**Problem 1**

We can define with the standard exponential distribution using what we learned above.

Since for , we have cumulative distribution function

.

Thus, we have the quantile function and therefore,

Because is a monotonic function,

As proved above, defined this way will have the standard exponential distribution.

**Problem 2**

Similar to Problem 1, we have cumulative distribution function

.

Then, we have the quantile function and therefore,

By definition of exponential distribution, because is the rate parameter and is the mean time between events and thus must be positive. Therefore,

As proved above, defined this way will have the general exponential distribution.

**Problem 3**

1. Gaussian quadrature rule is used to evaluate (approximate) integrals of product of a function and a weight function over an interval using . So totally N points () are selected throughout the interval as samples, and the sum of product of their values and corresponding weights would be our approximation of the actual integral. However, for polynomial functions with highest degree less than , the results we get from sum of product is exactly the same as the real value of their integrals. Why this is true will be proved in part (ii) below.
2. Firstly, we could find an orthogonal set of polynomials , where n indexes degree of the polynomial (so is constant and is linear), given a specified normalization convention such as Laguerre and an inner product where . We can construct an n-point quadrature rule exact for polynomials of degree or less by making it exact for the first n elements in the orthogonal set , which means to find a set of n weights satisfying equations:

Then, we can exactly represent any polynomial of degree or less as a linear combination of these elements due to their orthogonality, and therefore our quadrature rule is exact for all those polynomials.

Furthermore, we could write as . If polynomial is or less, because is n-degree, both and are of degree or less.

Thus, we could write and more importantly:

where the first term equals 0 because for .

Therefore, integral of , a polynomial of degree or less, can always be reduced to , a polynomial of degree , for which we can always find a quadrature rule that is exact.

Conclusively, “an n-point Gaussian quadrature rule is exact for polynomials of degree or less.”

**Problem 4**

When we call runif in R, the program uses a procedure called pseudo-random number generation to simulate the effect of True Random Number Generators (TRNG). Although truly randomness is desired and can be generated using hardware random number generators, Pseudo Random Number Generators (PRNG) are more practical due to their computational efficiency and reproducibility.

The input for a PRNG is called a seed, which defined the initial state. Then, the generator would use some algorithm to produce a sequence of values initialized with that seed state. This sequence is treated as values generated randomly, although they are actually determined by a small set of initial values.

Take the classic Linear Congruential Generator (LCG) as an example. This algorithm computes a sequence of pseudo-random numbers with a seed and a discontinuous piecewise linear equation. More specifically, it generates the next value , with modulus , multiplier , increment , and seed . Obviously, the values generated would be within a certain range due to the modulus, and so it is possible to scale the numbers to any other range for the targeted distribution. At the same time, each PRNG has a period, which is the maximum length of repetition-free prefix of the sequence.

As LCG’s quality is regarded as inadequate, advanced techniques such as Linear Feedback Shift Registers (LFSR) have been introduced, and more complex PRNGs were invented. In R, the default PRNG is Mersenne Twister, which has a period of that is proved to be equidistributed (uniformly distributed) in up to 623 dimensions (32-bit values). Users can specify to use other RNGs in R.

The advantage of such PRNG is that with the same seed, we could always reproduce the same sequence of “randomly” generated values. In R, we could do this by calling set.seed(). Additionally, for the purpose of security, it is hard to determine the pseudo-random sequence if the period is long and an attacker only knows the algorithm but does not have the seed.

Coming back to function runif(n, min = 0, max = 1), we could specify n, the number of observations or length of sequence to be generated, and the range of distribution, which by default is [0, 1]. This function would then return a sequence of values generated using a PRNG that greatly simulates a real uniform distribution and scales to the desired range.