Exercise 2: Ranking binary numbers with a perceptron

A **perceptron** takes two binary numbers of 10 digits each and returns either +1 or -1.

$\vec{S}$ has 20 elements (± 1). The first 10 elements are the **top** dataset n_t and the others are the **bottom** dataset n_b.

$$\sigma(\{S_i^\mu\} = \{\text{sgn}(\sum_i S_i J_i)\})$$

The goal of the perceptron is to **tell whether** $n_t > n_b$ **or viceversa**.

Step 1: Set up the perfect perceptron

Let us define the perfect perceptron as the function that given the following weights array

```
array([ 512.,  256.,  128.,   64.,   32.,   16.,    8.,    4.,    2.,
          1., -512., -256., -128.,  -64.,  -32.,  -16.,   -8.,   -4.,
         -2.,   -1.])
```

returns the dot product between the weights and the input array

$$\sigma = \vec{w} \cdot \vec{s}$$

which amounts to comparing the two numbers in binary representation.

Step 2: Define and evaluate a test error ϵ of the PP

We compare said number with the true value, which we choose to be the function such that it returns +1 if the integer comparison between n_t and n_b is such that $n_t \geq n_b$, and -1 otherwise.

In a sense, we are checking whether the comparison in decimal representation is the same as the comparison in binary representation.

$$\sigma_{\text{true}} = \text{sgn}(n_t - n_b)$$

except when $n_t = n_b$ this number yields +1.

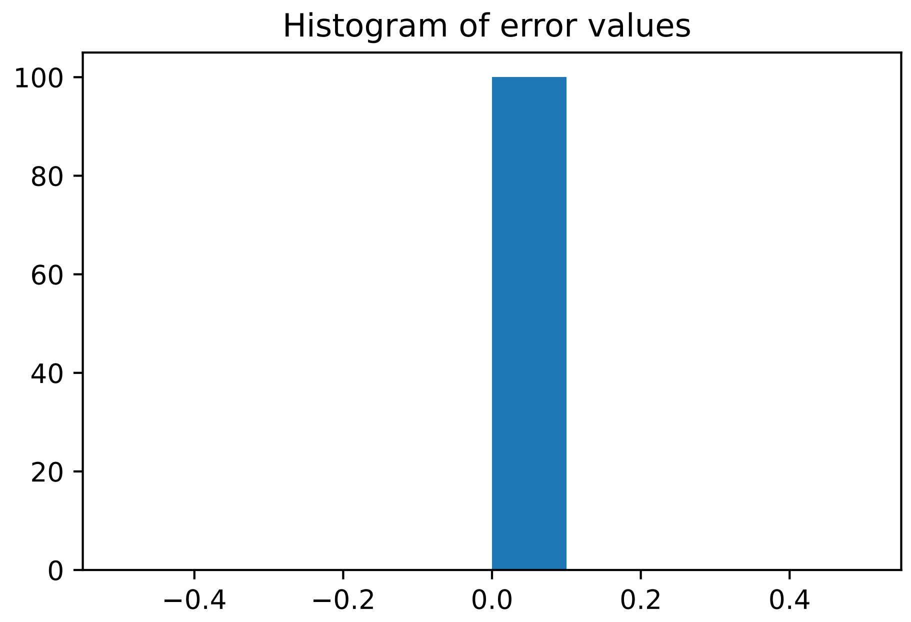
Notice that I have inserted the definition for the **equality** case, otherwise it would get an error point due to the fact that the binary comparison rule does not account for the equality case.

```
inputs =
 [[0 1 0 ... 0 0 1]
 [1 1 1 ... 0 0 1]
 [0 1 0 ... 0 1 1]
 ...
 [1 1 0 ... 1 1 1]
 [0 0 1 ... 1 1 1]
 [0 1 0 ... 0 1 1]]

inputs is a matrix of shape (100, 20)


weights =
 [ 512.  256.  128.   64.   32.   16.    8.    4.    2.    1. -512. -256.
 -128.  -64.  -32.  -16.   -8.   -4.   -2.   -1.]

weights is a matrix of shape (20,)
```

We may define a trivial error measure such as

$$\epsilon = |\sigma - \sigma_{\text{true}}|$$

Let us compare, for example, 100 instances of this comparison.



We see that we have exactly error 0 on the 100 runs, each with 100 samples. The **perfect perceptron** is, in fact, perfect (with the caveat of the definition of the *border* situation). The error is a delta function in 0, which is reasonable.

Step 3: Assign random synaptic weights

Until now we had a *perfect perceptron*, meaning that we used a perceptron where we knew the model (the knowledge of the specific weights that would reproduce the comparison behaviour) that would trustfully yield the correct data.

Now let us show instead a real learning process, where we do not have any prior knowledge. Weights are, in fact, extracted from $\mathcal{N}(0, 1)$.

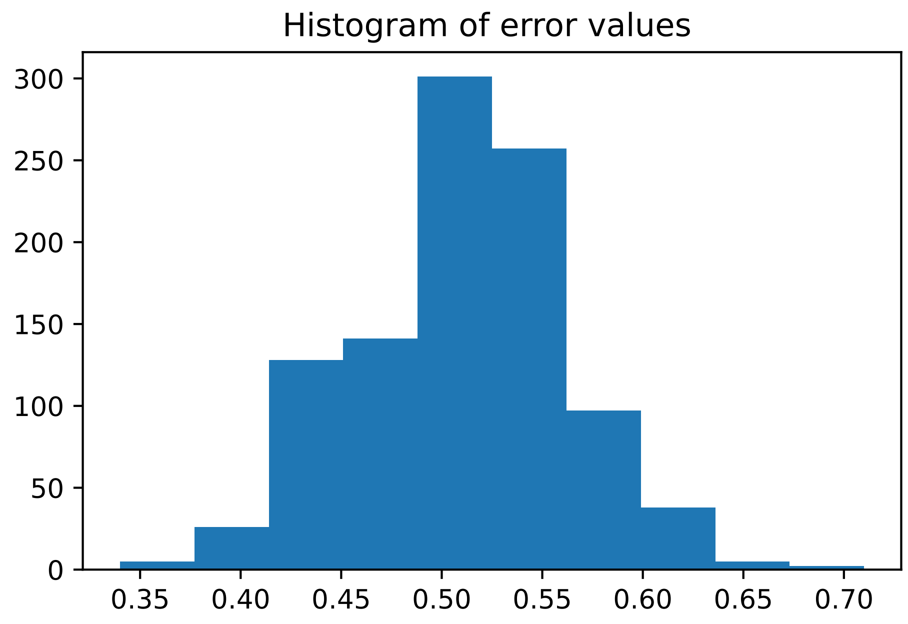
Our previous perceptron had 20 weights. Likewise, we find 20 Gaussian weights in this situation as well.

```
inputs =
 [[1 1 1 ... 0 1 1]
 [1 0 0 ... 0 1 1]
 [0 1 1 ... 0 0 1]
 ...
 [0 0 1 ... 0 0 1]
 [0 0 0 ... 1 0 1]
 [0 1 1 ... 0 1 0]]
```

```
inputs is a matrix of shape (100, 20)


weights =
 [-1.13647016  0.56230051  1.29375658  1.46575574  0.50626951 -0.64551866
 -0.67421973  0.03989592  0.62494372  0.27636509 -0.60234094  0.73615364
  0.05056666  0.65932691 -0.97182693  1.55816665 -1.54780708 -0.52608695
  0.08600766 -0.25694034]


weights is a matrix of shape  (20,)
```

For now we have not done any learning, so I do not expect a good result. Instead, I expect that half of the times we get the right answer, and the other half of the times we get it wrong.

Let us repeat the evaluation with the errors as we have done in Step 2.



We notice that over many runs (100 runs), each made of 100 samples, the (relative) error is centered around 0.5, meaning that the hypothesis that about half of the times we get it right and half wrong was correct.

Step 5: Produce $\vec{\xi}^\mu$ for learning

So we until know have used the so-called *perfect perceptron* and the *random perceptron*. Let us know treat the case for learning. In order to do this we should produce a **training set**.

The problem is the following: we have p input vectors $\vec{\xi}^\mu$, where $\mu = 1, ..., p$. These p vectors are a random subsample of the configuration space. Each μ-th data point has σ_T^μ value ± 1, which is the *label*.

We want two datasets, dataset 1 and 2, respectively containing $P_1 = 500$ and $P_2 = 2000$ points, containing two numbers (the x, encoded as a vector 20 binary digits long) and a value ± 1 (the y). We can use the decimal conversion as well to create the dataset, since we know that the perfect perceptron coincides with the integer comparison from Step 2.

We should however redefine the comparison, remember the caveat about the $n_t = n_b$ case. We choose to include it with the $>$ case.

Step 6.1: Training with the first set

Let us again initialise some random weights.

```
weights =
 [ 1.76379023  1.47243731  0.54408697  1.11878767  0.90008723 -1.72742265
  0.56355411  0.86023426  0.42489023 -0.4028883   1.25945539 -1.42490485
 -0.64684433  0.2881261   0.10075826  0.68808102 -0.30957092 -0.95896245
 -1.71561502 -1.140369  ]


weights is a matrix of shape  (20,)
```

And let us now update the weights for learning. We are using the following rule

$$\vec{J}(t+1) = \vec{J}(t) + \frac{1}{\sqrt{N}} \sigma_T^\mu \odot \xi^\mu \quad \text{if} \quad \sigma^\mu = \sigma(\vec{\xi}^\mu, \vec{J}) \neq \sigma_T^\mu$$

which is *kind of a* gradient descent. In fact if the true value and the predicted value are not the same, we try to go to a *lower energy* situation by adding a contribution to the weights given by the *element-wise product* $(\cdot)$ between the current training data input x^μ instance and the true label σ_T^μ. This *pushes* the values of the weights in the p-dimensional direction where the true values are lying.

The error measure we use for each run is the following:

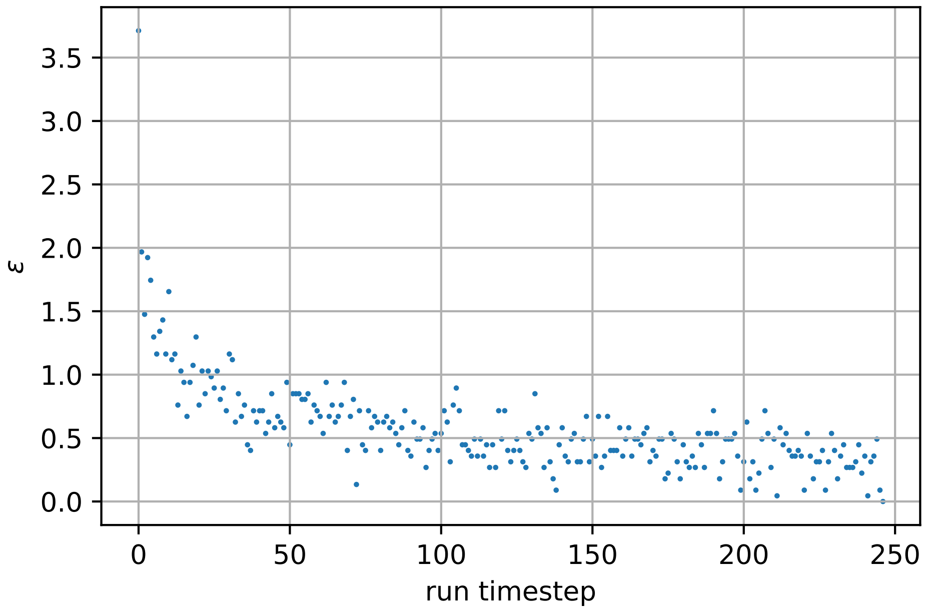
$$\epsilon = \frac{|\sigma_{\text{true}} - \sigma_{\text{pred}}|}{2}$$

We perform several runs N_{runs}, and in each run we scan through the entire P-sized dataset. The general error is

$$\epsilon_{\text{tot}} = \frac{\sum_{j=1}^{N_{\text{runs}}} \epsilon_j}{\sqrt{N_{\text{runs}}}}$$

We use a naif stopping condition (we stop whenever the error reaches a zero value once), so we are not reaching *full convergence*, but it should be a sufficient convergence criterion for this simple example. Let's see.

For each time step the convergence process of ϵ is the following:

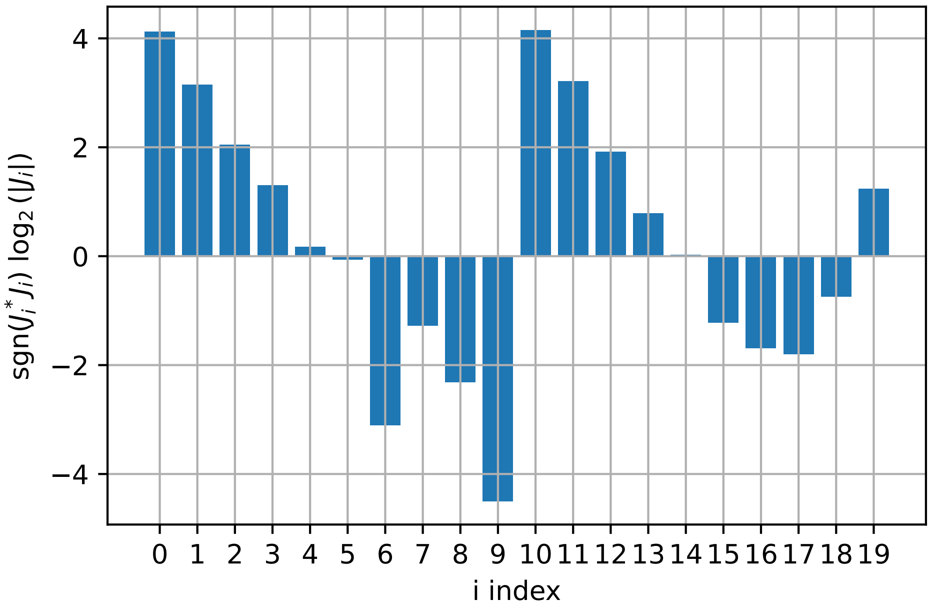


Let's see our weights after they have been updated during the training phase.

```
weights =
 [ 17.41626607   8.85146164   4.12179574   2.46042846   1.12369402
   0.95585892   0.11634051   0.41302067   0.20128343   0.04432529
 -17.74712242  -9.25114277  -3.7773395   -1.72433508  -1.01727573
  -0.42995297  -0.30957092  -0.28814206  -0.59758103   0.42487858]
```

```
weights is a matrix of shape  (20,)
```

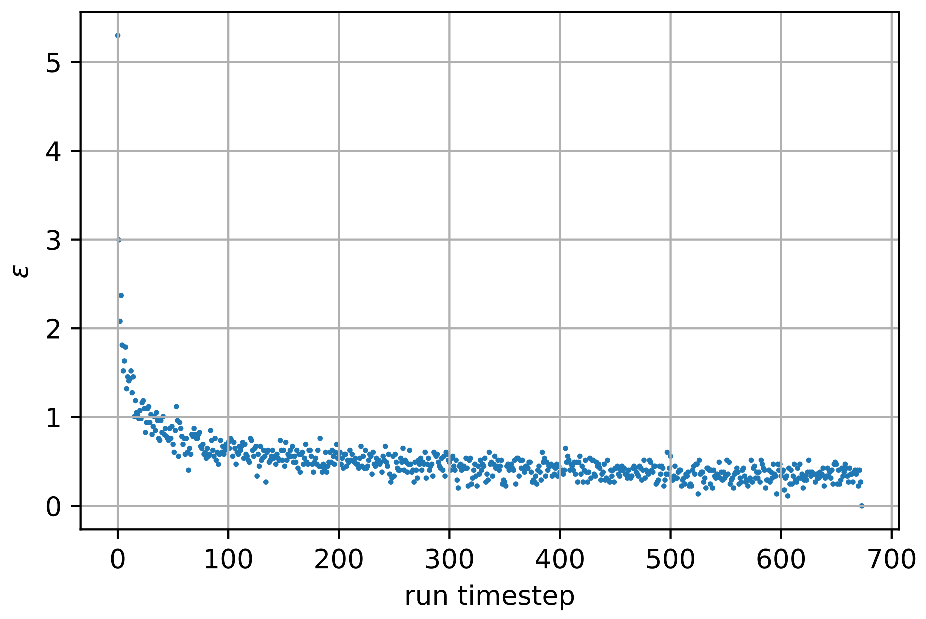
```
array([ 3.67596435e+04,  7.12856376e+03,  1.07801011e+03,  2.04535941e+02,
        6.04994171e+00, -9.96087520e-01, -2.88857160e+00, -2.10758519e+00,
       -9.31016257e-01, -1.99274377e-01,  3.77046625e+04,  7.60134658e+03,
        9.27047573e+02,  8.67453831e+01,  8.04405426e-01, -8.37719841e+00,
       -4.18950534e+00, -2.06903037e+00, -8.87758899e-01,  5.24672991e-01])
```



Step 6.2: Training with the second set

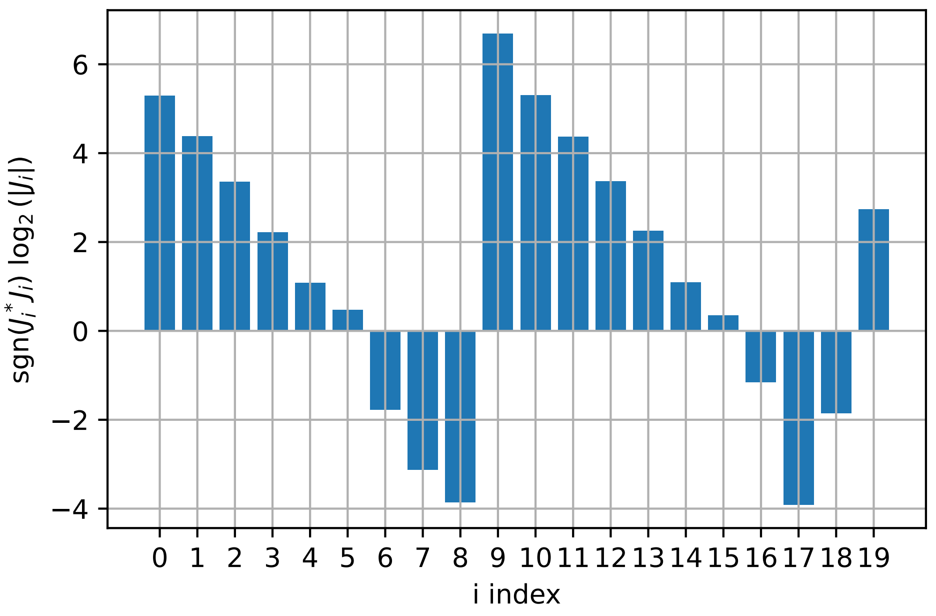
```
weights =
 [-0.53103905 -0.1459637   0.61457526 -0.48655697  0.09529066  0.04374121
  1.41039967 -0.33302403 -0.15440486 -0.45691921 -0.99647201  0.76041124
 -0.2258336   0.3849887   0.33370523 -1.27088678 -0.89588083  0.3805874
  1.95865632  1.04459835]
```

```
weights is a matrix of shape  (20,)
```



Let's see our weights after they have been updated during the training phase.

```
weights =
 [ 3.92709710e+01  2.08730753e+01  1.02296676e+01  4.65639938e+00
  2.10775184e+00  1.38538200e+00  2.92365681e-01  1.14189569e-01
  6.92019412e-02 -9.70561772e-03 -3.94568412e+01 -2.07058413e+01
 -1.02881395e+01 -4.75796765e+00 -2.12596955e+00 -1.27088678e+00
 -4.48667230e-01 -6.66262005e-02 -2.77411656e-01  1.50171164e-01]


weights is a matrix of shape  (20,)
```



Step 7: Comment on the metrics of the training step

The metrics used in Step 6 the most relevant weights are more or less in descending order 0, 1, 2, ... and in a parallel manner entries with i given by 10, 11, 12, ... This makes sense, since it means that binary positions with higher weights are more relevant in deciding whether a number is bigger or not with respect to another. As we get into positions referring to smaller numbers, we loose information (i.e. we do not have anymore a monotonically decreasing behaviour).

This statistics becomes more consistent as the size p of the training set is increased, as we can see by comparing the plots of the two cases $p = 500$ and $p = 2000$. From a computational point of view, as expected, the training of the bigger set takes more time to be completed.

Step 8: Evaluate the test error ϵ on the trained perceptrons

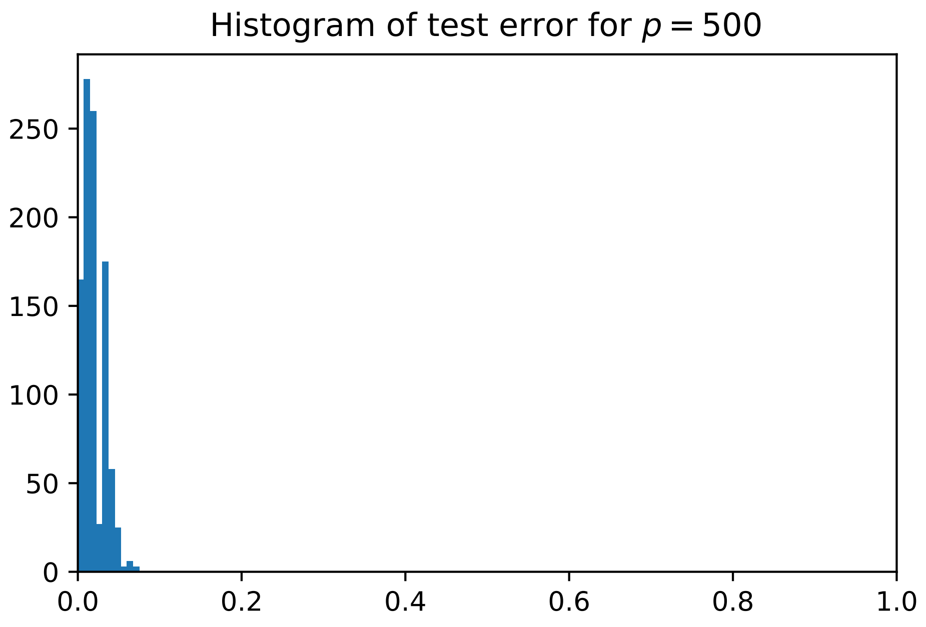
Now we evaluate the trained perceptrons over unseen test datasets. How they will perform?

We use the usual error measure

$$\epsilon = \frac{|\sigma_{\text{true}} - \sigma_{\text{pred}}|}{2}$$

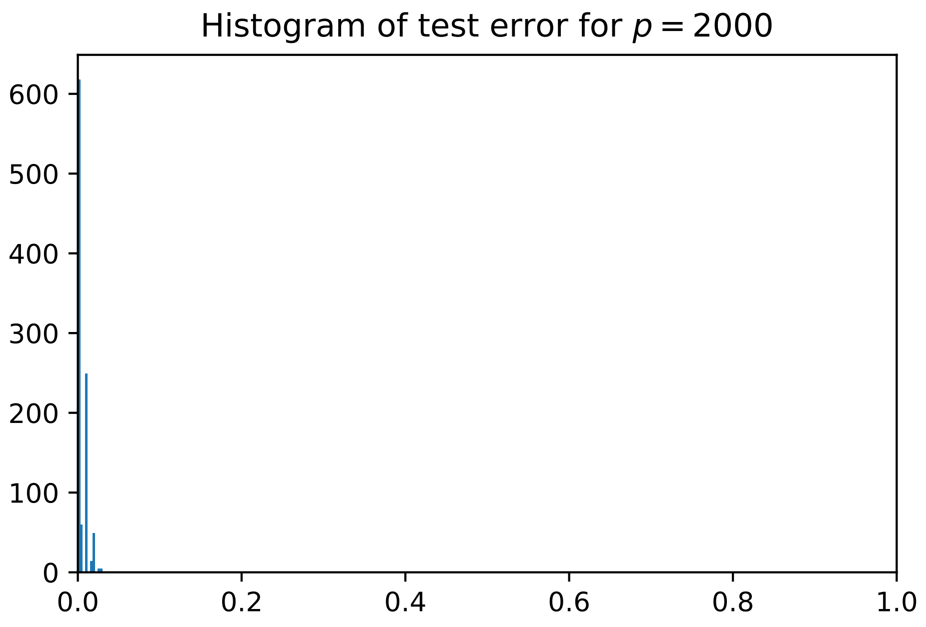
this time normalized also over the number of samples in the current run, in order to get statistics.

We can see this time that the distribution of errors over the 1000 runs is now much improved with respect to the random case, where recall that we saw a peak around 0.5 (half of the time we got it right, half wrong). Here instead we have a peak at values < 0.1, meaning that we have much improved performances.



The distribution of the test errors for the $p = 2000$ case, with 1000 runs is given by the following histogram.

By increasing the size of the training set the generalization gets better and better, reaching really small values as it can be seen in the following plot.


Histogram of test error for $p = 2000$
600
500
400
300
200
100
0
0.0
0.2
0.4
0.6
0.8
1.0