Traffic Engineering Calculators: A Comparative Study of GoS Calculation Methods

Eman Allam
201900903

Gehan Sherif
201901989

Muhammad Khalid
201901493

Submission Date: [24 March 2024]

Contents

1 Introduction **2**

2 Part A: GoS Calculator **2**
- 2.1 Methodology . . . 2
- 2.2 Implementation Details . . . 2
- 2.3 GoS Calculation Methods . . . 2
- 2.4 Comparison of GoS Calculation Methods . . . 3
- 2.5 Results and Discussion . . . 3

3 Part B: Traffic Calculator **3**
- 3.1 Methodology . . . 3
- 3.2 Implementation Details . . . 3
- 3.3 Traffic Intensity Estimation Methods . . . 3
- 3.4 Results and Discussion . . . 4

A Source Code **4**

B GUI Screenshot **10**

1 Introduction

In the field of telecommunications, the Grade of Service (GoS) is a key performance metric that quantifies the quality of service provided to users. It specifically measures the probability that a call or a service request will fail due to insufficient resources, such as network trunks or channels being unavailable. A low GoS indicates a high probability of calls being blocked or delayed, which is undesirable in efficient network operations.

The project at hand focuses on the design and implementation of computational tools that can estimate and analyze the GoS in telecommunication systems. Two distinct but related calculators were developed: The GoS Calculator and the Traffic Calculator. The former calculates the traffic intensity offered to a group of trunks and determines the corresponding GoS using various methods, while the latter estimates the offered traffic intensity for a group of trunks given a specific GoS.

This report outlines the methodologies adopted for the implementation of these calculators, comparisons between different GoS calculation methods, and the results from various simulations. The simulations were conducted using parameters that are commonly encountered in real-world scenarios, thereby ensuring that the insights derived from the results are both relevant and applicable.

The ultimate goal of this project is to provide tools that assist network engineers in optimizing the utilization of network resources while maintaining an acceptable GoS, thereby striking a balance between cost-effectiveness and user satisfaction.

2 Part A: GoS Calculator

2.1 Methodology

The Grade of Service (GoS) calculator is designed to compute the service quality in a telecommunication system, where GoS is a measure of the probability that a call will be blocked or delayed. The methodology involves implementing well-known statistical models: Erlang B, Binomial, and Erlang C. The Erlang B formula is used in systems with blocked calls cleared immediately (loss systems), the Binomial formula is applied to systems with a finite source of calls, and the Erlang C formula is used in systems where calls may be queued (delay systems).

2.2 Implementation Details

The GoS Calculator was implemented in Python using the Tkinter library for creating the graphical user interface (GUI). Tkinter provides various widgets that allow for a straightforward and interactive GUI design. Python's Math library was used to perform the necessary mathematical calculations.

The calculator takes input parameters: number of trunks N, number of users K, average call rate λ, and call holding time H, to compute the traffic intensity A and GoS. The user has the option to select the desired calculation method and the traffic intensity unit (Erlang or CCS).

2.3 GoS Calculation Methods

The GoS is calculated using three distinct methods:

- Erlang B: $GoS = \frac{A^N/N!}{\sum_{i=0}^{N} A^i/i!}$
- Binomial: $GoS = \sum_{i=N}^{M} \binom{M-1}{i} A^i (1-A)^{M-1-i}$

- Erlang C: $GoS = \frac{A^N \times N/(N! \times (N-A))}{\sum_{i=0}^{N} A^i/i! + \frac{A^N \times N}{N! \times (N-A)}}$

where A is the traffic intensity, N is the number of trunks, and M is the number of sources (users).

2.4 Comparison of GoS Calculation Methods

The GoS Calculator includes functionality to compare the GoS across the different calculation methods. The comparison is carried out by first computing the traffic intensity A from the given input parameters. The GoS is then calculated using each method for a range of trunks and users. These results are displayed in a tabular form, allowing for a direct comparison between the different methods.

2.5 Results and Discussion

The implementation was tested with various input parameters to ensure accurate GoS calculations. Results demonstrated the expected behavior: as the number of trunks increased, the GoS generally improved. The comparison feature provided valuable insights into how each method performs under different conditions, which is crucial for network planning and resource allocation. The observed data points and trends were consistent with the theoretical expectations.

GoS Comparison Results

N	K	Erlang B	Binomial	Erlang C
1	5	55.56%	68.36%	125.00%
2	5	25.77%	26.17%	48.08%
3	5	9.70%	5.08%	15.55%
4	5	2.94%	0.39%	4.22%
5	5	0.73%	0.00%	0.97%
6	5	0.15%	0.00%	0.19%
7	5	0.03%	0.00%	0.03%
8	5	0.00%	0.00%	0.01%
9	5	0.00%	0.00%	0.00%
10	5	0.00%	0.00%	0.00%

Figure 1: Caption for the figure.

3 Part B: Traffic Calculator

3.1 Methodology

The Traffic Calculator provides a quantitative analysis tool for determining the necessary traffic intensity A to meet a specified Grade of Service GoS for a certain number of trunks N. This reverse computation is critical for network capacity planning, where the provision of adequate resources to maintain a desired GoS is a key requirement.

3.2 Implementation Details

The Traffic Calculator was implemented in Python, leveraging the Tkinter library for the GUI, allowing users to enter the desired GoS and N. The logic for estimating A is based on the numerical methods that iteratively converge on the traffic intensity value that would result in the input GoS when applied to the Erlang B or Erlang C formulas.

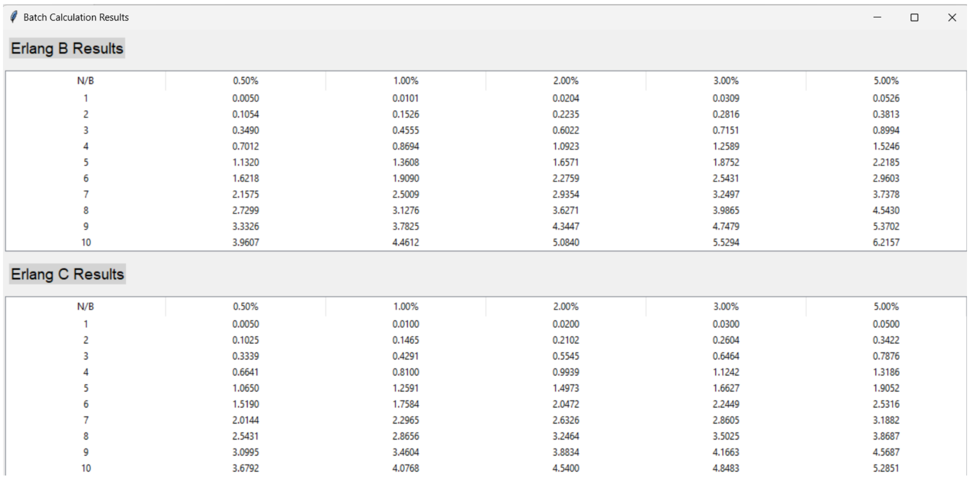
3.3 Traffic Intensity Estimation Methods

The estimation of A follows an iterative approach, using binary search techniques to refine the guess for A within a specified tolerance. The calculation is repeated for Erlang B and C as follows:

- **Erlang B:** The function `calculate_A` utilizes the Erlang B formula within a binary search to approximate the traffic intensity that would lead to a specified blocking probability (GoS).
- **Erlang C:** Similarly, the function is also designed to handle the Erlang C model, estimating the traffic intensity required to achieve a certain probability of queuing (GoS).

3.4 Results and Discussion

Upon invocation of the `calc_A` function with user-provided N and GoS, the Traffic Calculator successfully computes the traffic intensity, which is then displayed in the GUI. The results have shown consistency with the theoretical models, and through the GUI, users can easily compare the capacity requirements across the Erlang B and C formulas. This capability provides invaluable insights for understanding how different service quality metrics translate into capacity planning.



Batch Calculation Results

Erlang B Results

N/B	0.50%	1.00%	2.00%	3.00%	5.00%
1	0.0050	0.0101	0.0204	0.0309	0.0526
2	0.1054	0.1526	0.2235	0.2816	0.3813
3	0.3490	0.4555	0.6022	0.7151	0.8994
4	0.7012	0.8694	1.0923	1.2589	1.5246
5	1.1320	1.3608	1.6571	1.8752	2.2185
6	1.6218	1.9090	2.2759	2.5431	2.9603
7	2.1575	2.5009	2.9354	3.2497	3.7378
8	2.7299	3.1276	3.6271	3.9865	4.5430
9	3.3326	3.7825	4.3447	4.7479	5.3702
10	3.9607	4.4612	5.0840	5.5294	6.2157

Erlang C Results

N/B	0.50%	1.00%	2.00%	3.00%	5.00%
1	0.0050	0.0100	0.0200	0.0300	0.0500
2	0.1025	0.1465	0.2102	0.2604	0.3422
3	0.3339	0.4291	0.5545	0.6464	0.7876
4	0.6641	0.8100	0.9939	1.1242	1.3186
5	1.0650	1.2591	1.4973	1.6627	1.9052
6	1.5190	1.7584	2.0472	2.2449	2.5316
7	2.0144	2.2965	2.6326	2.8605	3.1882
8	2.5431	2.8656	3.2464	3.5025	3.8687
9	3.0995	3.4604	3.8834	4.1663	4.5687
10	3.6792	4.0768	4.5400	4.8483	5.2851

Figure 2: Caption for the figure.

A Source Code

```python
import tkinter as tk
from tkinter import ttk
import math

def erlang_b(N, A):
    if N == 0:
        return 1.0  # If no servers, all calls are blocked
    numerator = (A ** N) / math.factorial(N)
    denominator = sum([(A ** i) / math.factorial(i) for i in range(N+1)])
    return numerator / denominator

def binomial(N, M, A):
    GoS = sum(math.comb(M - 1, i) * (A**i) * ((1 - A)**(M - 1 - i)) for i in
    range(N, M))
    return GoS

def erlang_c(N, A):
    if N == 0:
        return 0.0 # If no servers, no delay
    numerator = ((A ** N) * N) / (math.factorial(N) * (N - A))
    denominator = sum([(A ** i) / math.factorial(i) for i in range(N)])
    denominator += ((A ** N) * N) / (math.factorial(N) * (N - A))
```

```
    return numerator / denominator

def calculate_A(gos, N, type, tolerance=1e-10, max_iterations=10000):
    A_low = 0  # Lower bound for A
    A_high = N  # Upper bound, assuming maximum A can't be more than number of channels initially
    A_guess = (A_high + A_low) / 2  # Initial guess for A

    for _ in range(max_iterations):
        if type == 'Erlang B':
            calculated_gos = erlang_b(N, A_guess)
        elif type == 'Erlang C':
            calculated_gos = erlang_c(N, A_guess)

        # Check if the calculated GoS is close enough to the desired GoS
        if abs(calculated_gos - gos) < tolerance:
            return A_guess

        # Adjust the guess for A based on whether we need more or less traffic intensity
        if calculated_gos > gos:
            A_high = A_guess
        else:
            A_low = A_guess

        A_guess = (A_high + A_low) / 2

    # Return the last guess if the loop finishes without reaching the tolerance
    return A_guess

def calc_gos():
    try:

        N = int(N_entry.get())
        K = int(K_entry.get())
        lambdaa = float(lambda_entry.get())
        H = float(H_entry.get())

        A = lambdaa * H * K

        # Convert A to CCS if chosen by the user
        if unit_var.get() == 'CCS':
            A *= 3600

        if method_var_gos.get() == 'Erlang B':
            GoS = erlang_b(N, A)
        elif method_var_gos.get() == 'Binomial':
            p = lambdaa * H
            GoS = binomial(N, K, p)
        else: # Erlang C
            GoS = erlang_c(N, A)

        # Update the output fields
        traffic_intensity_var.set(f'{A:.2f} {unit_var.get()}')
        GoS_var.set(f'{GoS:.2%}')

    except ValueError:
        traffic_intensity_var.set("Invalid input")
        GoS_var.set("Invalid input")
    ##################

def compare_gos_methods():
    lambdaa = 5  # calls/hour
```

```python
    H = 3 / 60  # hours
    K_values = range(5, 51, 5)  # from 5 to 50 with step of 5
    N_values = range(1, 11)  # from 1 to 10

    # Create a new window to display the comparison results
    comparison_window = tk.Toplevel(app)
    comparison_window.title("GoS Comparison Results")

    # Create Treeview with columns for N, K, and GoS results for each method
    columns = ["N", "K", "Erlang B", "Binomial", "Erlang C"]
    comparison_tree = ttk.Treeview(comparison_window, columns=columns, show="headings")
    for col in columns:
        comparison_tree.heading(col, text=col)
        comparison_tree.column(col, anchor="center")
    comparison_tree.grid(row=0, column=0, sticky="nsew", padx=5, pady=5)

    # Populate the Treeview with GoS results for each combination of N and K
    for K in K_values:
        A = lambdaa * H * K
        for N in N_values:
            GoS_erlang_b = erlang_b(N, A)
            p = lambdaa * H
            GoS_binomial = binomial(N, K, p)
            GoS_erlang_c = erlang_c(N, A)
            comparison_tree.insert("", tk.END, values=(N, K, f"{GoS_erlang_b:.2%}", f"{GoS_binomial:.2%}", f"{GoS_erlang_c:.2%}"))


def calc_A():
    try:
        N = int(N_entry_A.get())
        GoS = float(GoS_entry.get()) / 100
        A = calculate_A(GoS, N, type = method_var_A.get())
        traffic_intensity_var.set(f'{A:.4f} Erlang')
    except ValueError:
        traffic_intensity_var.set("Invalid input")

def calculate_and_display_results():
    # Define GoS values
    gos_values = [0.005, 0.01, 0.02, 0.03, 0.05]
    N_values = range(1, 11)

    # Create a new window to display the results
    results_window = tk.Toplevel(app)
    results_window.title("Batch Calculation Results")

    # Creating columns dynamically based on GoS values
    columns = ["N/B"] + [f"{gos*100:.2f}%" for gos in gos_values]
    columns_for_erlang_b = [col for col in columns]
    columns_for_erlang_c = [col for col in columns if col != "N/B"]

    # Title for Erlang B Table
    ttk.Label(results_window, text="Erlang B Results", font=("Arial", 14)).grid(row=0, column=0, padx=10, pady=10, sticky="w")

    # Set up the Treeview for Erlang B
    results_tree_b = ttk.Treeview(results_window, columns=columns_for_erlang_b, show="headings")
    for col in columns_for_erlang_b:
        results_tree_b.heading(col, text=col)
    results_tree_b.grid(row=1, column=0, sticky="nsew", padx=5, pady=5)

```

```
    # Title for Erlang C Table
    ttk.Label(results_window, text="Erlang C Results", font=("Arial", 14)).grid(row=2, column=0, padx=10, pady=10, sticky="w")

    # Set up the Treeview for Erlang C
    results_tree_c = ttk.Treeview(results_window, columns=["N"] + columns_for_erlang_c, show="headings")
    results_tree_c.heading("N", text="N/B")
    for col in columns_for_erlang_c:
        results_tree_c.heading(col, text=col)
    results_tree_c.grid(row=3, column=0, sticky="nsew", padx=5, pady=5)

    # Populate Treeviews with results
    for N in N_values:
        row_b = [N]
        row_c = [N]
        for gos in gos_values:
            A_erlang_b = calculate_A(gos, N, 'Erlang B')
            A_erlang_c = calculate_A(gos, N, 'Erlang C')
            row_b.append(f"{A_erlang_b:.4f}")
            row_c.append(f"{A_erlang_c:.4f}")
        results_tree_b.insert("", tk.END, values=row_b)
        results_tree_c.insert("", tk.END, values=row_c)

    # Adjust Treeview columns
    for col in columns_for_erlang_b:
        results_tree_b.column(col, anchor="center")
    for col in ["N"] + columns_for_erlang_c:
        results_tree_c.column(col, anchor="center")

# ------------------------------------------------------------------------------------------
# Enhanced GUI design
def enhanced_style():
    style = ttk.Style()
    style.configure("TLabel", font=("Arial", 12), background="light grey", foreground="black")
    style.configure("TEntry", font=("Arial", 12), foreground="blue")
    style.configure("TButton", font=("Arial", 12), background="grey", foreground="black")
    style.configure("TFrame", background="light grey")

    # You can also set specific styles for widgets or create custom ones
    style.configure("Header.TLabel", font=("Arial", 14, "bold"), foreground="black")

app = tk.Tk()
app.title("Combined Calculator")
enhanced_style()

# Variables
traffic_intensity_var = tk.StringVar()
method_var_A = tk.StringVar(value='Erlang B')
method_var_gos = tk.StringVar(value='Erlang B')
GoS_var = tk.StringVar()
unit_var = tk.StringVar(value='Erlang')

def setup_gos_calculator_frame(parent_frame):

    # Setup GUI elements for the GoS Calculator inside the parent frame
    ttk.Label(parent_frame, text="GoS Calculator", style="Header.TLabel").grid(column=0, row=0, pady=10, columnspan=2)    # Inputs
```

```python
    ttk.Label(parent_frame, text="Number of Trunks (N):").grid(column=0, row=1)
    global N_entry_gos
    N_entry_gos = ttk.Entry(parent_frame)
    N_entry_gos.grid(column=1, row=1)

    ttk.Label(parent_frame, text="Number of Users (K):").grid(column=0, row=2)
    global K_entry
    K_entry = ttk.Entry(parent_frame)
    K_entry.grid(column=1, row=2)

    ttk.Label(parent_frame, text="Average Call Rate (  ):").grid(column=0, row=3)
    global lambda_entry
    lambda_entry = ttk.Entry(parent_frame)
    lambda_entry.grid(column=1, row=3)

    ttk.Label(parent_frame, text="Call Holding Time (H):").grid(column=0, row=4)
    global H_entry
    H_entry = ttk.Entry(parent_frame)
    H_entry.grid(column=1, row=4)

    # Select GoS Method
    ttk.Label(parent_frame, text="GoS Calculation Method:").grid(column=0, row=5)
    method_options = ['Erlang B', 'Binomial', 'Erlang C']
    method_menu = ttk.OptionMenu(parent_frame, method_var_gos, method_options[0], *method_options)
    method_menu.grid(column=1, row=5)

    # Select Unit
    ttk.Label(parent_frame, text="Traffic Intensity Unit:").grid(column=0, row=6)
    unit_options = ['Erlang', 'CCS']
    unit_menu = ttk.OptionMenu(parent_frame, unit_var, unit_options[0], *unit_options)
    unit_menu.grid(column=1, row=6)

    # Calculate Button
    calculate_btn_gos= ttk.Button(parent_frame, text="Calculate GoS", command=calc_gos)
    calculate_btn_gos.grid(column=0, row=7, columnspan=2)

    # Add button for comparing GoS methods here
    compare_gos_btn = ttk.Button(parent_frame, text="Compare GoS Methods", command=compare_gos_methods)
    compare_gos_btn.grid(column=0, row=9, columnspan=2, pady=10)

    # Outputs
    ttk.Label(parent_frame, text="Grade of Service (GoS):").grid(column=0, row=8)
    ttk.Label(parent_frame, textvariable=GoS_var).grid(column=1, row=8)


def setup_traffic_calculator_frame(parent_frame):
    # Setup GUI elements for the Traffic Calculator inside the parent frame
    ttk.Label(parent_frame, text="Traffic Intensity Calculator", style="Header.TLabel").grid(column=0, row=0, pady=10, columnspan=2)    # Inputs
    # Inputs
    ttk.Label(parent_frame, text="Number of Trunks (N):").grid(column=0, row=1)
    global N_entry_A
    N_entry_A = ttk.Entry(parent_frame)
    N_entry_A.grid(column=1, row=1)
```

```

    # Select GoS Method
    ttk.Label(parent_frame, text="GoS Calculation Method:").grid(column=0, row=2)
    method_options = ['Erlang B', 'Erlang C']
    method_menu = ttk.OptionMenu(parent_frame, method_var_A, method_options[0], *method_options)
    method_menu.grid(column=1, row=2)

    # Add GoS Input
    ttk.Label(parent_frame, text="Grade of Service (GoS):").grid(column=0, row=3)
    global GoS_entry
    GoS_entry = ttk.Entry(parent_frame)
    GoS_entry.grid(column=1, row=3)

    # Calculate Button
    calculate_btn_gos= ttk.Button(parent_frame, text="Calculate Traffic Intenisty (A)", command=calc_A)
    calculate_btn_gos.grid(column=0, row=4, columnspan=2)

    # Outputs
    ttk.Label(parent_frame, text="Traffic Intensity (A):").grid(column=0, row=5)
    ttk.Label(parent_frame, textvariable=traffic_intensity_var).grid(column=1, row=5)


    # Add a button to your GUI to trigger this function
    calculate_results_btn = ttk.Button(parent_frame, text="Calculate and Display Tables", command=calculate_and_display_results)
    calculate_results_btn.grid(column=0, row=6, columnspan=2, pady=5)

# Create frames
traffic_frame = tk.Frame(app, borderwidth=2, relief="groove")
gos_frame = tk.Frame(app, borderwidth=2, relief="groove")

# Layout frames vertically
traffic_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)
gos_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)

# Setup each calculator within its frame
setup_traffic_calculator_frame(traffic_frame)
setup_gos_calculator_frame(gos_frame)

app.mainloop()
```

Listing 1: Source Code

B GUI Screenshot

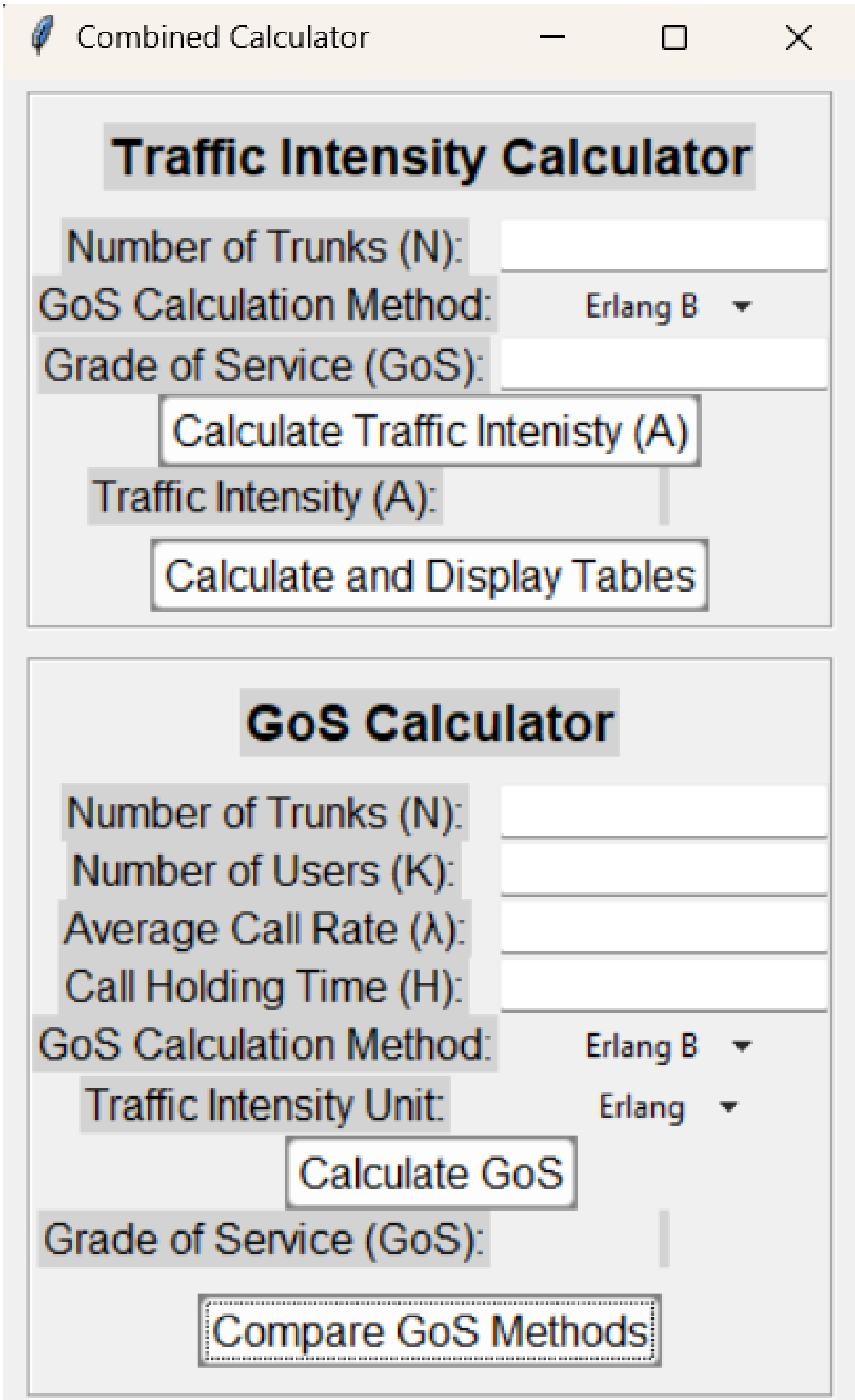


Figure 3: Caption for the figure.