Computer-Intensive in Stats - Project 01

Albert Sidney Hancock III

February 2024

Assignment Statement

One simple test to determine whether a random number generator is giving random values from a distribution with distribution function $F(x)$ is that of Kolmogorov and Smirnov.

Using the random number generator, obtain a sample of size n.

Order the sample with x_r being the rth smallest observation.

Define the sample distribution function S_n to be

$$S_n(x) = \begin{cases} 0 \text{ for } x < x_{(1)} \\ \frac{r}{n} \text{ for } x_{(r)} \leq x < x_{(r+1)} \\ 1 \text{ for } x_{(n)} \leq x \end{cases}$$

Then, for n larger than 80, it can be shown that if the x's are truly coming from F, then, with probability 0.99,

$$\sup_x |S_n(x) - F(x)| < \frac{1.6276}{\sqrt{n}}.$$

Use the random number generator, $X_{n+1} = 7^5 X_n \text{mod}(2^{31}-1)$, to generate a sample of size $1,000$ from the uniform distribution on the unit interval.

Does your sample pass the Kolmogorov-Smirnov test?

Minimal Requirements

Generate Data

For this step, I defined the function for the random number generator as provided in the assignment description.

For generating the values, I started with a seed value of 1, since no seed value was provided. After generating the $1,000$ values, I dropped the seed value from the list, since it seemed pretty far off from the remaining values.

For each of the values generated, I scaled it into the unit interval by dividing by the modulus, $2^{31} - 1$.

I also used the numpy generator to generate $1,000$ random samples from $U(0,1)$ to use as our differencing values. That is, these are assumed to properly be drawn from $F(x)$.

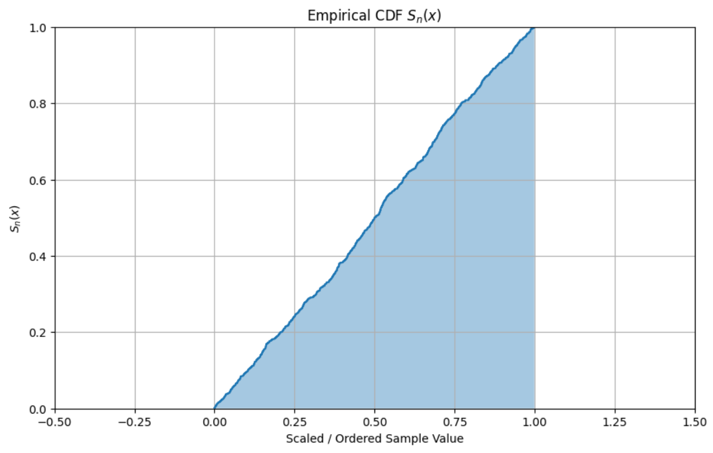
The scaled RNG values and uniform values are sorted, and then the $S_n(x)$ values are generated. Finally, for each index I stored the absolute values of $S_n(x) - F(x)$.

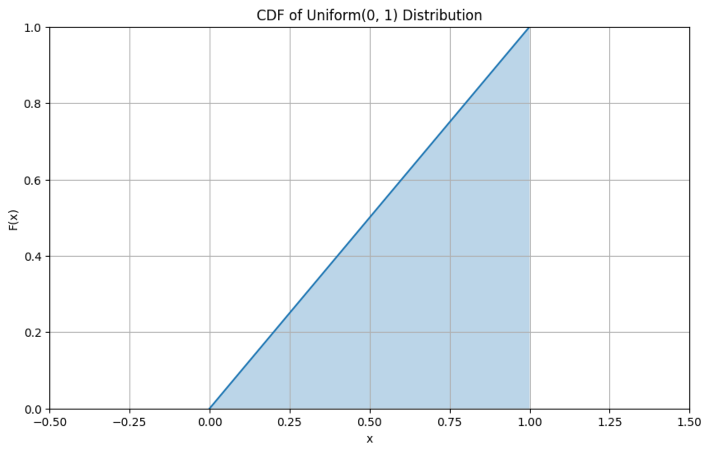
The data points were all saved to a CSV file at this point.

Graphical Validation

Before implementing the K-S test, I wanted to check the shape of my $S_n(x)$ values (empirical CDF) against a generated graph of the $U(0,1)$ CDF.

The shape of the empirical CDF has roughly the correct shape by observation. That is, it looks like a right triangle with the right angle on the bottom right of the graph, and whose hypotenuse is drawn from $(0,0)$ to $(1,1)$.





K-S Test

To find if $\sup_x |S_n(x) - F(x)| < \dfrac{1.6276}{\sqrt{n}}$, I calculated $\dfrac{1.6276}{\sqrt{n}}$ directly in python. This value returned as 0.05146923119690054.

For $|S_n(x) - F(x)|$, I found the maximum value of the 'differences' column computed at the end of the 'Generate Data' step. This value returned as 0.04322838604234547.

I ended up with the following test result: $0.04322838604234547 < 0.05146923119690054$, which is true.

Therefore, at least in this iteration, the Kolmogorov-Smirnov test indicates that there is no statistically significant difference between the two distributions.

Additional Requirements

To check if the results *truly* simulate a uniform distribution of $U(0,1)$, I decided to test with a few methods going from least to most empirical.

First, I'll check if the histograms of the simulation values look similar to those of the true distribution.

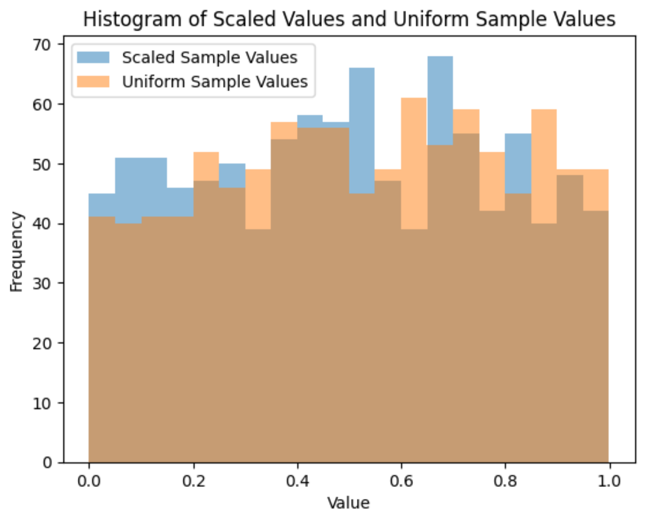
Second, I'll look at the mean and variance of the simulation values against the known analytic values.

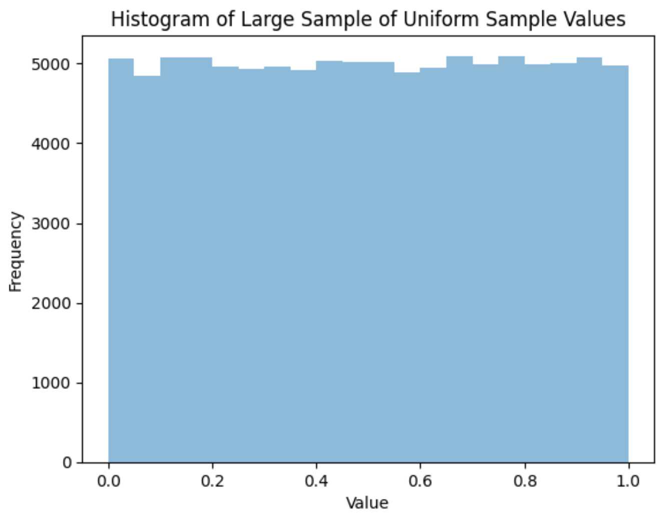
Finally, I'll use one of the integration simulation problems we did earlier in the course, and compare the result when feeding in our RNG values against samples drawn with the vetted numpy uniform generator.

Graphical Verification

To compare the histograms, I just plot one on top of the other. I also separately plotted a separate set of samples from $U(0,1)$ but with $100,000$ samples.

The number of bins for these were all 20.





While I don't get a fairly rectangular shape when looking at the scaled RNG samples, which I might expect given a large random sample size from $U(0,1)$ (the right-hand graph), I do get a shape that looks fairly similar to one generated by only $1,000$ samples from $U(0,1)$.

At this point, they still seem roughly the same.

Mean and Variance Comparison

When the statistics between the scaled RNG and $U(0,1)$ samples, I generated the following results:

- Mean of the scaled sample values: ≈ 0.498
- Variance of the scaled sample values: ≈ 0.079
- The absolute difference between the sample mean and analytical mean: ≈ 0.002
- This is difference of ≈ 0.004 percent of the actual value.
- The absolute difference between the sample variance and analytical variance: ≈ 0.005
- This is difference of ≈ 0.055 percent of the actual value.

So, the mean and variance of the sample are *very* close to the analytic values.

Use in a Monte-Carlo Simulation with Known Result

Finally, I thought a good way to test if our samples are from our uniform distribution would be a kind of 'unit test', where we use our values in an algorithm that already has a known result.

In an earlier exercise for this course, we had the following objective:

Use simulation to approximate the integral $\int\limits_0^\infty x(1+x^2)^{-2}dx$.

Our algorithm for this process is as follows:

1. Generate $y \sim U(0,1)$
2. Evaluate $h(y) = (\frac{1}{y}-1)(1+(\frac{1}{y}-1)^2)^{-2}(\frac{1}{y^2})$
3. Repeat steps 1 and 2 a total of n times, for n large.
4. Find $\dfrac{\sum h(y_i)}{n} \dot\sim \int_0^\infty x(1+x^2)^{-2}dx$

Here, our 'n large' would be the same as the number of samples / iterations of our random number generators. That is, $1,000$.

I ran the above algorithm with $n = 1,000$ for $10,000$ trials. By the CLT, we can use those trial results to generate the theoretical average and standard deviation values for samples drawn of $n = 1,000$.

We can then run the scaled samples of our random number generator through the algorithm and find that value.

We will then use a Z test, at $\alpha = 0.05$, to test if our sample deviates from that of the theoretical distribution generated by the repeated trials.

I'll note here that $\alpha = 0.05$ is being used as a baseline, but is somewhat arbitrary. If a high-risk or critically important process depended on the sample values, I would use a smaller α to guarantee a higher level of accuracy for the number generator.

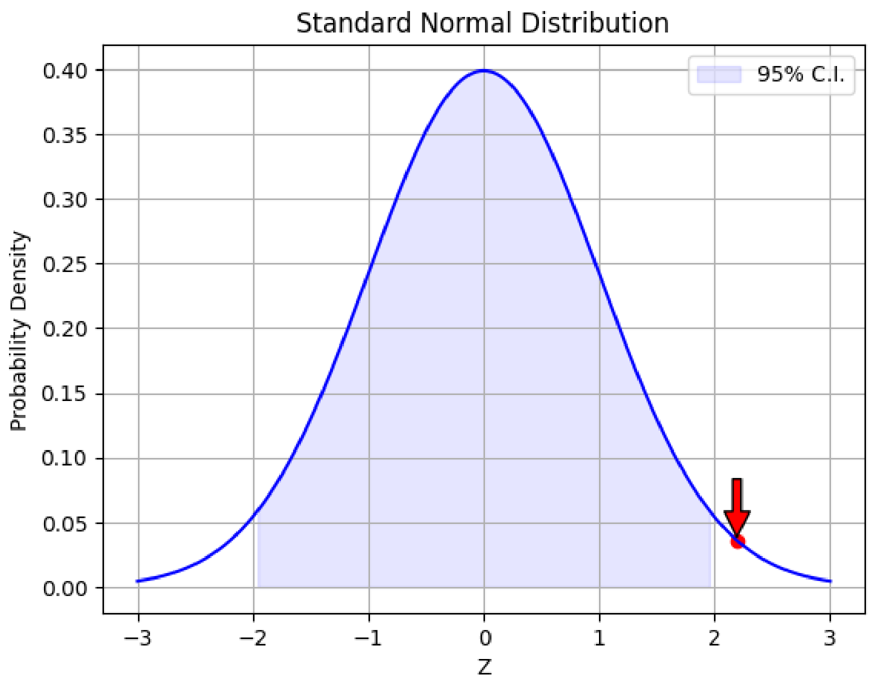
I got the following results:

- Minimum: 0.4614629348629908
- First quartile (Q1): 0.4927920001383537
- Second quartile (Median, Q2): 0.5002363238959706
- Third quartile (Q3): 0.5074907874105098
- Maximum: 0.5351464497495539
- The 95th Percentile: 0.5173483131984175
- The Sample Value: 0.523517872740267

Scaling to the Standard Normal Distribution, we get a Z-score of ≈ 2.194.

For the $\alpha = 0.05$, this is actually outside of the 95^{th} percentile range, which indicates that we would reject the null hypothesis.

That is, the integration value given by using our scaled RNG values is statistically significantly different from the general population of values generated using what we have assumed to be 'true' $U(0, 1)$ sample sizes of $n = 1,000$.



If I have done this test right, this indicates that the pseudo-random values generated by our random number generator are **not** valid, at least in this context.

Conclusion

Given that I can only generate 1,000 samples at a time from this random number generator, I will make the following observations:

1. The sample passes the Kolmogorov-Smirnov test.
2. The CDF of the scaled samples appears to be roughly the same as the CDF of $U(0,1)$.
3. The histograms look roughly similar for the RNG sample and the $U(0,1)$ samples.
4. The expected value and variance values appear to be close to the theoretical values.

For these reasons, it seems likely that we can consider the random numbers (scaled to the unit interval) roughly uniformly distributed.

However, when used in application, we got an integral value that was outside of the 95th percentile range of the implied population distribution.

For 'further steps' before trusting this RNG, I would want to verify with another statistician that I haven't made any assumptions or errors in my unit test (the integration procedure).

Assuming I haven't made any errors, I actually don't think that I would trust this RNG over the numpy implemented generator in any industrial context, unless there was some compelling reason to do so. (Though, I might do another round of unit tests within the context of that problem.)

Further investigations would be to use larger sample sizes and see if the integration procedure output moves toward the center of the results distribution.

I may also try using various seed sizes, though I tried a few and didn't get results that were significantly different.

Appendix - Code

The code for this project was done in a Jupyter notebook using Python, so the snippets are all saved separately.

I've tried to label the sections to correspond with the steps taken to make it easier to reference, though these are all in the same order as the steps taken throughout the project.

Generate Data

```python
# Declare the values for the random number generator.
increment = 0
multiplier = 7**5
modulus = (2**31)-1
number_of_iterations = 1000

# The seed value is not explicitly provided, so I'll start with 1.
seed_value = 1
```

```python
# We create our starting lists
values = [seed_value]

function_values = []
scaled_ordered_values = []
uniform_values = []
step_function_levels = []

# I want to create an index list for the dataframe.
index_list = np.arange(1, number_of_iterations + 1)
```

```python
# Generate the random number generated values.
for _ in range(number_of_iterations):

    # Calculate the next value.
    next_value = (multiplier * values[-1] + increment) % modulus

    # Store the value to the list.
    values.append(next_value)

# We now have 1001 values, including our seed value.
# We drop the seed value, which is currently the first one in the list, leaving us with the 1,000 we
    wanted.
values.pop(0)

# We put the values in order
ordered_values = sorted(values)

# To scale the values to be on the unit inverval, I will divide through by the modulus.
for number in ordered_values:
    scaled_value = number/modulus
    scaled_ordered_values.append(scaled_value)

# We generate values from the theoretical distribution.
uniform_values = np.random.uniform(0,1,number_of_iterations)

# Sort the uniform values.
sorted_uniform_values = sorted(uniform_values)

# Create the S values.
for i in range(number_of_iterations):
    step_value = (i+1) / number_of_iterations
    step_function_levels.append(step_value)

# We'll need to find the absolute value of the differences for the S and F so that we can find the
    supremum / max of those.
differences = [abs(s - f) for s, f in zip(scaled_ordered_values, sorted_uniform_values)]
```

```python
# Create a DataFrame with the ordered values and their corresponding S_n(x) values
df = pd.DataFrame({
    'Index': index_list,
    'X': scaled_ordered_values,
    'S': step_function_levels,
    'F': sorted_uniform_values,
    'Difference': differences
})

# Save the results to a csv file.
df.to_csv('../data/RNG Results')

df
```

CDF Plots

```python
# Plotting the step function for S
plt.figure(figsize=(10, 6))

# We adjust the plot limits so that we make sure we can see the whole shape.
plt.xlim(-0.5, 1.5)
plt.ylim(0, 1)

# We plot the step function with the space under S filled in.
plt.step(df['X'], df['S'], where='post', label='Empirical CDF $S_n(x)$')
plt.fill_between(df['X'], df['S'], step="post", alpha=0.4)

# We label the axes and the plot itself.
plt.xlabel('Scaled / Ordered Sample Value')
plt.ylabel('$S_n(x)$')
plt.title('Empirical CDF $S_n(x)$')
plt.grid(True)

# Display the plot
plt.show()
```

```python
# Plotting the step function for S
plt.figure(figsize=(10, 6))

# We adjust the plot limits so that we make sure we can see the whole shape.
plt.xlim(-0.5, 1.5)
plt.ylim(0, 1)

# Generate our x values.
x = np.linspace(0, 1, 1000)

# Since the CDF of a uniform distribution is F(x) = x for 0 <= x <= 1
cdf = x

# Plot the function and fill beneath.
plt.plot(x, cdf, label='CDF for Uniform(0, 1)')
plt.fill_between(x, 0, cdf, alpha=0.3)

# We label the axes and the plot itself.
plt.xlabel('x')
plt.ylabel('F(x)')
plt.title('CDF of Uniform(0, 1) Distribution')
plt.grid(True)

plt.show()
```

Test Results

```python
supremum = max(df['Difference'])
check_value = (1.6276)/(number_of_iterations**(1/2))

print(check_value)
print(supremum)

print(supremum < check_value)
```

Histograms

```python
# Generate histograms
plt.hist(df['X'], bins=20, alpha=0.5, label='Scaled Sample Values')
plt.hist(df['F'], bins=20, alpha=0.5, label='Uniform Sample Values')

# Adding labels and title
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Scaled Values and Uniform Sample Values')
plt.legend()

# Display the plot
plt.show()
```

```python
# Generate a separate, larger sample of U(0,1) values.
separate_uniform_values = np.random.uniform(0, 1, 100000)

# Generate histograms
plt.hist(separate_uniform_values, bins=20, alpha=0.5, label='Uniform Sample Values')

# Adding labels and title
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Large Sample of Uniform Sample Values')

# Display the plot
plt.show()
```

Expected Value and Variance

```python
# The known analytic values for mean and variance of U(0,1)
known_uniform_mean = 0.5
known_uniform_variance = 1/12

# Sample mean and variance
sample_mean_value = np.mean(df['X'])
sample_variance_value = np.var(df['X'])

# Sample statistic results.
print(f"Mean of the scaled sample values: {mean_value}")
print(f"Variance of the scaled sample values: {variance_value}")

# Differences between the sample statistics and analytic values.
print(f"The absolute difference between the sample mean and analytical mean: {abs(known_uniform_mean -
    sample_mean_value)}")
print(f"This is difference of {(abs(known_uniform_mean - sample_mean_value))/known_uniform_mean} percent
    of the actual value.")

print(f"The absolute difference between the sample variance and analytical variance: {abs(
    known_uniform_variance - sample_variance_value)}")
print(f"This is difference of {(abs(known_uniform_variance - sample_variance_value))/
    known_uniform_variance} percent of the actual value.")
```

Using Values in a MC Simulation with Known Result

```python
def monte_carlo_integration(function, number_of_samples):
    # Draw the prescribed number of random samples from a uniform distribution U(0,1).
    samples = np.random.uniform(0,1,number_of_samples)

    # Evaluate the function for each of the drawn values.
    function_values = function(samples)

    # Take the average of the function values.
    integral_estimate = np.sum(function_values)/number_of_samples

    return integral_estimate
```

```python
# This is our transformed integral - it needs to have bounds of 0 and 1.
def integral(y):
    return ((1/y - 1)*(1 + (1/y - 1)**2)**(-2))*(1/y**2)
```

```python
def run_trials(number_of_rounds):
    for _ in range(number_of_rounds):
        trial_result = monte_carlo_integration_result = monte_carlo_integration(integral,
    number_of_samples)
        trial_results.append(trial_result)
```

```python
def monte_carlo_from_sample(original_sample_results, function):
    # Draw the prescribed number of random samples from a uniform distribution U(0,1).
    samples = original_sample_results

    # Evaluate the function for each of the drawn values.
    function_values = function(samples)

    # Take the average of the function values.
    integral_estimate = np.sum(function_values)/len(samples)

    return integral_estimate
```

```python
number_of_samples = 1000
number_of_rounds = 10000

trial_results = []
```

```python
run_trials(number_of_rounds)
sample_generated_result = monte_carlo_from_sample(df['X'], integral)
```

```python
trial_results_mean = np.mean(trial_results)
trial_results_standard_deviation = np.sqrt(np.var(trial_results))
```

```python
# Calculate Quartile Values
Min = min(trial_results)
Q1 = np.percentile(trial_results, 25)
Q2 = np.percentile(trial_results, 50)
Q3 = np.percentile(trial_results, 75)
Max = max(trial_results)

# Print quartile values
print(f"Minimum: {Min}")
print(f"First quartile (Q1): {Q1}")
print(f"Second quartile (Median, Q2): {Q2}")
print(f"Third quartile (Q3): {Q3}")
print(f"Maximum: {Max}")

print("\n")

# 95th Percentile and Sample Generated Result value
ninety_fifth_percentile_value = np.percentile(trial_results, 95)
print(f"The 95th Percentile: {ninety_fifth_percentile_value}")
print(f"The Sample Value: {sample_generated_result}")
```

```python
# Calculate the Z score

# Population parameters
population_mean = trial_results_mean
population_standard_deviation = trial_results_standard_deviation

# Calculate the Z-score
Z = (sample_generated_result - population_mean) / population_standard_deviation

print(f"Z-score: {Z}")
```

```python
# We will generate a plot of a standard normal distribution.
mean = 0
std_dev = 1
x = np.linspace(-3, 3, 1000)
y = norm.pdf(x, mean, std_dev)
plt.plot(x, y, color='blue')

# Z-score to highlight on the plot
Z_value = Z
y_value = norm.pdf(Z_value, mean, std_dev)

# Mark the Z-value on the plot and fill in the 95% confidence interval.
plt.scatter(Z_value, y_value, color='red')
plt.fill_between(x, y, where=(x >= -1.96) & (x <= 1.96), color='blue', alpha=0.1, label='95% C.I.')

# Optionally, you can use an arrow to annotate the Z-value
plt.annotate('',xy=(Z_value, y_value), xytext=(Z_value, y_value + 0.05),
             arrowprops=dict(facecolor='red', shrink=0.05), color='red')

# Plot values.
plt.title('Standard Normal Distribution')
plt.xlabel('Z')
plt.ylabel('Probability Density')
plt.legend(loc='upper right')
plt.grid(True)

plt.show()
```

```python
# Create a DataFrame with the ordered values and their corresponding S_n(x) values
df2 = pd.DataFrame({
    'Simulation Results': trial_results,
})

# Save the results to a csv file.
df2.to_csv('../data/Simulation Results')

df2
```

Appendix - Data

I have saved the CSV files of the results both for the Monte Carlo integration trials and for the random number generator, empirical CDF, and generated Uniform variables.

These too long to be put at the end of this document.

These can be provided upon request, or you can find them here at the following link:

https://github.com/Goodfeelings/spring-2024-comp-intensive-stats-projects-01/tree/main/data