. sum(p[i]) = 1 for i in [1, 6]

*2. sum(p[i]*x[i]) = y (for a given y)*

Defining the constraints:

The above constraints can be specified in a similar way in SciPy:

```
cons = ({'type': 'eq', 'fun': lambda p:  1 - sum(p)},
          {'type': 'eq', 'fun': lambda p: sum([p[i] * (i+1) for i in range(0, 6)]) -
           average_value})
```

Starting position:

SciPy requires a starting position to compute the values. For that, we can start from a fair dice (the '.' after 6 makes sure that we get the non-integer division)

```
start_pos = np.ones(6)*(1/6.)
```

Bounds:

In addition, we need to make sure all the solutions are in the range of [0, 1]. We pass this information as a tuple of size 6 and with the ranges (0,1) and passing it to SciPy:

```
bnds = tuple((0,1) for x in range(0,6))
```

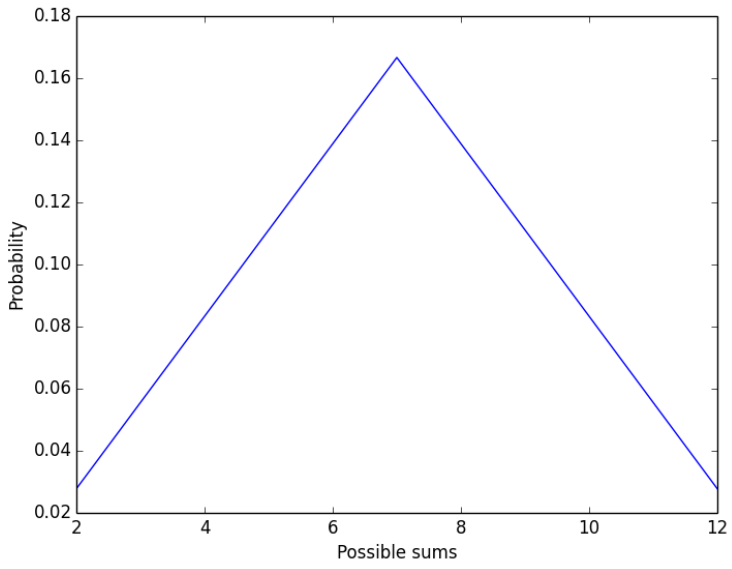
Finally:

We have all the pieces together and can call SciPy minimize function with the negated objective:

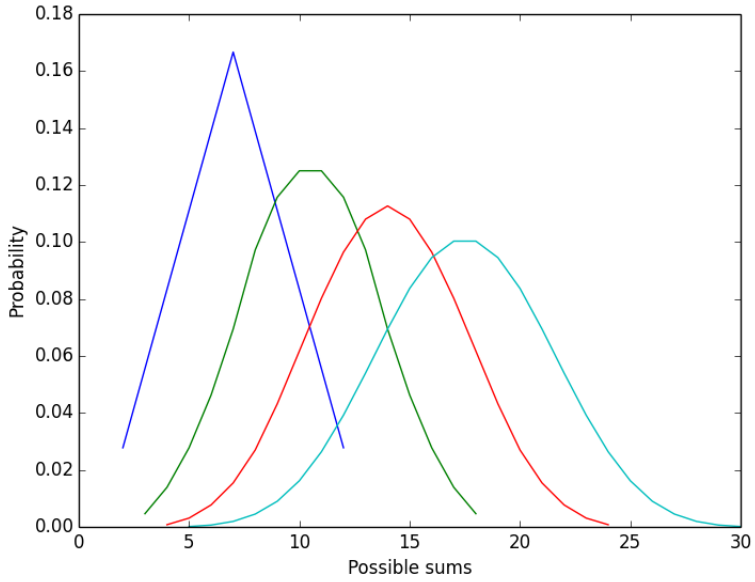
- The args = (-1.0) passes the sign to the objective function
- We are going to use the Sequential Least Square Programming
- We would like to see the detailed information about number of iterations

```
minimize(objective_func, start_pos, args=(-1.0,),
         constraints=cons, method='SLSQP', bounds=bnds)
```

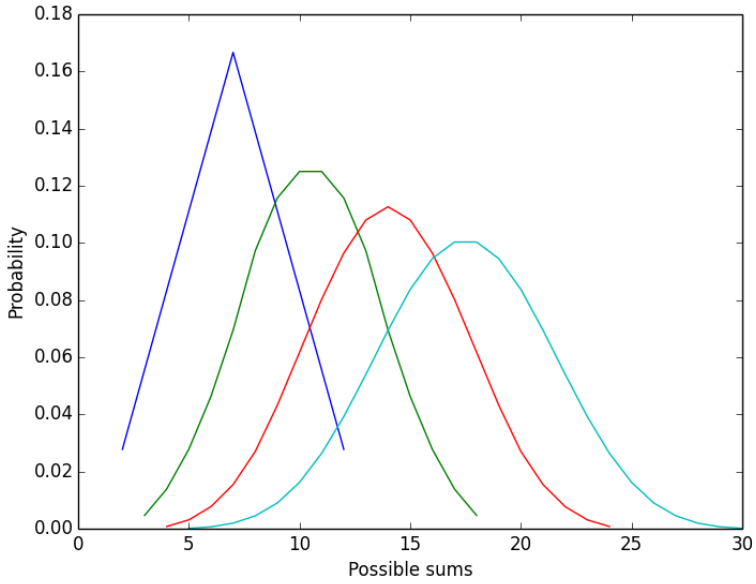
In the compute_probabilities function we generate PyPlot graphs as well as shown below. First, we show the distribution for two dice with mean value of 3.5:



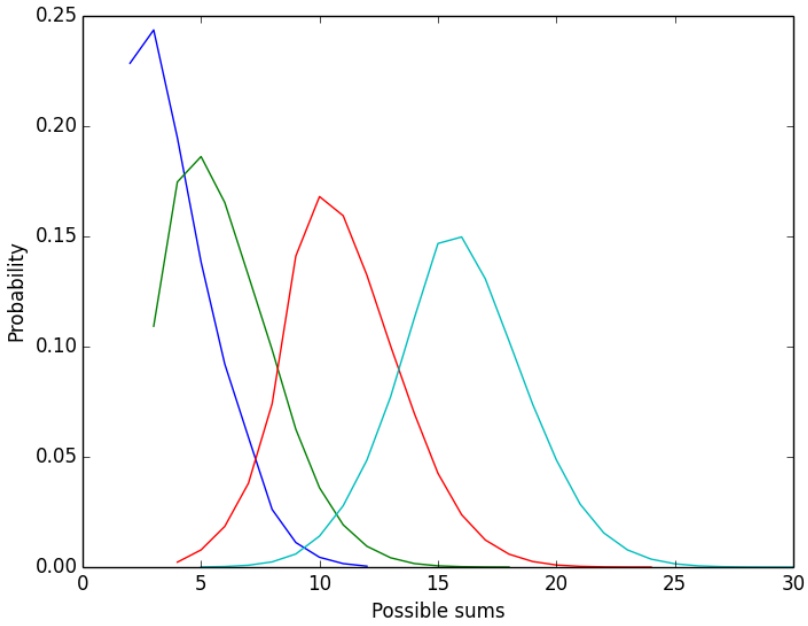
And then expand it to more dice up to 5 dice again with mean value of 3.5:



We can also define unfair dice with mean value of 5:



In addition, we can define each die with different mean value. In the following figure we have used mean values [2, 2, 2, 5, 5] for five unfair dice:



2- Income Distribution based on Tax Brackets

Consider the problem of estimating the income distribution of the households in the US. There are $k=1, \ldots, 7$ different tax brackets.

The observed information we have is the taxable income for single filers, married filers, and head of household filers for each tax bracket.

Brackets	**Single Filers**	**Married Joint Filers**	**Head of Household Filers**
10%	$0 to $9,225	$0 to $18,450	$0 to $13,150
15%	$9,225 to $37,450	$18,450 to $74,900	$13,150 to $50,200
25%	$37,450 to $90,750	$74,900 to $151,200	$50,200 to $129,600
28%	$90,750 to $189,300	$151,200 to $230,450	$129,600 to $209,850
33%	$189,300 to $411,500	$230,450 to $411,500	$209,850 to $411,500
35%	$411,500 to $413,200	$411,500 to $464,850	$411,500 to $439,000
39.6%	$413,200+	$464,850+	$439,000+

Assume that the true (unknown) income distribution is

10%	15%	25%	28%	33%	35%	39.6%
0.271	0.418	0.237	.046	0.018	0.002	0.008

a. Derive the ME model for that problem.
b. Estimate the income distribution using only the data on the single filers.
c. Re-estimate the income distribution using only the data on the married filers.
d. Re-estimate the income distribution using all of the available information (single, married, and head of household filers).

e. Repeat the above but this time generate the observed information using Uniform distribution and add noise, using Normal (0,3). Compare the results.

Constructing means of the constraints:

First, we construct the means of each one of the constraints using the true distribution and for the highest bracket we take the value given instead.

```
def expected_value(probs, values):
  s = 0
  for i in range(0, len(probs)):
     s += probs[i] * values[i]
  return s
```

Uniform and Normal Distributions:

In the next step, we create random values in each computed value, add noise and compute the average of the generated distribution:

```
def compute_random_average(low, high, samples):
   dist = np.random.uniform(low, high, size=samples)
   noise = np.random.normal(0, 3, samples)
   for i in range(0, len(dist)):
      dist[i] = dist[i] + noise[i]
   return np.average(dist)
```

The rest of the process is similar to the previous problem. The results are shown below:

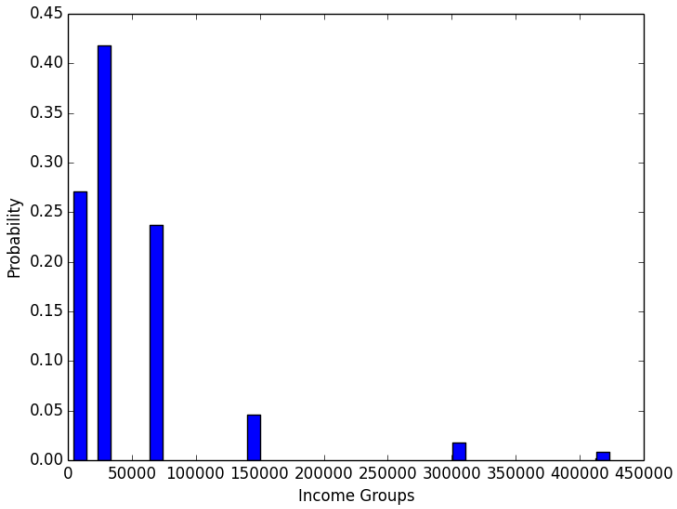


Figure 1-Income Distribution (True Probabilities)

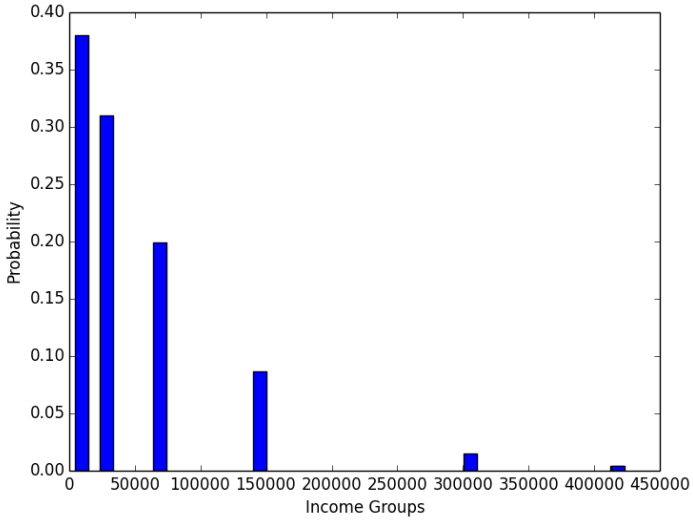


Figure 2- Income Distribution (Estimated Probabilities)

In the following table we have compared *True p* and the *p** for single filers data:

Tax bracket	True p	P* using data on single filers
10%	0.271	0.380
15%	0.418	0.310
25%	0.237	0.199
28%	0.046	0.087
33%	0.018	0.015
35%	0.002	0.004
39.6%	0.008	0.004

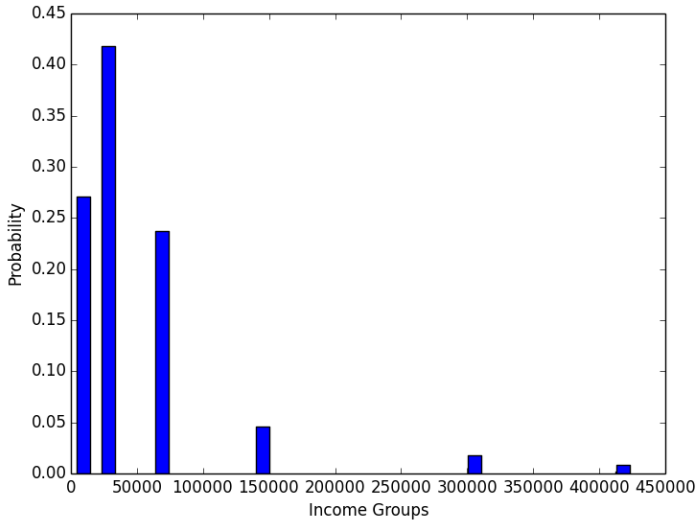


Figure 3- Income Distribution (True Probabilities)

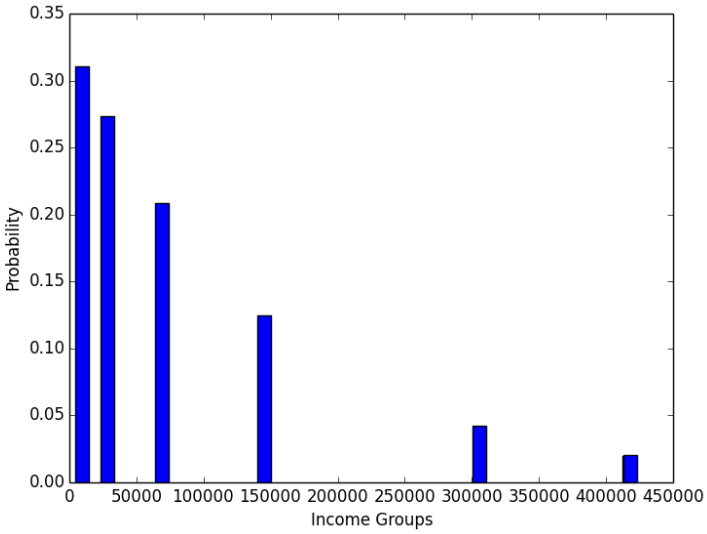


Figure 4- Income Distribution (Estimated Probabilities)

In the following table we have compared *True p* and the *p** for married filers data:

Tax bracket	True p	P* using data on married filers
10%	0.271	0.311
15%	0.418	0.274
25%	0.237	0.208
28%	0.046	0.125
33%	0.018	0.043
35%	0.002	0.020
39.6%	0.008	0.020

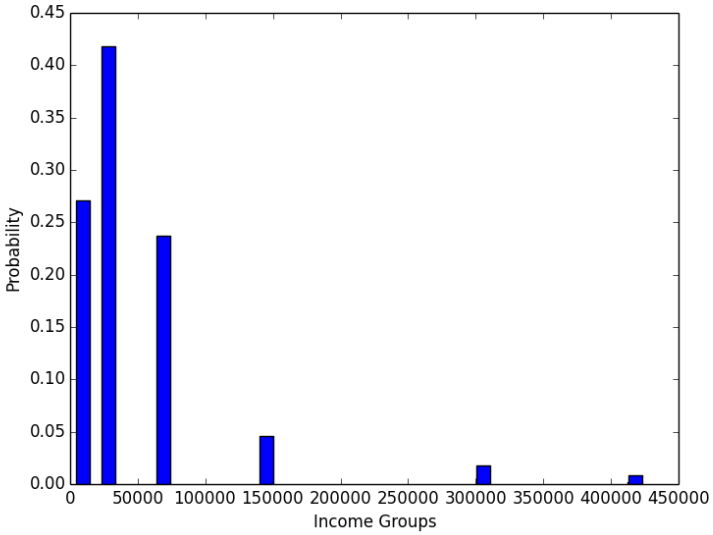


Figure 5- Income Distribution (True Probabilities)

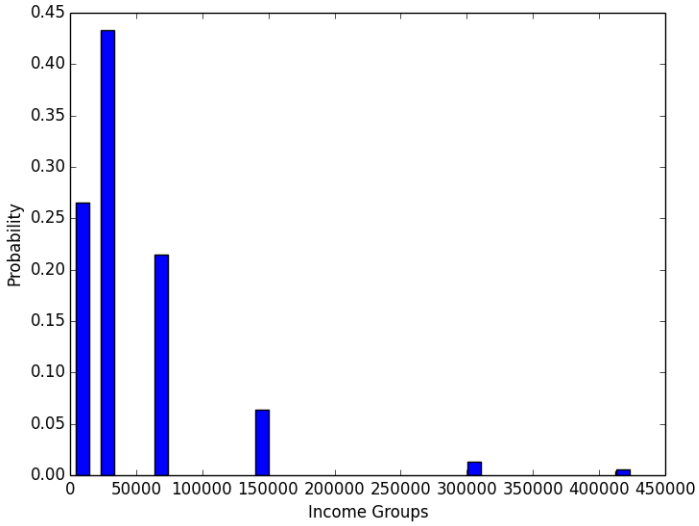


Figure 6- Income Distribution (Estimated Probabilities)

Finally, in the following table we have compared *True p* and the *p** for all three pieces of information:

Tax bracket	True p	P* using 3 piece of information
10%	0.271	0.265
15%	0.418	0.433
25%	0.237	0.215
28%	0.046	0.064
33%	0.018	0.013
35%	0.002	0.004
39.6%	0.008	0.006