Andreu Boix Torres
andreu.boix.torres@student.uva.nl

Stochastic Simulation
First assignment

2024-10-05

NOTE: We specify a seed in MATLAB/C++ in other to make all the experiments replicable. The seed is 1234. The procedure that precedes every function or code provided (in MATLAB, for instance) is:

```
seed = 1234;
rng(seed);
```

For convenience, the code displays will not always mention setting the seed, but all results and graphics presented are based on initializing the seed to 1234 before starting each code or function. The seed is only set when appropriate, i.e. it is not fixed during loops that run multiple simulations, otherwise we would get the same result repeated for each iteration.

Exercise 1. Please use your programming language or mathematical software package to perform the following steps for n large enough:

Draw n samples from a two-dimensional random variable $U = (U_1, U_2)$, where U_1 and U_2 are mutually independent, U_1 is uniformly distributed on $[-1, 1]$ and U_2 is uniformly distributed on $[0, 1]$. Call your samples $U^{(1)}, U^{(2)}, \ldots$, where $U^{(i)} = (U_1^{(i)}, U_2^{(i)})$

We will provide a unified code and explanation for both sections of the exercise.

Numerically compute the following formula:

$$\frac{4}{n}\sum_{i=1}^{n} \mathbb{1}_{\{(U_1^{(i)})^2+(U_2^{(i)})^2\leq 1\}}$$

When we compute $U_1 \sim U(0, 1)$ and $U_2 \sim U(-1, 1)$, and join it into $U = [U_1, U_2]$, we can get the following interpretation. Sampling from U we are "throwing darts" to a rectangular area of $[-1, 1] \times [0, 1]$. Also, we know that $x^2 + y^2 \leq 1$ is the equation of the area of the unit circle centered at $(0, 0)$, and is restricted to the points that come from the rectangle (see 1). Hence, as we take x as U_1 and y as U_2 in the result that we have to compute ($\frac{4}{n}\sum_{i=1}^{n} \mathbb{1}_{\{(U_1^{(i)})^2+(U_2^{(i)})^2\leq 1\}}$), we are modelling the proportion of the points of the rectangle (n points on the plane ($\mathbb{R}^2$) of the form $((U_1^{(i)})^2, (U_2^{(i)})^2)$) that fall into the semicircle (characteristic function of the points that comply the equation of the circle applied to the n points $((U_1^{(i)})^2, (U_2^{(i)})^2)$) multiplied by 4, that is:

$$4 \cdot \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\{(U_1^{(i)})^2+(U_2^{(i)})^2\leq 1\}} = 4 \cdot \frac{\text{Points of the rectangle that fall into the semicircle}}{\text{Points of the rectangle}} =$$

$$4 \cdot \frac{\text{Area of the semicircle inside the rectangle}}{\text{Area of the rectangle}} = 4 \cdot \frac{(\pi R^2)/2}{(2R) \cdot R} = 4 \cdot \frac{\pi R^2}{4R^2} = 4 \cdot \frac{\pi}{4} = \pi,$$

where we have used that the area of the circle that falls in the rectangle is $\pi R^2/2$ with $R = 1$ and the area of the rectangle is $2R \cdot R = 2R^2 = 2$.

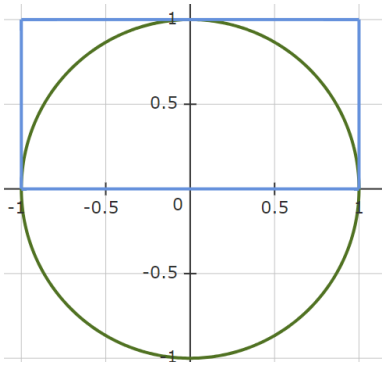


Figure 1: Representation of the rectangle area (blue) and the unit circle (green)

Now let us formalize the simulation for the model explained above using MATLAB Code:

```
function [n, result] = Result_ExerciseOne(n)
    % We define U1 as the n-vector for which, at every position,
    % there is a sample of a uniform distribution on [-1, 1].
    U1 = -1 + 2 * rand(n, 1);

    % We define U2 as the n-vector for which, at every position,
    % there is a sample of a uniform distribution on [0, 1].
    U2 = rand(n, 1);

    U_samples = [U1, U2];

    sum_value = 0;

    % Loop over each sample and check if it lies within the unit
    % circle
    for i = 1:n
        U1_i = U_samples(i, 1);
        U2_i = U_samples(i, 2);

        if (U1_i^2 + U2_i^2) <= 1
            sum_value = sum_value + 1;
        end
    end

    % Compute the final result
    result = (4 / n) * sum_value;

    disp('Computed result:');
    disp(result);
end
```

MATLAB Results and interpretation:

```
Result_ExerciseOne(1e4)
Computed result:
    3.1492
Result_ExerciseOne(1e5)
Computed result:
    3.1449
Result_ExerciseOne(1e6)
Computed result:
    3.1406
Result_ExerciseOne(2e6)
Computed result:
    3.1414
Result_ExerciseOne(3e6)
```

```
Computed result:
    3.1415
```

We can see that, as n approaches ∞, we can derive that the computed result approaches π.

That's why we use the following code to test its convergence, with a tolerance of $1e-6$, starting from $n = 1e6$ samples. Since the code runs quickly, there is no need to set a larger n-step, so we fix it to n-step $= 1$.

```
function Convergence_ExerciseOne
    n = 1e6;
    tol = 1e-6;
    p = pi;
    [n, result] = Result_ExerciseOne(n);
    while (abs(result - p) > tol)
        n = n + 1;
        [n, result] = Result_ExerciseOne(n);
        %disp([n, result]);
    end
    X = sprintf("The number of samples is %0.2e and the result to
       which it approaches is %0.5d", n, result);
    disp(X);
end
```

The output of this function is:

```
Convergence_ExerciseOne
The number of samples is 1.00010e+06 and the result to which it
   approaches is 3.1415930e+00
```

As a result, we have justified (also numerically) that, as n grows to infinity, the computed result approaches $3.1416 \approx \pi$.

In summary, after presenting the MATLAB code that computes the result and generates the n samples of U, let us now introduce the preview of a JavaScript code (see Appendix A for the code). This visualization simulates dart throwing, where green points represent those that land inside the semicircle, and red points represent those that fall outside the semicircle but still within the rectangle.

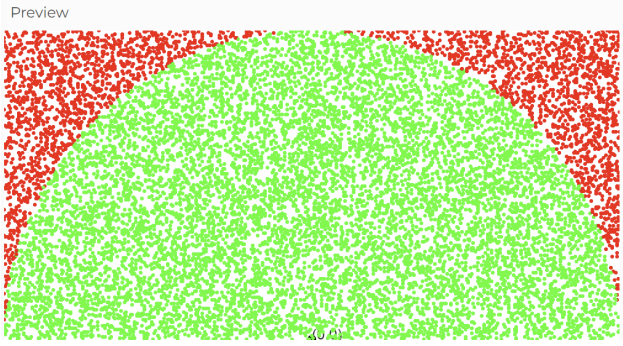


Figure 2: Representation of the dart throwing

Exercise 2 (2.1 [AG]). Write a routine for generation of a Weibull r.v. X with tail $\mathbb{P}(X > x) = e^{-x^\beta}$, $x > 0$, by inversion. Check the routine via a histogram of simulated values plotted against the theoretical density, say for $\beta = 1/2$.

The tail distribution function $\mathbb{P}(X > x) = e^{-x^\beta}$ is related to the cumulative distribution function (CDF) $F(x)$ in the following way:

$$F(x) = 1 - P(X > x) = P(X \leq x) = 1 - e^{-x^\beta}.$$

To generate X, we will use the inversion method first by generating a uniform random variable U in $[0, 1]$ and then set the CDF equal to U:

$$U = F(x) = 1 - e^{-x^\beta}$$

Solving for x:

$$U = 1 - e^{-x^\beta} \iff -x^\beta = log(1-U) \iff x^\beta = -log(1-U) \iff x = (-log(1-U))^{1/\beta} \implies x = (-log(U))^{1/\beta}.$$

Notice that we have used that $1 - U \sim U$, since it's a Uniform random variable in $[0, 1]$.

Therefore, to generate X as a Weibull random variable, we will need to draw a uniform random variable $U \sim U(0, 1)$ and then use the formula that we have just found:

$$X = (-log(U))^{1/\beta}$$

```
% Weibull random variables using inversion method
function X = weibull_rv(n, beta)
    U = rand(n, 1);
    X = (-log(U)).^(1/beta);  % Apply inversion formula
end

% Theoretical Weibull PDF for comparison
function pdf_vals = weibull_pdf(x, beta)
    pdf_vals = beta * x.^(beta - 1) .* exp(-x.^beta);  % Weibull PDF
end

beta = 1/2;       % Shape parameter (beta)
n = 1e7;
bins = 3e2;       % Number of bins for the histogram

X = weibull_rv(n, beta);

figure;
% The histogram is for the simulated values
histogram(X, bins, 'Normalization', 'pdf', 'FaceAlpha', 0.6, '
   EdgeColor', 'none');
hold on;
% Generate x-axis values for the plot
x_vals = linspace(0, max(X), 1000);
% The line is for the theoretical density
plot(x_vals, weibull_pdf(x_vals, beta), 'r-', 'LineWidth', 2);

title(['Histogram of Weibull Random Variables (\beta = ', num2str(beta
   ), ')']);
xlabel('x');
ylabel('Density');
legend('Simulated', 'Theoretical PDF');
grid on;
hold off;
```

This is the combined plot:

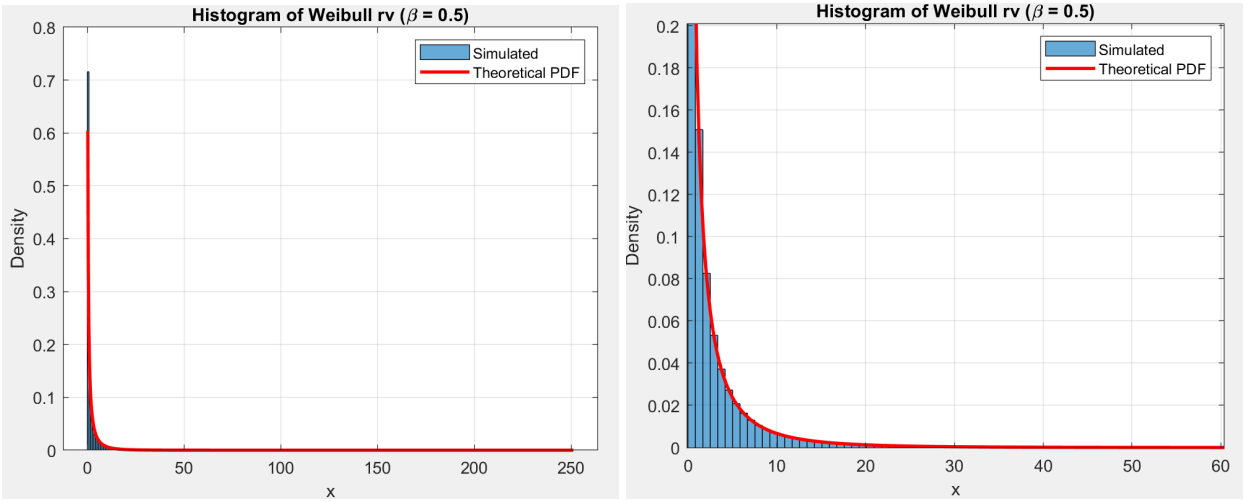


Figure 3: Combined histogram (blue) with line plot (red) and zoom-in

For comparison, we have also defined the theoretical density for the Weibull and, for $n = 1e7$, we observe that our routine is effectively correct. For more clarification, we also plot both graphics separately, to show how each plot is distributed:

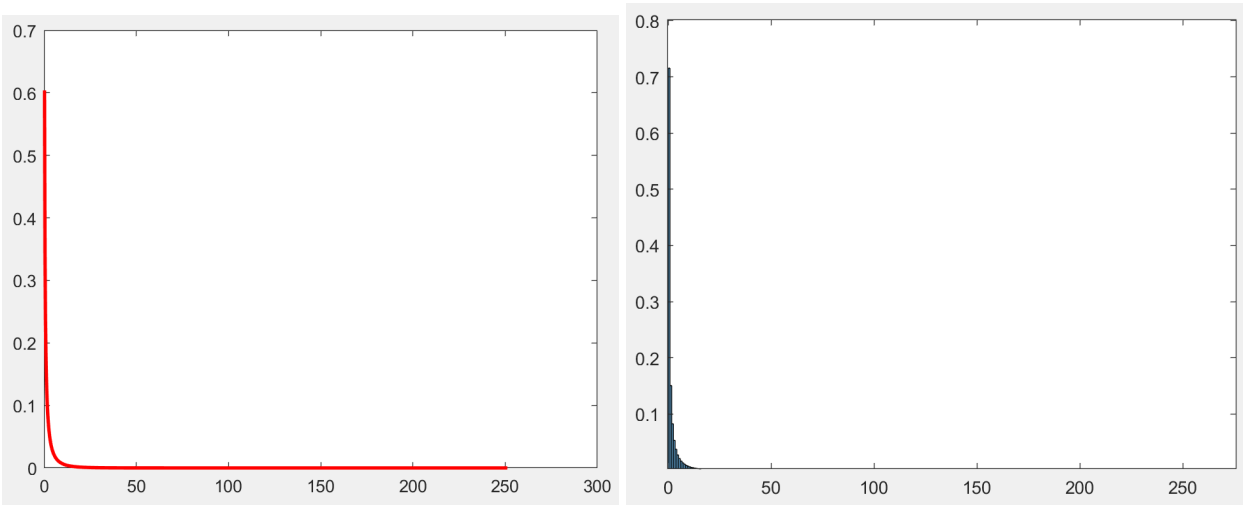


Figure 4: Separate histogram (blue) and line plot (red)

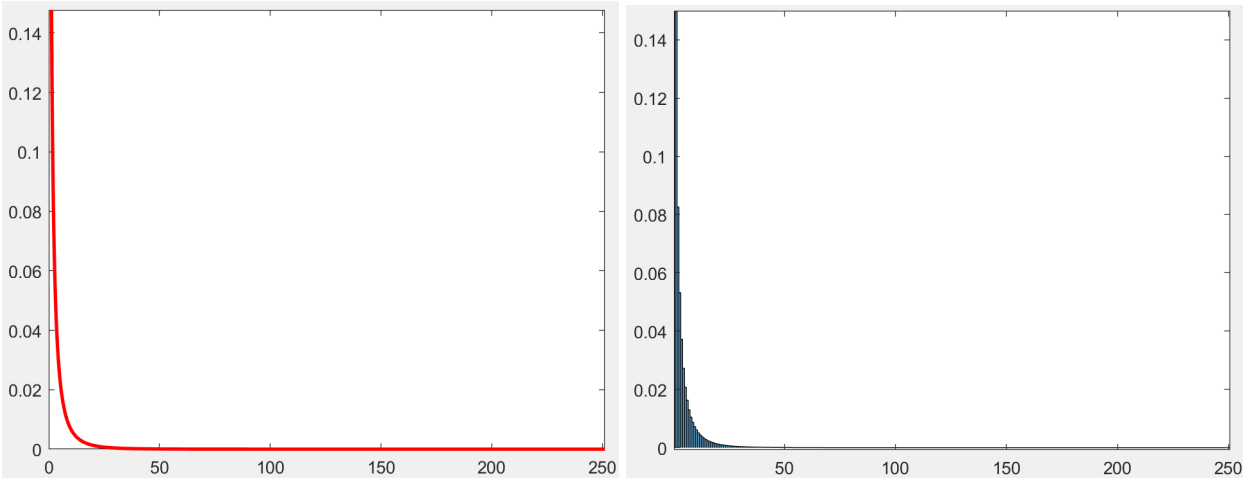


Figure 5: Respective zoom-ins around the values near 0

Exercise 3 (2.8 [AG]). Assume that the density of X can be written as $f(x) = cg(x)\bar{H}(x)$, where $g(x)$ is the density of an r.v. Y and $\bar{H}(x) = \mathbb{P}(Z > x)$ the tail of an r.v. Z. Show that X can be generated by sampling Y, Z independent and rejecting until $Y \leq Z$.

Let us assume that the density of X can be written as $f(x) = cg(x)\bar{H}(x)$. Here, $g(x)$ is the probability density function (PDF) of the random variable Y, and $\bar{H}(x) = \mathbb{P}(Z > x)$ is the tail probability of the random variable Z.

First, let's sample Y from its distribution with PDF $g(y)$ and, independently, we will also sample Z from its distribution.

Since our non-rejection condition is $Y \leq Z$, we will only accept the sample $Y \iff Y \leq Z$. Let us show, for every y, the probability that the pair (Y, Z) is such that $Y \leq Z$ is given by:

$$\mathbb{P}(Y \leq Z | Y = y) = \mathbb{P}(Z \geq y) = \bar{H}(y)$$

Therefore, the joint distribution of Y and Z conditional on $Y \leq Z$ will ensure that the marginal density of the accepted values of Y is proportional to $g(y)\bar{H}(y)$, because the probability density of the accepted values of Y is:

$$\mathbb{P}(Y \text{is accepted}) = g(y)\mathbb{P}(Z \geq Y) = g(y)\bar{H}(y)$$

and this is proportional to the target density $f(x) = cg(x)\bar{H}(x)$. Let us see that:

$$F_{Y,Z|Y \leq Z}(x) = \int_{-\infty}^{x} \bar{c}g(y)P(Z \geq Y)dy = \int_{-\infty}^{\infty} \bar{c}g(y)\bar{H}(y)dy$$

with $\bar{c}$ as the normalization constant.

The normalization constant can also be seen at the PDF of X, with c as the normalization constant, i.e. the constant that ensures that

$$\int_{-\infty}^{\infty} f(t)dt = \int_{-\infty}^{\infty} cg(x)\bar{H}(x)dx = 1$$

The CDF of X is:

$$F_X(x) = \int_{-\infty}^{x} f(t)dt = \int_{-\infty}^{x} cg(x)\bar{H}(x)dx$$

But if c and $\bar{c}$ are two normalization constants, then $c = \bar{c}$ and hence, we are indeed simulating X via Y and Z.

And, to sum up, we now have a procedure that simulates X as follows:

i) Sample $Y \sim g(y)$.

ii) Sample $Z \sim \bar{H}(x)$.

iii) Accept Y if $Y \leq Z$. Else, reject and resample.

Exercise 4 (2.10 [AG]). Write a routine for generation of an inverse Gaussian r.v. X with density (A1.2) by A-R when $\xi = c = 1$. Check the routine via confidence intervals for $\mathbb{E}X, \mathbb{E}X^2$, and (a little harder!) $\mathbb{V}arX$, using the known formulas $\mathbb{E}X = c/\xi$, $\mathbb{V}arX = c/\xi^3$.

The PDF of the Inverse Gaussian distribution $IG(c, \xi)$ is:

$$f(x) = \frac{c}{x^{3/2}\sqrt{2\pi}} \exp\left\{\xi c - \frac{1}{2}\left(\frac{c^2}{x} + \xi^2 x\right)\right\}, \quad x > 0.$$

Given the known formulas for the mean $\mathbb{E}X$ and variance $\mathbb{V}arX$ and our exercise assumption which is that $c = \xi = 1$, we have:

- $\mathbb{E}X = \frac{c}{\xi} = 1$
- $\mathbb{V}arX = \frac{c}{\xi^3} = 1$

We are going now to generate random samples from the $IG(c = 1, \xi = 1)$ and verify the sample statistics through confidence intervals for the mean and the variance.

To propose a routine for the generation of the inverse Gaussian r.v. X, we will first need to suggest a proposal distribution.

Our proposed distribution is the exponential distribution. We now outline some reasons that justify why the exponential distribution works well.

Justification of the proposed distribution for the A-R method

Similarity in tail behaviour. The Inverse Gaussian distribution has a heavy tail (slow decay as x increases). In the exponential distribution, the decay can be controlled by the rate parameter λ, so we can also control the decay, with the PDF:

$$g(x) = \lambda e^{-\lambda x}, x > 0.$$

Positivity of both distributions. Both distributions are defined over positive real numbers, the support of both distributions is $x > 0$. This avoids unnecessary rejections due to negative values.

Mathematical convenience in the ratio of PDFs of the A-R method.

$$\frac{f(x)}{C \cdot g(x)} = \frac{\frac{c}{x^{3/2}\sqrt{2\pi}} \exp\{\xi c - \frac{1}{2}(\frac{c^2}{x} + \xi^2 x)\}}{C \cdot \lambda e^{-\lambda x}}$$

The exponential terms cancel out parts of each other, leading to a more tractable expression. We will justify the election of the λ parameter below, at 4.2i).

Routine for the A-R method

Then, let us define the routine for the generation of the Inverse Gaussian. Let $g(x)$ be the PDF of the exponential distribution, where $g(x) = \lambda e^{-\lambda x}$ for $x > 0$, and λ is the rate parameter. We choose a scaling constant C such that $C \cdot g(x)$ is always greater than or equal to $f(x)$, the target distribution.

i) **Choose an adequate proposal distribution**. We chose the exponential distribution, but we can also fix the λ parameter. Since it models the tail behaviour of the exponential distribution, we can fix it to $\lambda = \xi^2/2$, this way it has a similar behaviour to the Inverse Gaussian for large numbers of x. Let us justify this further; since the inverse Gaussian PDF has the following form:

$$f(x; c, \xi) = \frac{c}{x^{3/2}\sqrt{2\pi}} exp\left(\xi c - \frac{c^2}{2x} - \frac{\xi^2 x}{2}\right)$$

for $x > 0$, at large values of x (in the tail of the distribution), the dominant term in the exponential function is $-\frac{\xi^2 x}{2}$, which suggests that the Inverse Gaussian PDF decays approximately as:

$$f(x) \approx exp\left(-\frac{\xi^2 x}{2}\right)$$

and hence, choosing as λ the value of $\xi^2/2$, we can see that it decays as $exp(-\frac{\xi^2 x}{2})$, and the decays align. With this simple justification, we have improved the efficiency of the Acceptance-Rejection method, especially in the tail.

Now, we can optimize and calibrate the C parameter. For the C, we will approximate it numerically with the ratios of $f(x)/g(x)$, see the argumentation and code below.

ii) **Generate a candidate sample**. We sample X^* from the exponential distribution with rate λ, and U from the uniform distribution $U(0,1)$.

iii) **Acceptance criterion**. We first compute the acceptance probability $\frac{f(X^*)}{Cg(X^*)}$. We accept the candidate X^* if:

$$U \leq \frac{f(X^*)}{Cg(X^*)}$$

Otherwise, we reject X^* and repeat the process until a sample is accepted.

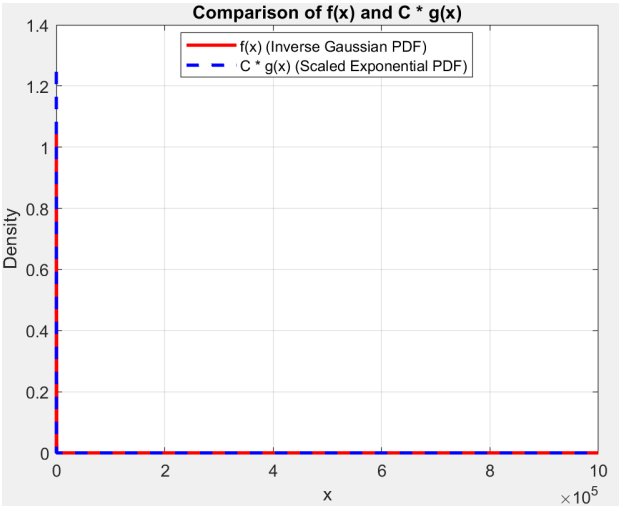
Our approach to tune scale C accordingly is based on the following code:

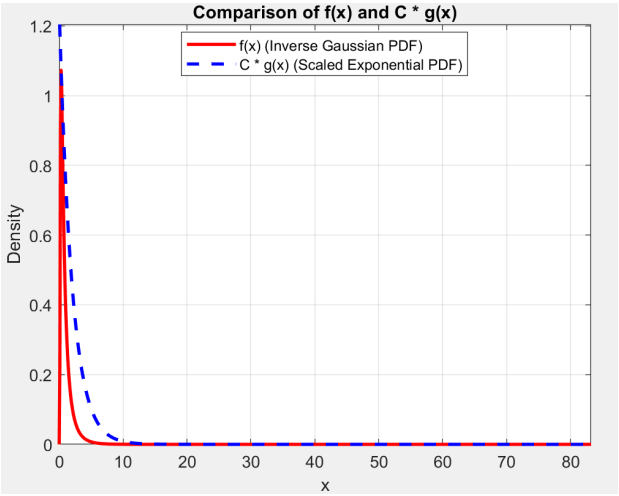
```
function C = tune_scaling_constant(c, xi)
    % Range for x
    x_values = linspace(1e-3, 1e6, 1e7);  % the pdf is not defined on
        x = 0, so we avoid it.

    % Compute f(x) (Inverse Gaussian PDF)
    f_values = arrayfun(@(x) inverse_gaussian_pdf(x, c, xi), x_values)
        ;

    % Compute g(x) (Exponential PDF)
    lambda = xi^2/2;
    g_values = arrayfun(@(x) exponential_proposal(x, lambda), x_values
        );

    % Compute the ratio f(x) / g(x) for values above the tolerance
    ratio_values = f_values ./ g_values;

    C = max(ratio_values);

    fprintf('Tuned scaling constant C: %.4f\n', C);

    figure;
    plot(x_values, f_values, 'r', 'LineWidth', 2); % Plot f(x) in red
    hold on;
    plot(x_values, C * g_values, 'b--', 'LineWidth', 2); % Plot C * g(
        x) in blue
    xlabel('x');
    ylabel('Density');
    title('Comparison of f(x) and C * g(x) with tolerance');
    legend('f(x) (Inverse Gaussian PDF)', 'C * g(x) (Scaled
        Exponential PDF)', 'Location', 'Best');
    grid on;
    hold off;
end

% Inverse Gaussian PDF
function pdf = inverse_gaussian_pdf(x, c, xi)
    if x <= 0
        pdf = 0;
    else
        pdf = (c / (x^(3/2) * sqrt(2 * pi))) * exp(xi * c - 0.5 * (c^2
            / x + xi^2 * x));
    end
end

```

```
% Exponential proposal PDF
function pdf = exponential_proposal(x, lambda)
    if x <= 0
        pdf = 0;
    else
        pdf = lambda * exp(-lambda * x);
    end
end
```

We do various assumptions:

- Our range for x is $I = [1e^{-3}, 1e^{6}]$ and we make use of $1e^{7}$ equal spaced points.
- The optimal C is attained at the maximum of the ratios of the PDFs along the x-values. In this case, it is an approximation, since I is our range discretization along the x-axis.
- We also plot both lines ($f(x)$ and $C \cdot g(x)$), to make sure that our pseudo-optimal C complies with $f(x) \leq C \cdot g(x)$ for all x in our range.





The result for our calibrated constant C is:

```
    Tuned scaling constant C: 2.4944
```

Once we have calibrated our C, we can plug it in our Acceptance-Rejection Method code, which is the following:

```
function HA_StochSim_Ex4
    % Parameters of the Inverse Gaussian distribution
    c = 1;
    xi = 1;
    n_samples = 1e6; % Number of samples wanted

    lambda = xi^2/2;  % Parameter for the exponential distribution
    C = 2.4944; % Tuned scaling constant

    % Initialize sample array
    samples = zeros(n_samples, 1);
    num_accepted = 0;

    % Generate samples using the acceptance-rejection method
    while num_accepted < n_samples
        U1 = rand;
        X_star = -log(U1) / lambda; % inversion method for the
            exponential distribution

        U = rand;

```

```
        accept_prob = inverse_gaussian_pdf(X_star, c, xi) / (C *
            exponential_proposal(X_star, lambda));

        % Accept or reject the candidate
        if U <= accept_prob
            num_accepted = num_accepted + 1;
            samples(num_accepted) = X_star;
        end
    end

    % Compute the sample mean, squared mean, and variance
    sample_mean = mean(samples);
    sample_squared_mean = mean(samples.^2);
    sample_variance = var(samples);

    fprintf('Sample Mean: %.4f\n', sample_mean);
    fprintf('Sample Squared Mean: %.4f\n', sample_squared_mean);
    fprintf('Sample Variance: %.4f\n', sample_variance);

    % Compare with theoretical values
    theoretical_mean = c / xi;
    theoretical_variance = c / xi^3;

    fprintf('Theoretical Mean: %.4f\n', theoretical_mean);
    fprintf('Theoretical Squared Mean: %.4f\n', theoretical_mean^2 +
        theoretical_variance);
    fprintf('Theoretical Variance: %.4f\n', theoretical_variance);
end

% Inverse Gaussian pdf
function pdf = inverse_gaussian_pdf(x, c, xi)
    if x <= 0
        pdf = 0;
    else
        pdf = (c / (x^(3/2) * sqrt(2 * pi))) * exp(xi * c - 0.5 * (c^2
            / x + xi^2 * x));
    end
end

% Exponential proposal PDF
function pdf = exponential_proposal(x, lambda)
    if x <= 0
        pdf = 0;
    else
        pdf = lambda * exp(-lambda * x);
    end
end
```

This is the *output* at which we arrive:

```
Sample Mean: 1.0015
Sample Squared Mean: 2.0076
Sample Variance: 1.0045
Theoretical Mean: 1.0000
Theoretical Squared Mean: 2.0000
Theoretical Variance: 1.0000
```

Let us do a convergence algorithm, to check that effectively the sample values are tending to the theoretical ones. In this convergence algorithm, we are going also to compute the confidence intervals, for the highest number of samples that we compute ($256e^6$).

We use the large-sample approximation based on the Central Limit Theorem to construct confidence intervals. For a large number of samples n, the sample mean $\bar{X}$ is approximately normal distributed with mean $\mathbb{E}[X]$

and variance $\mathbb{V}\text{ar}/n$. The (95%) confidence interval for $\mathbb{E}$ is hence given by:

$$\bar{X} \pm z_{0.975} \cdot \frac{\sigma}{\sqrt{n}}$$

where $z_{0.975} \approx 1.96$ is the z-value for a 95% confidence interval, $\bar{X}$ is the sample mean, σ^2 is the sample variance and n the number of samples computed.

For $\mathbb{E}[X^2]$ the confidence interval is similarly computed:

$$\bar{X^2} \pm z_{0.975} \cdot \frac{\sigma_{X^2}}{\sqrt{n}}.$$

For normally distributed data (which we assume for a large number of samples), $(n-1)S^2$, where S^2 is the sample variance ($S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$, follows a chi-squared distribution with $n-1$ degrees of freedom:

$$\frac{(n-1)S^2}{\sigma^2} \sim \chi^2_{n-1}.$$

For a 95% confidence interval, we use the properties of the χ^2 distribution, and the formula is:

$$\left(\frac{(n-1)S^2}{\chi^2_{0.975,n-1}}, \frac{(n-1)S^2}{\chi^2_{0.025,n-1}} \right).$$

Now, in the code:

```
function convergence_Ex4
    seed = 1234;
    rng(seed);
    % Parameters of the Inverse Gaussian distribution
    c = 1;
    xi = 1;
    tolerance = 1e-4; % Tolerance for comparison

    % Initialize parameters for sample generation
    lambda = xi^2/2;  % Parameter for the exponential distribution
    C = 2.4944; % Tuned scaling constant

    % Initialize sample array
    num_samples = 256e6; % Starting number of samples
    samples = zeros(num_samples, 1);
    sample_mean = 0;
    sample_squared_mean = 0;
    sample_variance = 0;
    diff_mean = inf;
    diff_squared_mean = inf;
    diff_variance = inf;
    % Compare with theoretical values
    theoretical_mean = c / xi;
    theoretical_variance = c / xi^3;
    theoretical_squared_mean = theoretical_mean^2 +
       theoretical_variance;

    while (diff_mean > tolerance || diff_squared_mean > tolerance ||
       diff_variance > tolerance) && num_samples < 4e8
        num_accepted = 0;

        while num_accepted < num_samples
            U1 = rand;
            X_star = -log(U1) / lambda; % inversion method
            U = rand;
            accept_prob = inverse_gaussian_pdf(X_star, c, xi) / (C *
               exponential_proposal(X_star, lambda));
            if U <= accept_prob
```

```
                num_accepted = num_accepted + 1;
                samples(num_accepted) = X_star;
            end
        end

        % Compute the sample mean, squared mean, and variance
        sample_mean = mean(samples);
        sample_squared_mean = mean(samples.^2);
        sample_variance = var(samples);

        % Compute differences from theoretical values
        diff_mean = abs(sample_mean - theoretical_mean)
        diff_squared_mean = abs(sample_squared_mean -
            theoretical_squared_mean)
        diff_variance = abs(sample_variance - theoretical_variance)

        % Compute 95% confidence intervals
        z_975 = 1.96; % Z-value for 95% confidence interval
        n = num_samples;

        % Confidence interval for mean
        std_error_mean = sqrt(sample_variance / n);
        CI_mean = [sample_mean - z_975 * std_error_mean, sample_mean +
            z_975 * std_error_mean];

        % Confidence interval for squared mean
        sample_variance_X2 = var(samples.^2);
        std_error_squared_mean = sqrt(sample_variance_X2 / n);
        CI_squared_mean = [sample_squared_mean - z_975 *
            std_error_squared_mean, sample_squared_mean + z_975 *
            std_error_squared_mean];

        % Confidence interval for variance
        chi2_975 = chi2inv(0.975, n-1);
        chi2_025 = chi2inv(0.025, n-1);
        CI_variance = [(n-1) * sample_variance / chi2_975, (n-1) *
            sample_variance / chi2_025];

        % Print the results
        fprintf('Sample Mean: %.4f\n', sample_mean);
        fprintf('95%% Confidence Interval for Mean: [%.4f, %.4f]\n',
            CI_mean(1), CI_mean(2));

        fprintf('Sample Squared Mean: %.4f\n', sample_squared_mean);
        fprintf('95%% Confidence Interval for Squared Mean: [%.4f, %.4
            f]\n', CI_squared_mean(1), CI_squared_mean(2));

        fprintf('Sample Variance: %.4f\n', sample_variance);
        fprintf('95%% Confidence Interval for Variance: [%.4f, %.4f]\n
            ', CI_variance(1), CI_variance(2));

        fprintf('Theoretical Mean: %.4f\n', theoretical_mean);
        fprintf('Theoretical Squared Mean: %.4f\n',
            theoretical_squared_mean);
        fprintf('Theoretical Variance: %.4f\n', theoretical_variance);

        % Increase number of samples
        if diff_mean > tolerance || diff_squared_mean > tolerance ||
            diff_variance > tolerance
            num_samples = num_samples * 2;
            samples = zeros(num_samples, 1); % Reset samples array
```

```
            fprintf('Increasing number of samples to %d\n',
                num_samples);
        end
        if num_samples > 4e8 && (diff_mean > tolerance ||
            diff_squared_mean > tolerance || diff_variance > tolerance)
            fprintf('Reached the limit for the number of samples\n');
        end
    end
end

% Inverse Gaussian pdf
function pdf = inverse_gaussian_pdf(x, c, xi)
    if x <= 0
        pdf = 0;
    else
        pdf = (c / (x^(3/2) * sqrt(2 * pi))) * exp(xi * c - 0.5 * (c^2
            / x + xi^2 * x));
    end
end

% Exponential proposal PDF
function pdf = exponential_proposal(x, lambda)
    if x <= 0
        pdf = 0;
    else
        pdf = lambda * exp(-lambda * x);
    end
end
```

Due to the limited computational power of my personal computer, we capped the algorithm at $4e8$ accepted samples. The output for $256e6$ accepted samples is as follows:

```
Increasing number of samples to 256000000

diff_mean =

   2.7774e-04


diff_squared_mean =

   7.3169e-04


diff_variance =

   1.7614e-04

Sample Mean: 1.0002
95% Confidence Interval for Mean: [1.0001, 1.0003]
Sample Squared Mean: 2.0007
95% Confidence Interval for Squared Mean: [2.0000, 2.0014]
Sample Variance: 1.0003
95% Confidence Interval for Variance: [1.0002, 1.0005]
Theoretical Mean: 1.0000
Theoretical Squared Mean: 2.0000
Theoretical Variance: 1.0000
Increasing number of samples to 512000000
Reached the limit for the number of samples
```

Since the differences have significantly decreased, we conclude that this justification is sufficient to justify that it provides an approximation for Inverse Gaussian samples.

Exercise 5. Write a discrete event simulation program to perform a single run for the following model.

Ships arrive at a harbor with interarrival times that are i.i.d. Weibull distributed (see exercise $II.2.1$ of [AG]) with a mean of 1.25 days. The harbor has a dock with two berths and two cranes for unloading the ships. Ships arriving when both berths are occupied join a FIFO queue until they can moor at a berth. The time for one crane to unload a ship has a $Normal(1, 0.1)$ distribution which is truncated at its left-hand side at 0.5 and at its right-hand side at 1.5. If there are no other ships in the harbor when a ship arrives at a berth, unloading of the ship will be done by both cranes and the unloading time is cut in half. When this happens, and a second ship moors at the other berth in the meantime, unloading will then need to wait until unloading of the first ship is completed. Whenever there are other ships in the harbor upon a ship's arrival, however, the ship will be unloaded by a single crane. Assuming that no ships are in the harbor at time 0, the management wants to know the average waiting time of the ships until they are unloaded, as well as the waiting time of the longest waiting ship during these 90 days. What are your estimates? How reliable do you deem your estimates?

First of all, notice that we can define the waiting times in two ways:

(i) The waiting time of a ship being understood as the maximum between the difference of the last departure (that lets our ship occupy a crane and hence go into service) and the arrival of our ship and zero.

(ii) The waiting time of a ship being understood as the previous waiting time plus the service time of that ship. In other words, waiting time is understood as the previous waiting time adding the time until the ship is unloaded.

For the sake of practicing different situations, we will tackle both cases, and we are going to provide a confidence interval for the mean of the waiting time of type (i) [from now on, Mean((i))] and for the maximum of the waiting time of type (i) [from now on, Max((i))] and also a confidence interval for the mean of the waiting time of type (ii) [from now on, Mean((ii))] and for the maximum of the waiting time of type (ii) [from now on, Max((ii))] along 90 days.

Also, we compute these results at the same time, so it is (roughly) equally computationally intensive as computing just one result.

Disclaimer: These confidence intervals (both for (i) and for (ii)) will not be tightened to a specific accuracy, they will only be 95% confidence intervals for the number of simulations that we will perform of the run of the 90 days. This number of simulations is going to be $1e6$. More precisely, our estimates are going to be measured as the 5-percentile and the 95-percentile of the values of Max(i), Max(ii), Mean(i), Mean(ii) along our $1e6$ simulations of the 90 days.

Let us list the different characteristics of the problem:

- $T_n := A_{n+1} - A_n \sim$ Weibull($f(\beta)$, $g(\beta)$) such that the mean of the Weibull distribution is 1.25 (days). Hence, we will need to calibrate the parameters to find $f(\beta), g(\beta)$. T_n represents the interarrival time between the n-th ship and the $(n+1)$-st ship.
- According to (i), $W_{n+1} = [D_n - A_{n+1}]^+$ are the waiting times, defined as the maximum of the difference of the departure of the n-th ship (D_n) and the arrival of the $n+1$-st ship (A_{n+1}) and 0.
- Also according to (i), $D_n = A_n + W_n + V_n$, where V_n is the service time of customer n, i.e. the unloading time for the ship n.
- According to (ii), $W_{n+1} = [D_n - A_{n+1}]^+ + V_{n+1}$, where we sum the unloading time of the ship and the waiting time, hence actually becoming the "time in harbor" of the ship n.
- According to (ii), now the departure time of D_n would be $D_n = A_n + W_n$, since it is the time at which the n-th ship arrived plus the "time in harbor" of this ship.
- Back to the general case (both for (i) and (ii)), the assumption of the problem is that:
  - If there are no boats at the harbor, then a boat occupies both cranes and the service time $V_n = \frac{1}{2} \cdot N(1, 0.1)_{\text{TRUNC}}$.
  - Else, $V_n = N(1, 0.1)_{\text{TRUNC}}$.

  Notice that $N(1, 0.1)_{\text{TRUNC}} = N(1, 0.1)$ except at the points smaller than 0.5 and larger than 1.5. In practice, to sample from $N(1, 0.1)_{\text{TRUNC}}$ we will sample from $N(1, 0.1)$ and, if the sample is smaller than 0.5, we will set it to 0.5 and if it is larger than 1.5, we will set it to 1.5.

Also, we assume that when there are no boats, cutting the time in half means that, once we have a sample from $N(1, 0.1)_{\text{TRUNC}}$, we divide it by 2, and not in the opposite order.

Notice that we cannot apply the Lindley recursion (at least not in a trivial way), since there are cases at which there is just one server (the cranes act like one server when there are no ships at the harbor) but also cases at which there are two servers (when there are ships at the harbor at a ship's arrival, a ship will occupy only one crane). Therefore, this is not a (typical) G/G/1 queue.

To sum up, now that we have defined the problem, our objectives are, both for Type (i) and Type (ii) waiting times:

i) To calculate $\max(W_i)$ for $i = 1, 2, \ldots, 90$, that is, the waiting time of the longest waiting time of a ship during 90 days. Notice that a single run consists in simulating 90 days of the harbor.

ii) To calculate $\frac{1}{90}\sum_{i=1}^{90} W_i$, that is, the average waiting time of the ships during these 90 days of the harbor.

iii) *Extra*. Justify the estimates and their reliability.

Let's first calibrate the Weibull distribution with the help of Exercise 2 (2.1 [AG]). Let's suppose we have a Weibull distribution with CDF $\mathbb{P}(X \leq x) = 1 - e^{-x^\beta}$, $x > 0$. Given the CDF, to find the mean (which should be equal by assumption to 1.25), we can compute the PDF via calculating its derivative and then we can find the expected value. Hence, given the CDF:

$$PDF = \frac{d}{dx}\left(1 - e^{-x^\beta}\right) = \beta \cdot x^{\beta-1} \cdot e^{-x^\beta}.$$

Hence, the mean (expected value) should be given by:

$$\mu = \int_0^x x \cdot \beta \cdot x^{\beta-1} \cdot e^{-x^\beta} dx = \int_0^x \beta \cdot x^\beta \cdot e^{-x^\beta} dx = 1.25$$

Now, substituting $u = x^\beta$,

$$du = \beta x^{\beta-1} dx \implies dx = \frac{du}{\beta x^{\beta-1}} = \frac{du}{\beta u^{(\beta-1)/\beta}} = \left(u^{1/\beta-1}\right)\frac{du}{\beta}$$

$$\mu = \int_0^\infty \left(u^{1/\beta-1}\right)\frac{du}{\beta} \cdot \beta \cdot u \cdot e^{-u} = \int_0^\infty \left(u^{1/\beta-1}\right) \cdot u \cdot e^{-u} du = 1.25$$

Let's remember the Gamma function:

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$$

Now, back to our μ:

$$\mu = \int_0^\infty \left(u^{1/\beta-1}\right) \cdot u \cdot e^{-u} du = \int_0^\infty u^{1/\beta} \cdot e^{-u} du = 1.25$$

Using that $1/\beta = z - 1$, we get $z = \frac{\beta+1}{\beta}$. Therefore, we see that $\mu = \Gamma(z = \frac{\beta+1}{\beta}) = 1.25$. Via WolframAlpha's calculator, we obtain that the positive solution (approximately) of β for μ is

$$\beta_1 \approx 0.709426714391227$$

Hence, since we have a Weibull of CDF $F(x) = 1 - e^{-x^\beta}$, as the General Weibull is of CDF $F(x) = 1 - e^{(x/\lambda)^\beta}$, our $\lambda = 1 = f(\beta)$ which is the scale parameter and our β is β_1, the shape parameter. Therefore, $T_n \sim Weibull(\beta = \beta_1, \lambda = 1)$. We will make use of $T_n \sim Weibull(\beta = \beta_1, \lambda = 1)$, and, using our results from Exercise 2 (2.1 [AG]), using $rand()$ (that is, uniform sampling) we will simulate $Weibull$ samples.

Once calibrated T_n, we can already compute our W_i, using C++, in order to get a more efficient solution (via objects such as min-heaps).

For both i) and ii), we need to compute $W_1, W_2, \ldots, W_{90}$. Hence, we provide the following C++ code:

```
#include <iostream>
#include <queue>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;
using P = pair<double,int>;
using PQ = priority_queue<P, vector<P>, greater<P>>;
```

```
// We use a priority_queue such that we use a min-heap (the .front() gives us the smallest value)
// This translates for us that we get the earliest event (the event with smallest time value)

int maxqueuelength;
void add_Qmax(int i){
    maxqueuelength = max(maxqueuelength, i);
}

double weibull_rv(double lambda, double beta){
    double u = rand()/(double)RAND_MAX; // rand() produces a uniform sample from 0 to RAND_MAX
    double w = pow(-log(u),1/beta);
    return w;
}

double pi(){
// well-known function in the C++ domain to get pi up to certain precision
    return atan(1)*4;
}

double trunc_normal(){ // Box - Muller Transformation
    double v = rand()/(double)RAND_MAX;
    double u = rand()/(double)RAND_MAX;
    double z = sqrt(-2*log(u))*cos(2*pi()*v);
    // We truncate the normal such that values greater than 1.5 are capped to 1.5
    // and values smaller that 0.5 are set to 0.5.
    double mean = 1.0;
    double stddev = sqrt(0.1); // Standard deviation is sqrt(0.1)
    z = mean + stddev * z;
    if(z > 1.5)
        z = 1.5;
    else if(z < 0.5)
        z = 0.5;
    return z;
}

pair<pair<double,double>, pair<double,double>> days_run_90(){
    double simulation_clock = 0.;
    int ARRIVAL = 1; // arrival of a boat
    int DEPARTURE = 2; // departure of a boat
    // we also use DEPARTURE + 1,
    // which signals the case that a boat that was using two cranes departs

    queue<double> fifoqueue;
    double totalNumberOfArrivals = 0.;
    double totalWaitingTimeOfCustomers = 0.;
    double totalUnloadingTime = 0.;
    double maximumWaitingTimeOfCustomers = 0.;
    double totalWaitingTimeOfCustomersPlusUnloadingTime = 0.;
    double maximumWaitingTimeOfCustomersPlusUnloadingTime = 0.;
    int maximumQueueLength = 0;
    int currentNumberOfCranesBusy = 0;
    double beta = 0.709426714391227;
    double firstArrivalTime = weibull_rv(1,beta);
    PQ events;
    events.push(make_pair(firstArrivalTime, ARRIVAL));
    int c = 2;
    while(events.top().first < 90){ // 90 days
        //cout << events.top().first << " " << events.top().second << endl;
        // handle next event
        double nextEventTime = events.top().first;
        int nextEventType = events.top().second;
```

```
events.pop();
simulation_clock = nextEventTime;


if(nextEventType == ARRIVAL){
    // handle arrival
    totalNumberOfArrivals += 1;
    if(currentNumberOfCranesBusy < c){
        if(currentNumberOfCranesBusy == 0 & fifoqueue.size() == 0){
        // this is the condition that there are no boats at the harbor
            currentNumberOfCranesBusy = 2; // the boat uses two cranes
            double serviceTime = trunc_normal()/2.;
            events.push(make_pair(simulation_clock+serviceTime, DEPARTURE+1));
            // departure + 1 means that the boat that leaves was using two cranes
            totalUnloadingTime += serviceTime;
            totalWaitingTimeOfCustomersPlusUnloadingTime += serviceTime;
            maximumWaitingTimeOfCustomersPlusUnloadingTime = max(maximumWaitingTimeOfCustomersl
        }
        else if(fifoqueue.size() == 0){
        // we handle the queues in the departure of the boats
        // in this case we handle the situation in which there arrives a new boat
        // and there is a free crane
            double serviceTime = trunc_normal();
            currentNumberOfCranesBusy += 1;
            events.push(make_pair(simulation_clock+serviceTime, DEPARTURE));
            totalUnloadingTime += serviceTime;
            totalWaitingTimeOfCustomersPlusUnloadingTime += serviceTime;
            maximumWaitingTimeOfCustomersPlusUnloadingTime = max(maximumWaitingTimeOfCustomersl
        }
        // notice that the case that currentNumberOfCranesBusy < c
        // and fifoqueue.size() > 0 does not happen by construction
    }
    else{ // currentNumberOfCranesBusy == c
        fifoqueue.push(nextEventTime); // we add it to the FIFO Queue
    }
    double nextArrivalTime = simulation_clock + weibull_rv(1,beta);
    events.push(make_pair(nextArrivalTime, ARRIVAL));
}
else{
    if(nextEventType == DEPARTURE+1){
            currentNumberOfCranesBusy -=2;
    }
    else{
        currentNumberOfCranesBusy -=1;
    }
    // handle departure
    if(fifoqueue.size() > 0){
        // there is a customer (or more) queued
        // we need to account for the situation in which the boat uses two cranes
        // hence, we fill the cranes given by the customer order in the FIFO queue
        int n = fifoqueue.size();
        maximumQueueLength = max(maximumQueueLength, n);

        while(fifoqueue.size() > 0 & currentNumberOfCranesBusy < c){
            // put into service and schedule departure

            cout << "INFORMATION ABOUT FILLING OF THE QUEUE:" << endl;
            cout << "currentNumberOfCranesBusy: " << currentNumberOfCranesBusy << endl;
            cout << "fifoqueue.size(): " << fifoqueue.size() << endl;
            cout << "fifoqueue.front(): " << fifoqueue.front() << endl;
```

```
                currentNumberOfCranesBusy += 1;
                double nextCustomer = fifoqueue.front();
                fifoqueue.pop();

                totalWaitingTimeOfCustomers += (simulation_clock - nextCustomer);

                maximumWaitingTimeOfCustomers = max(maximumWaitingTimeOfCustomers,
                                                    simulation_clock - nextCustomer);

                double nextServiceTime = trunc_normal();

                totalWaitingTimeOfCustomersPlusUnloadingTime +=
                (simulation_clock - nextCustomer + nextServiceTime);

                maximumWaitingTimeOfCustomersPlusUnloadingTime =
                max(maximumWaitingTimeOfCustomersPlusUnloadingTime,
                    simulation_clock - nextCustomer + nextServiceTime);

                events.push(make_pair(simulation_clock+nextServiceTime, DEPARTURE));
            }
        }
    }

    cout << "Day: " << simulation_clock << endl;

    cout << "Total number of arrivals: "
    << totalNumberOfArrivals << endl;

    cout << "Total waiting time of customers: "
    << totalWaitingTimeOfCustomers << endl;

    cout << "Total unloading time: "
    << totalUnloadingTime << endl;

    cout << "Current number of cranes busy: "
    << currentNumberOfCranesBusy << endl;

    cout << "Current queue length: "
    << fifoqueue.size() << endl;

    cout << "Maximum waiting time of customers: "
    << maximumWaitingTimeOfCustomers << endl;

    cout << "Maximum waiting time of customers plus unloading time: "
    << maximumWaitingTimeOfCustomersPlusUnloadingTime << endl;

    cout << "Total waiting time of customers plus unloading time: "
    << totalWaitingTimeOfCustomersPlusUnloadingTime << endl;

    cout << "Maximum queue length: " <<
    maximumQueueLength << endl;
}
add_Qmax(maximumQueueLength);
return make_pair(
       make_pair(
        maximumWaitingTimeOfCustomers,
        totalWaitingTimeOfCustomers/totalNumberOfArrivals)
       ,
       make_pair(
        maximumWaitingTimeOfCustomersPlusUnloadingTime,
        totalWaitingTimeOfCustomersPlusUnloadingTime/totalNumberOfArrivals)
```

```
            );
}

int main(){
    // we also time our code
    clock_t start = clock();
    int num_simulations = 1;
    vector<double> maxs(num_simulations);
    vector<double> means(num_simulations);
    vector<double> maxs_pls_unloading(num_simulations);
    vector<double> means_pls_unloading(num_simulations);
    pair<pair<double,double>, pair<double,double>> result;
    unsigned int seed = 1234;
    srand(seed);
    for(int i =0; i < num_simulations; ++i){
        result = days_run_90();
        maxs[i] = result.first.first;
        means[i] = result.first.second;
        maxs_pls_unloading[i] = result.second.first;
        means_pls_unloading[i] = result.second.second;
        // cout << maxs[i] << " " << means[i] << endl;
    }
    // we now compute the 95% confidence intervals for the maximum and the mean
    double alpha = 0.05;
    cout << "Maximum queue length is: " << maxqueuelength << endl;
    sort(maxs.begin(), maxs.end());
    sort(means.begin(), means.end());
    sort(maxs_pls_unloading.begin(), maxs_pls_unloading.end());
    sort(means_pls_unloading.begin(), means_pls_unloading.end());

    int q025 = int(num_simulations*alpha/2);
    int q975 = int(num_simulations*(1-alpha/2));

    double max_CI = maxs[q025], max_CI2 = maxs[q975];
    double mean_CI = means[q025], mean_CI2 = means[q975];
    double max_CI_pls_unloading =
    maxs_pls_unloading[q025], max_CI2_pls_unloading = maxs_pls_unloading[q975];
    double mean_CI_pls_unloading =
    means_pls_unloading[q025], mean_CI2_pls_unloading = means_pls_unloading[q975];

    cout << "The maximum waiting time of customers is between "
    << max_CI << " and " << max_CI2 << endl;
    cout << "The mean waiting time of customers is between "
    << mean_CI << " and " << mean_CI2 << endl;
    cout << "The maximum waiting time understood as waiting time until unloaded is between "
    << max_CI_pls_unloading << " and " << max_CI2_pls_unloading << endl;
    cout << "The mean waiting time understood as waiting time until unloaded is between "
    << mean_CI_pls_unloading << " and " << mean_CI2_pls_unloading << endl;
    clock_t end = clock();
    cout << "Time taken: "
    << ((double)(end - start))/CLOCKS_PER_SEC << " seconds" << endl;
}
```

We provide an explanation of the code in Appendix B. Please refer to it for more details on the technical aspects.

To also justify the estimates and reliability, notice that we have performed $1e6$ simulations of the 90 days, and we have computed confidence intervals for both estimates for (i) and (ii), to ensure the reliability of the values obtained in our simulations.

Hence, this is the output:

```
Maximum queue length is: 19
```

```
The maximum waiting time of customers is between 0.737284 and 3.58506
The mean waiting time of customers is between 0.109253 and 0.64599
The maximum waiting time understood as waiting time until unloaded is between 1.89907 and 4.70568
The mean waiting time understood as waiting time until unloaded is between 0.81614 and 1.51303
Time taken: 56.835 seconds
```

In other words:

- The 95% confidence interval of our $1e6$ simulations along the 90 days for Max((i)) is: $[0.737284, 3.58506]$, in days
- The 95% confidence interval of our $1e6$ simulations along the 90 days for Mean((i)) is: $[0.109253, 0.64599]$, in days
- The 95% confidence interval of our $1e6$ simulations along the 90 days for Max((ii)) is: $[1.89907, 4.70568]$, in days
- The 95% confidence interval of our $1e6$ simulations along the 90 days for Mean((ii)) is: $[0.81614, 1.51303]$, in days

In other words, given our number of simulations ($N = 1e6$) we can be 95% sure that, for a single run, the different estimates that we have computed for the waiting times (in days) for our problem will be found inside the confidence intervals.

Notice that, to get a single run of the 90 days, we would just need to extract one value of the $1e6$-sized vectors for each estimate that we have.

If we want a different run from one of the ones computed, we can also set number of simulations to 1, and the confidence intervals are going to consist in only one value, which represent our estimates for Max((i)), Mean((i)), Max((ii)) and Mean((ii)).

Let us provide a single run of our code (setting number of simulations to 1 and seed to 4321):

```
Maximum queue length is: 2
The maximum waiting time of customers is between 1.11864 and 1.11864
The mean waiting time of customers is between 0.194671 and 0.194671
The maximum waiting time understood as waiting time until unloaded is between 2.22293 and 2.22293
The mean waiting time understood as waiting time until unloaded is between 0.98036 and 0.98036
Time taken: 0.000136 seconds
```

As a remark, this code is capable of performing $1e6$ simulations in under 1 minute and 1 simulation in just 0.000136 seconds.

Lastly, we assess the **reliability of our simulation estimates**.

We started by sorting the results of the Monte Carlo simulations, to work with the ordered empirical distribution of our outcomes for the mean and maximum waiting times (both (i) and (ii)). We then extracted the 2.5th and 97.5th percentile from the sorted lists to form the 95% confidence intervals, where $\alpha = 0.05$, $q025$ refers to the 2.5 percentile and $q975$ refers to the 97.5% percentile.

Our approach was non-parametric, in the sense that we avoided making parametric assumptions about the underlying distribution of the data. This approach is robust because it does not rely on the distributional assumptions (for instance normality) that parametric confidence intervals would require.

The reliability of this method depends on the number of simulations. With a large number of simulations (in our case $N = 1e6 = 10^6$), the empirical confidence intervals should be reliable because the Monte Carlo error (which is the uncertainty due to finite sampling) gets smaller as $N \to \infty$.

Appendices

Appendix A: Exercise 1: JavaScript code (p5.js) for the painting of dart throws

```
function setup() {
  createCanvas(500, 500);
    background(255);
  radius = (height/2.0+width/2.0)/2.0;
   stroke(0);
   strokeWeight(7);
   point(width/2,height/2);
   strokeWeight(0);
   text("(0,0)",width/2,height/2);
}

function drawPoint( _x,  _y, _colors){
    strokeWeight(3);
    stroke(_colors);
    point(_x+width/2, height/2 - _y);
    strokeWeight(0);
}

x = 0;
y = 0;

color1 = [255,0,0]; // red
color2 = [0,255,0]; // green

function draw() {
   x = random(-radius,radius);
   y = random(0,radius);
   if ( x*x + y*y <= radius*radius) {
     drawPoint(x,y,color2);
   } else {
     drawPoint(x,y,color1);
   }
}
```

Appendix B: Exercise 5 - Review of the C++ code

Instead of going line by line, we will show a small preview of the code when the *cout*'s are uncommented, thus showing each step of the process during the simulation (seed $= 1234$, num simulations $= 1$). Another small disclaimer, the "Day" parameter is clearly not typically an integer, since it does not represent the day but the time in days that an event occurs.

```
1.77108 1
Day: 1.77108
Total number of arrivals: 1
Total waiting time of customers: 0
Total unloading time: 0.541525
Current number of cranes busy: 2
Current queue length: 0
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.541525
Total waiting time of customers plus unloading time: 0.541525
Maximum queue length: 0
```

Notice that, at 1.77108 we have the first arrival, since the first pair of the output signals the event time and the event type, which is 1 for arrivals, 2 for departures and 3 for departures of boats that occupy 2 cranes. This boat now occupies 2 cranes.

```
2.31261 3
Day: 2.31261
Total number of arrivals: 1
Total waiting time of customers: 0
Total unloading time: 0.541525
Current number of cranes busy: 0
Current queue length: 0
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.541525
Total waiting time of customers plus unloading time: 0.541525
Maximum queue length: 0
```

Now, at 2.31261, this boat that was occupying 2 cranes leaves. Therefore, the current number of cranes busy is 0, and there has been no waiting times at the moment (of type (i)), of type (ii) we have had one waiting time (only accounting for the unloading time of the first ship).

```
2.38573 1
Day: 2.38573
Total number of arrivals: 2
Total waiting time of customers: 0
Total unloading time: 0.86633
Current number of cranes busy: 2
Current queue length: 0
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.541525
Total waiting time of customers plus unloading time: 0.86633
Maximum queue length: 0
2.71054 3
Day: 2.71054
Total number of arrivals: 2
Total waiting time of customers: 0
Total unloading time: 0.86633
Current number of cranes busy: 0
Current queue length: 0
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.541525
Total waiting time of customers plus unloading time: 0.86633
Maximum queue length: 0
```

Same situation.

```
3.04563 1
```

```
Day: 3.04563
Total number of arrivals: 3
Total waiting time of customers: 0
Total unloading time: 1.41375
Current number of cranes busy: 2
Current queue length: 0
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.547421
Total waiting time of customers plus unloading time: 1.41375
Maximum queue length: 0
3.05724 1
Day: 3.05724
Total number of arrivals: 4
Total waiting time of customers: 0
Total unloading time: 1.41375
Current number of cranes busy: 2
Current queue length: 1
Maximum waiting time of customers: 0
Maximum waiting time of customers plus unloading time: 0.547421
Total waiting time of customers plus unloading time: 1.41375
Maximum queue length: 0
3.59305 3
INFORMATION ABOUT FILLING OF THE QUEUE:
currentNumberOfCranesBusy: 0
fifoqueue.size(): 1
fifoqueue.front(): 3.05724
Day: 3.59305
Total number of arrivals: 4
Total waiting time of customers: 0.535809
Total unloading time: 1.41375
Current number of cranes busy: 1
Current queue length: 0
Maximum waiting time of customers: 0.535809
Maximum waiting time of customers plus unloading time: 1.49569
Total waiting time of customers plus unloading time: 2.90944
Maximum queue length: 1
```

This situation is critical on the code, and has to be handled with care. Notice that at 3.04563 a boat came and occupied two cranes since there were no ships at the harbor. Another ship came at 3.05724, and since the two cranes were busy, it was therefore put in the FIFO queue, so the current queue length is 1. At time 3.59305, the boat that was occupying two cranes left, and now the boats waiting at the queue can be put into service, and the ship at the front of the queue (the one that waited the most) can be put into service, and occupies just one crane, because when it arrived there were boats at the harbor, and hence should not occupy both of the cranes, even though it has no more ships at the queue. Notice how now we do have a non-zero value for the "Maximum waiting time of customers", which was the waiting time of type (i), i.e. the time spent at the FIFO queue until it was put into service.

```
4.55293 2
Day: 4.55293
Total number of arrivals: 4
Total waiting time of customers: 0.535809
Total unloading time: 1.41375
Current number of cranes busy: 0
Current queue length: 0
Maximum waiting time of customers: 0.535809
Maximum waiting time of customers plus unloading time: 1.49569
Total waiting time of customers plus unloading time: 2.90944
Maximum queue length: 1
```

The next (and last event that we will show in this report) is the event of departure of this boat. Since it was not occupying two cranes, it is a "regular" departure.

Thank you very much for reading this assignment. As it can be shown by the number of pages and the excitement contained in them, I have had a lot of fun doing it, especially exercise 5.

Andreu Boix Torres
Stochastic Simulation 2024-2025