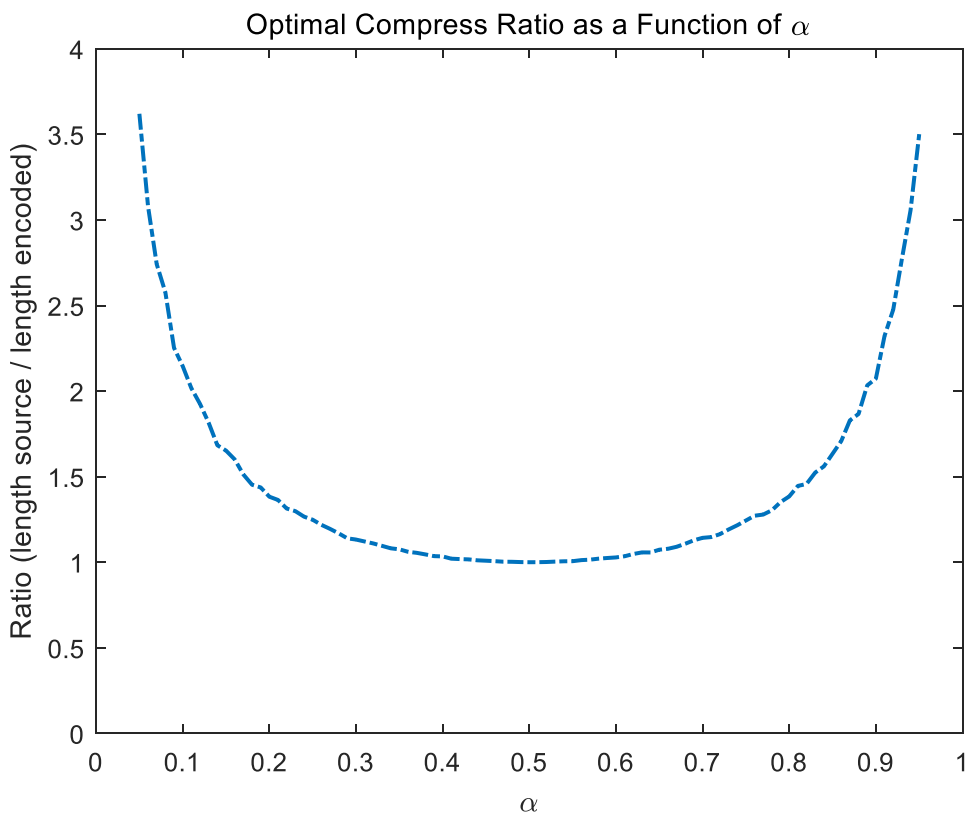
Problem 2

(a) *binaryToRunlength.m* is the function that can fulfill such requirements. The input *binaryString* should be a binary string. There are two return values: 1. *startBit* indicates what the input string start with, 2. *run-lengths* is a vector that stores the run lengths.

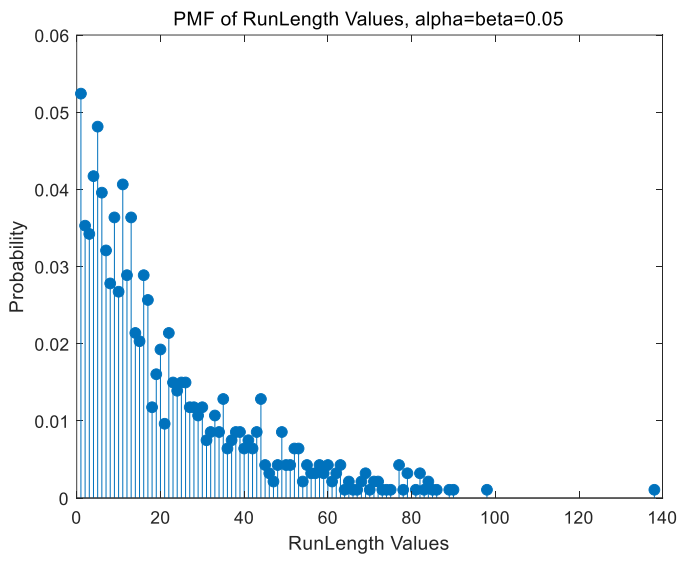
(b) *computeOptLength.m* is the desired function. There are 3 return values in this function. One of them is *bitLength,* which corresponds to 'the length (in bits) of the optimum binary stream'. The theoretical optimal code length for each symbol i is $l_i^* = -log_2 p_i$. Based on the optimal code length, we can further compute the length of the stream by multiplying every symbol in the runlength stream with its optimal code length and summing the results up.

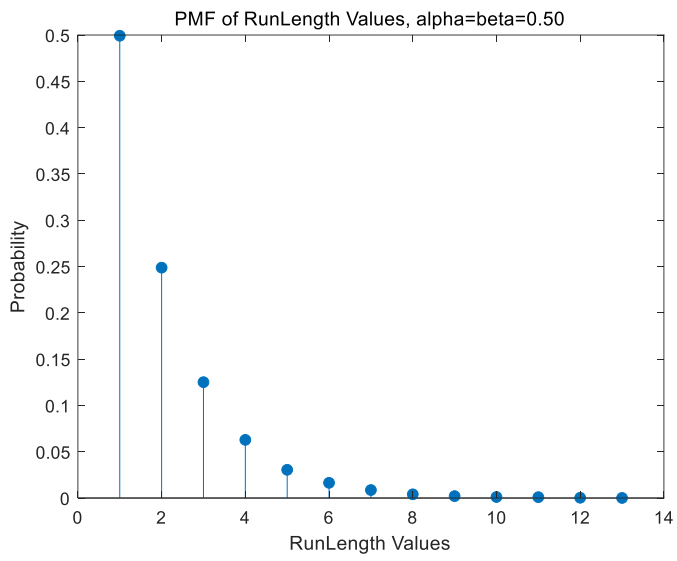
(c) The function for generating Markov-1 string of 0s and 1s, *generateMarkov1String.m*, is copied from the last Homework Assignment. The code in *Q2_c.m* fulfills the rest instructions. Here is the plot of compression ratio with respect to α, **under the theoretically optimal coding scenario as described in (b)**:

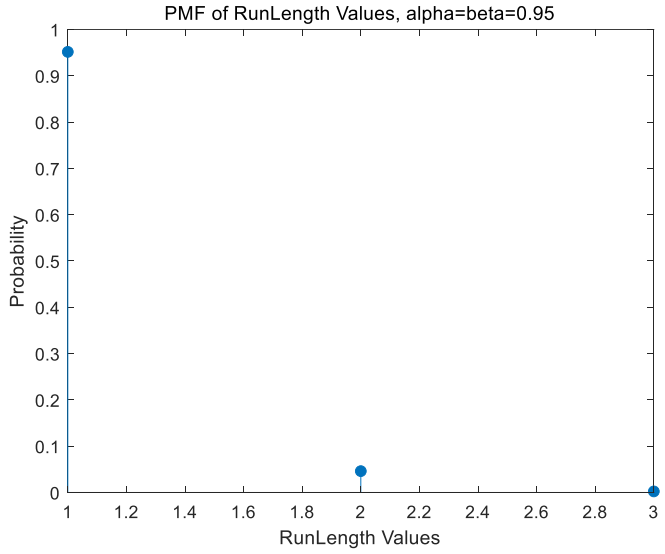


As α goes up, the ratio first decreases and then increases, that is, the compression efficiency first increases and then decreases. At $\alpha = 0.5$, we barely compress nothing. This agrees the theory because a Markov-1 source with $\alpha = \beta = 0.5$ is just a random source with two equally likely outputs (maybe identical to a Bernoulli source with p=0.5), in which we cannot conduct any compression. When α is much smaller or much bigger than 0.5, the next output of the source is highly correlated to its predecessor, which means we can utilize these correlations to conduct efficient compression. For instance, for $\alpha \to 0$, there are only several large numbers in the run-length code, encode these numbers using optimal code should result in very short length of code. For

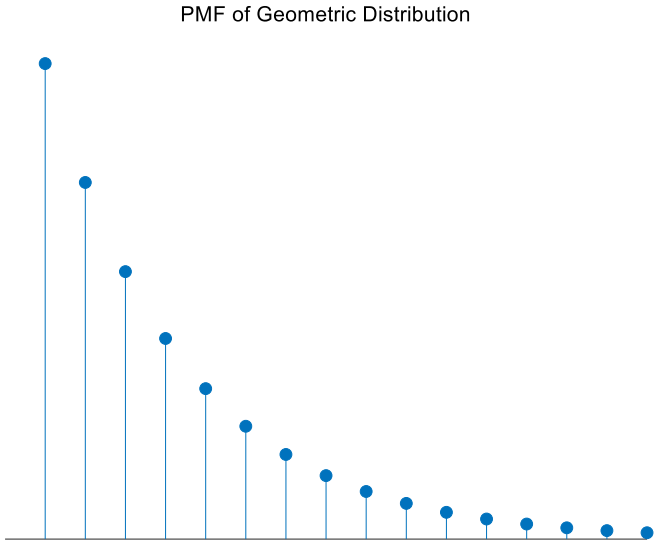
$\alpha \to 1$, the run-length code contains mostly 1s, where the entropy of the source is very small ($P_1 \approx 1, P_{others} \approx 0, H(X) \approx 0$), which leads to very short length of code. Hence, we can observe "symmetry" in the above figure.







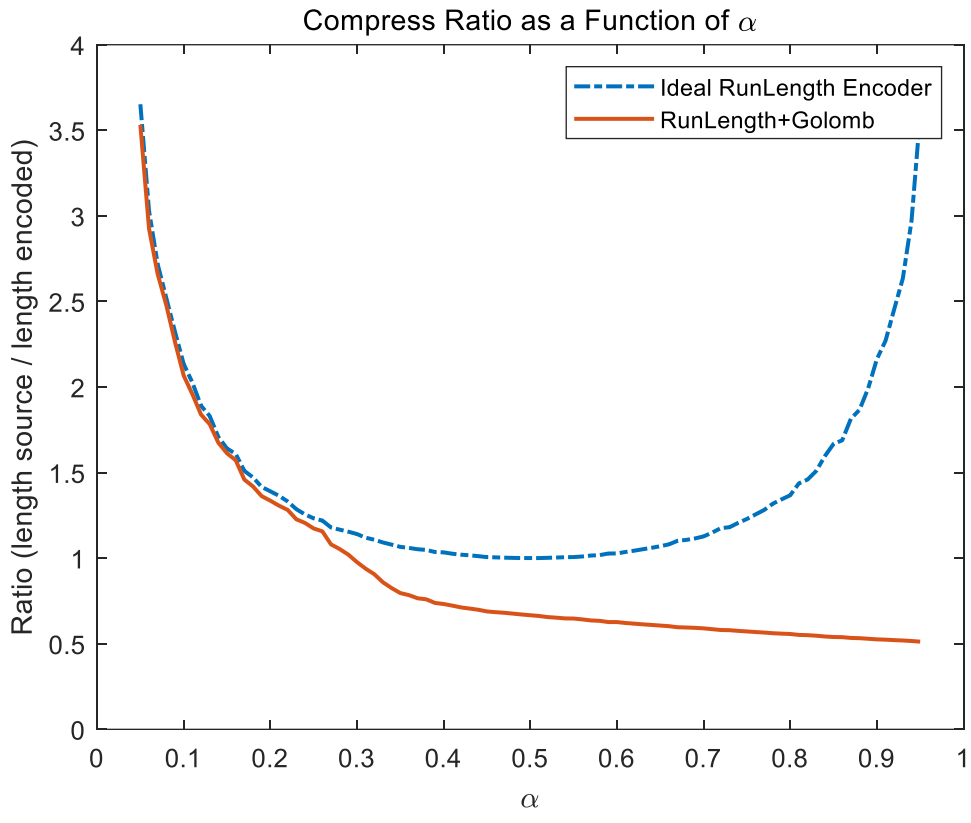
From the figure, we know that the higher α, the lower chance of occurrence of consecutive 0s or 1s. This agrees the definitions of Markov-1 source, where the α and β represents the chance of the next output is identical to its predecessor. For 0.05 case, we can even observe, though in very low probability, a block of nearly 140 consecutive characters. But for 0.95 case, the block of 4 consecutive characters does not even exist in this simulation. Moreover, the PMFs, especially for the cases when α is small, are like the geometric distribution in below figure, inspiring me to regard them as geometric distributions.



Problem 3

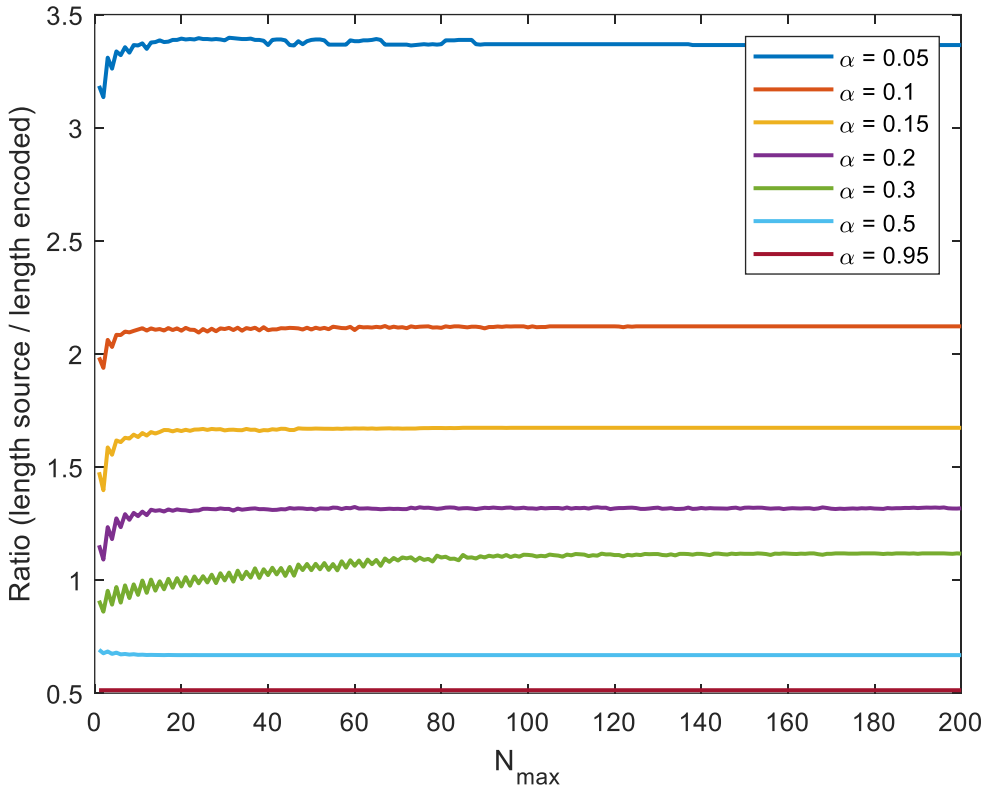
(a) *encodeGolomb.m* can encode run-lengths to adaptive Golomb code.

(b) *Q3_b.m* is for this sub-problem. The compress ratio with respect to α is as follows:



The plot shows that for binary Markov-1 source with small α, run-length + adaptive Golomb code can effectively compress the source, and even approach the (ideal) optimal code length. But for large α, the curve for the proposed scheme deviates much from the ideal one. This is easy to illustrate, considering the extreme case when $\alpha = \beta = 1$, where the source stream differs in every two consecutive bits, for example, we have 6-bit source stream [1 0 1 0 1 0]. Encode it to run-length code we have [1 1 1 1 1 1] $-1 \rightarrow$ [0 0 0 0 0 0], with another start bit 1. We can see every element in the run-length code is 0. Encode 0 by the given Golomb rule will at least produce two bits (i.e., 10). So, we have the shortest length of Golomb code [1 0 1 0 1 0 1 0 1 0 1 0], which is much longer than the source stream. This is even for a fixed k, if k is adaptive, under the above circumstance, the resulted Golomb code will be even longer. Generally speaking, as α grows larger, there will be less large run-length values in the run-length code, the "size" of the run-length stream will be closer to the "size" of the source stream. Encode such a run-length stream by Golomb code will be inefficient. This can be seen from the above plot, when α is large, we do not compress the source.

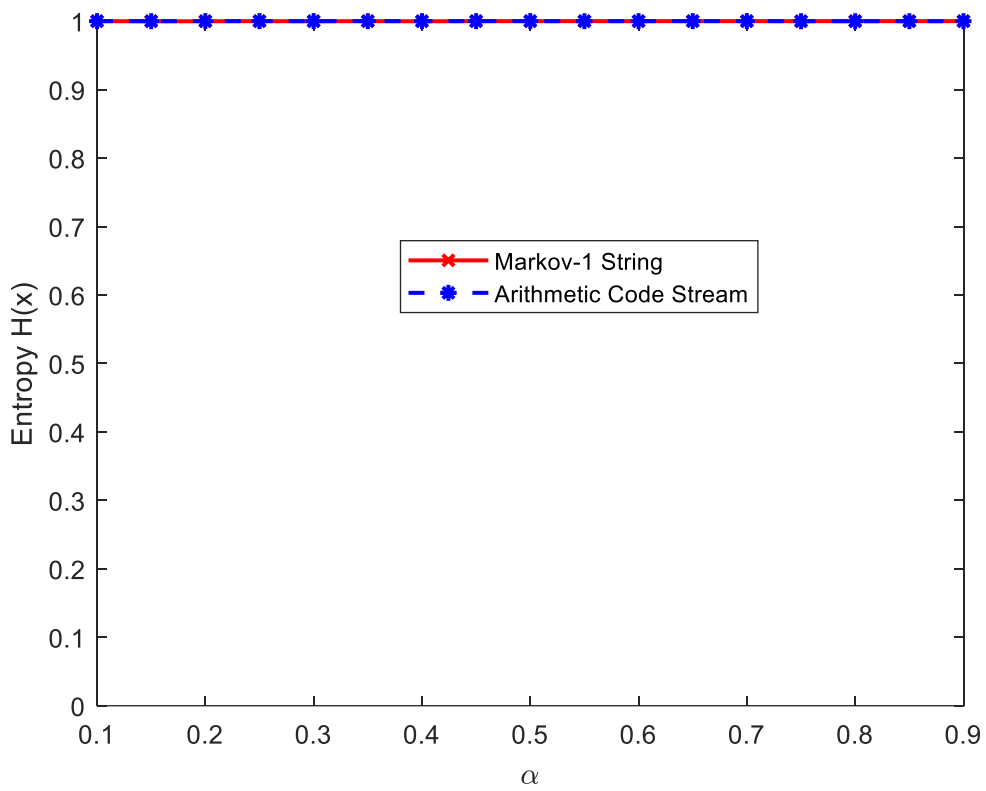
For the choice of parameters, I take A as the average value of the input run-lengths, as suggested in the problem description. For Nmax, If Nmax is set too small, the value of A will change frequently, which may lead to instability in the encoder's estimation of the true average run-length. If Nmax is set too large, the encoder may not be sufficiently responsive to changes in the run-length distribution. Below plot (from *test_scripts/ test_find_Nmax.m*) shows how the selection of Nmax affect the compression ratio under different α. We can conclude that Nmax=20 is a good-enough choice.


Ratio (length source / length encoded)
N max
α = 0.05
α = 0.1
α = 0.15
α = 0.2
α = 0.3
α = 0.5
α = 0.95
3.5
3
2.5
2
1.5
1
0.5
0
20
40
60
80
100
120
140
160
180
200

Problem 4

(a) *encodeArithmetic.m* satisfies the requirement for sub-problem (a). We have another *encodeArithmeticMarkov1Modi.m* that utilizes the consecutive symbol dependencies to compress Markov-1 binary source.

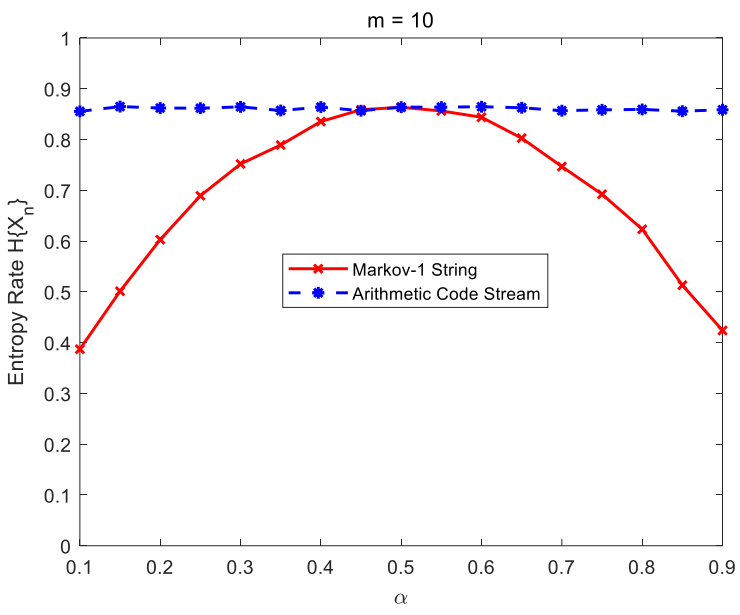
(b) *Q4_b.m* is for this sub-problem. The plot of estimated entropies is as follows:

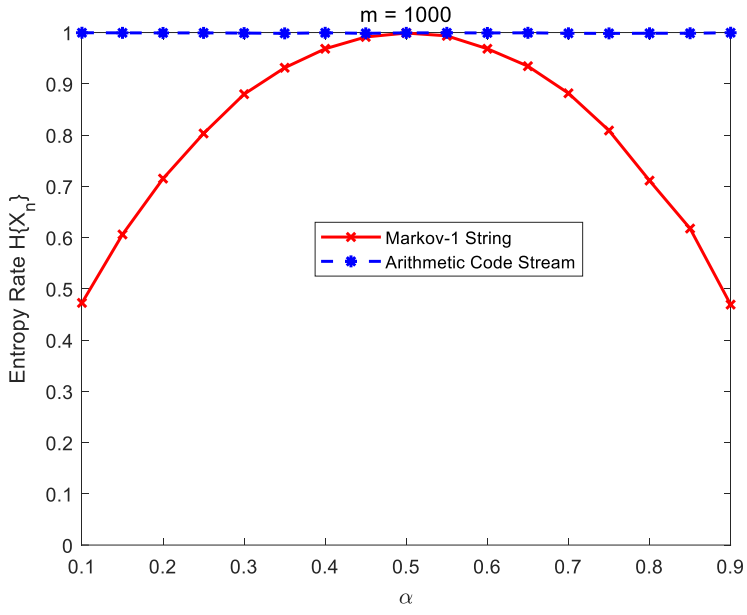


We can expect that for any $0 < \alpha < 1$, we have constant entropy 1 for Markov-1 sequence, and constant entropy 1 for the arithmetic coded sequence. For the Markov-1 sequence, since $\alpha = \beta$ for the Markov-1 binary source, we have equal probabilities for 0 and 1, which leads to constant source entropy 1 for all α, which is agreed by the red curve. For the arithmetic coded sequence, we can conclude that the probability of 1s and 0s are identical.

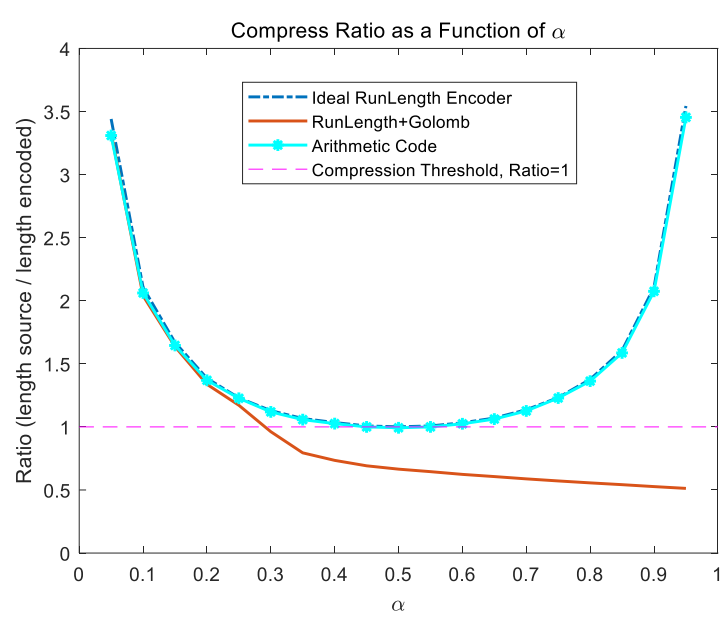
(c) *Q4_c_m_10.m* is the file for this subproblem, which computes the entropy rate of every 10 consecutive symbols and taking the average to estimate the entropy rate for both Markov-1 stream and the code stream. Left one is the corresponding plot. We can see that from the estimation of Markov-1 stream (red curve) the estimation is **not accurate**, since for $\alpha = 0.5$, the true entropy rate should equal to 1. However, the estimation is around 0.85. This is because the short length of consecutive blocks can poorly reflect the statistical properties of Markov source. **Hence, I further obtain a plot (from *Q4_c_m_1000.m*) for a larger m, m=1000, corresponding to the right-sided plot, where the estimation should be more accurate.** The entropy rate of the code stream approaches 1 as α increases. We have discussed the property of the curve for Markov-1 string in the previous assignment. Now focus on the blue curve for arithmetic code stream, we can expect the entropy rate to be constantly 1, which implies that there are no dependencies between every two consecutive symbols. Together with the result in (b), we can conclude that the arithmetic code stream is "distributed" with equal likelihood of value 0 and 1, with no

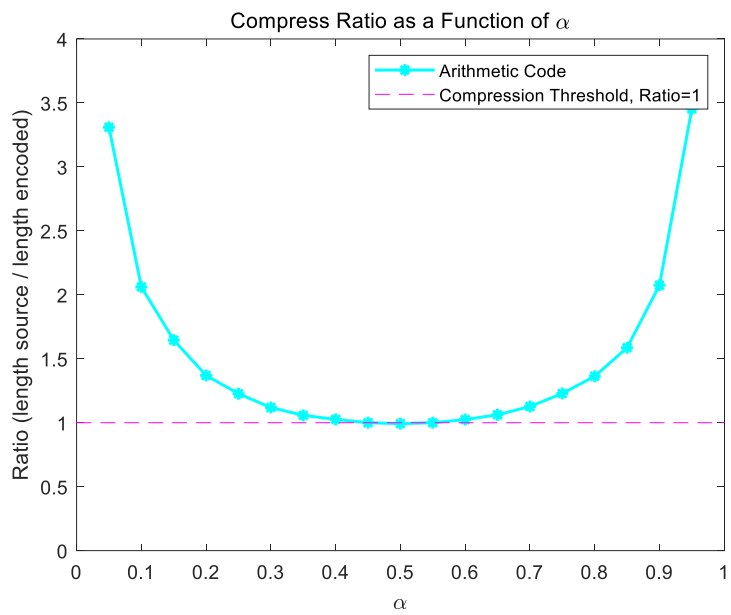
dependencies between consecutive symbols (memoryless sequence, or Bernoulli process).





(d) Please refer to *Q4_d.m*. The left is a plot of compression ratios of the same binary Markov-1 string arithmetic coding, run-length + Golomb coding, and the ideal run-length encoder from Q2 **(for consistency with previous compression ratio plots, the range of α is set to 0.05:0.05:0.95, rather than the requirements of 0.1:0.05:0.9)**. The right plot is a clearer demonstration of compression ratios of arithmetic coding.



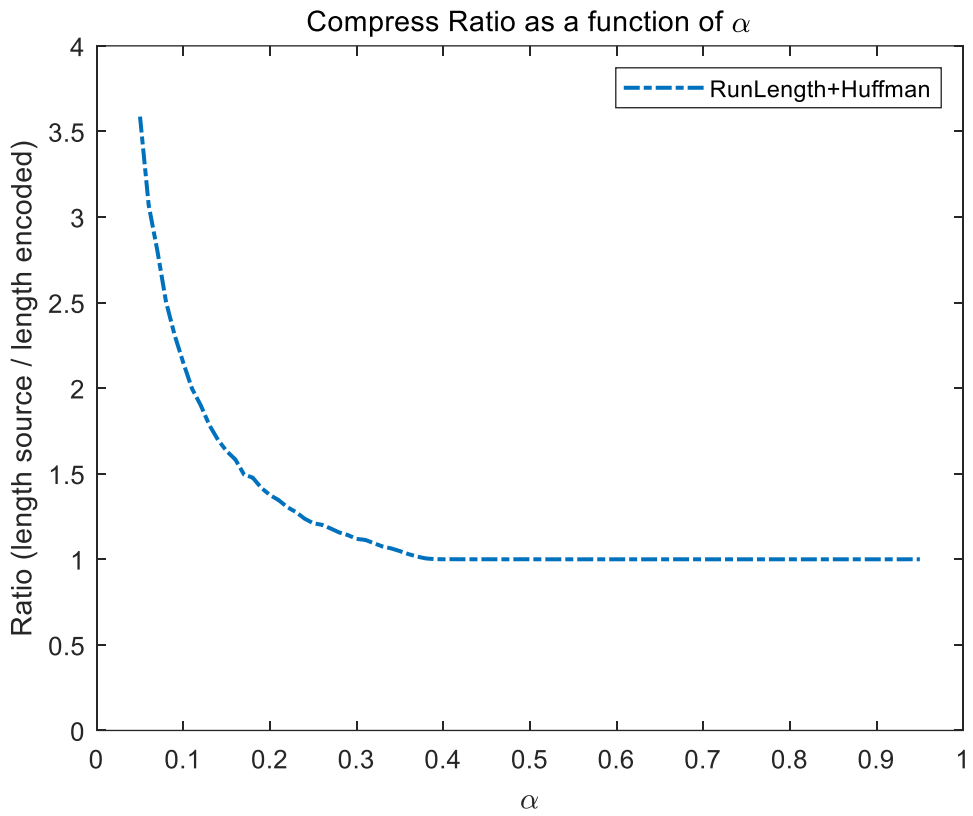


For arithmetic code, the efficiency is slightly lower than the ideal run-length encoder using the theoretical optimal code for all α. We also observe that the compression ratios of arithmetic code are **mostly** above one, which means we can expect at least not a plenty of extra overhead when using it to compress Markov source. But for run-length & Golomb, for medium and large α, we actually are not compressing the source, but using unnecessarily extra information to represent the source, which is very inefficient.

Hence, for binary Markov-1 source, if transition probabilities $\alpha = \beta$ is small, using run-length & Golomb coding scheme to compress the source sequence is an option. Arithmetic coding seems to be an optimal choice for all transition probabilities.

Extra, Run-Length + Huffman

I also test the performance of run-length+Huffman code on binary Markov-1 source, which is pretty good. The unique decidability can also be verified. The scripts are within *Huffman_RunLength/* folder. Here is the plot of efficiency.



For small α, the trend is similar to the Golomb case. But for medium and large α, the compression ratios of run-length & Huffman coding will not go below 1, which symbolics that we are still conducting some sort of "compression".