Investigating the accuracy of the Erlang formulae

Paul Kielty
School of Engineering
National Univeristy of Ireland, Galway
Galway, Ireland
p.kielty3@nuigalway.ie

***Abstract*— This document details a study of the ability of the Erlang-B and Erlang-C formulae in estimating the Grade of Service (GOS) of a telecommunication link. This is accomplished by comparison to the GOS results of Monte Carlo simulations under the same network specification. Two simulation models were used, one where calls are blocked if all channels are busy, for comparison to Erlang-B, and a second where calls are requeued until a channel is available, for Erlang-C. It was found both Erlang formulae prone to error with a tendency to underestimate the GOS when compared to a Monte Carlo simulation operating under the same assumptions about the network.**

Keywords—Erlang, telecommunications, grade of service

I. INTRODUCTION

The aim of this study is to compare and analyse the estimated performance of telecommunications network for handling traffic, by Erlang calculations and Monte Carlo simulations. The network specification chosen for analysis is based on the Advanced Mobile Phone System (AMPS) in the United States [1]. This is a first generation (1G) trunked cellular system, meaning separate frequency channels handle each call, and the area covered by the network is subdivided into cells. A typical cell in this system contains 57 traffic channels [2]. AWPS was deployed in the United States over the 1980s, and according to [3], the mean call holding time, μ_{CHT}, for local calls in 1987 was ~0.0388 hours (2.33 minutes). This figure is used with the mean number of calls per hour, μ_{CPH}, to calculate the offered traffic, A_0, the network must handle in Erlangs, according to (1):

$$A_0 = \mu_{CHT} \times \mu_{CPH} \quad (1)$$

The μ_{CPH} is incremented to yield a range of values for A_0, the independent variable for the tests in this study.

A. Erlang B

The Erlang-B formula (2) estimates the probability that an individual call will be dropped, E_1, by a telecommunications network with a specified number of channels, n, for offered traffic A_0:

$$E_1(n, A_0) = \frac{\frac{{A_0}^n}{n!}}{\sum_{j=0}^{n} \frac{{A_0}^j}{j!}} \quad (2)$$

The probability of blocking E_1, when converted to a percentage, is known as the GOS. This formula makes a number of assumptions about the system:

- The calls arrival times follow a Poisson distribution.
- The call holding times follow a negative exponential distribution.
- A call is blocked if all channels are already occupied on its arrival.

II. ERLANG C

The Erlang-C formula (3) builds on the Erlang-B for systems where call requeuing is implemented.

$$P_{delay}(n, A_0) = \frac{\frac{{A_0}^n}{n!}\frac{n}{n - A_0}}{\left(\sum_{j=0}^{n} \frac{{A_0}^j}{j!}\right) + \frac{{A_0}^n}{n!}\frac{n}{n - A_0}} \quad (3)$$

In such systems, instead of calls being blocked when all channels are busy, they are instead placed in a queue to be served at the next channel opening. The probability of a call being delayed as a result of being placed on a queue, P_{delay}, is the new result. The GOS of a network with queueing is given by P_{delay} as a percentage. This formula makes the following assumptions:

- The calls arrival times follow a Poisson distribution.
- The call holding times follow a negative exponential distribution.
- A call is added to a queue if all channels are already occupied on its arrival.
- A call waits in the queue indefinitely for a channel (no caller abandonment).
- The queue operates by First In, First Out (FIFO), and the moment a call finishes, the free channel is given a new call from the queue (if any are present).

III. MONTE CARLO SIMULATION MODEL

A. Call Simulation Specification

In the simulations, a call is simply composed of a start time and an end time. A call's start time is generated by randomly sampling a uniform distribution with a range of 0 to 1. This upper limit of 1 representing 1 hour. The sampling is repeated μ_{CPH} times to obtain a random start time for each call. The start times are then sorted such that they are served to the network in order of arrival. Next, each call is assigned an end time by adding a random holding time to the each start time. An exponential distribution has been shown to be suitable for 1G and second generation (2G) network modelling [4], so an exponential distribution with a scale of μ_{CHT} is sampled to obtain these holding times. Each simulation is repeated 1000 times, from which the mean and standard deviation is calculated.

B. Simulation A – No queue

The simulation steps through the calls in order of arrival, checking at each call start time if there are any channels free. If there are no free channels, the call is blocked, and the next start time is considered. If a free channel is found, the call finish time is saved to the channel. This channel is considered busy until this time has passed. Simulation A is run over the range $1 \leq A_0 \leq 100$ Erlangs. These values are calculated according to (1) with a μ_{CHT} of 0.03883 hours, and an increasing μ_{CPH} that yields the desired A_0 range of values. The actual resulting A_0 range has a small offset as the μ_{CPH} is limited to an integer for simulations, but this has been accounted for in the results and plots.

C. Simulation B – With Queue

In the variant designed with requeuing, instead of completely stopping the call if all channels are busy, it is placed on a queue. After a chosen delay, d_{queue}, the channels are all checked again for any openings, any of which are then filled with the queued calls according to FIFO. Simulations were run with a d_{queue} of 0 to mirror Erlang-C (channels immediately filled from queue when free), then again with a more realistic d_{queue} of 6s as used by [5]. Erlang-C assumes no call abandonment and blocks no calls, and as a result the calculation only holds if $A_0 < n$. This is because for $A_0 \geq n$, calls arrive faster than they can be processed, and the queue builds up indefinitely. For this reason, the GOS from Erlang-C and simulation B are obtained over the range $1 \leq A_0 \leq n\text{-}1$. This A_0 range is calculated with the same μ_{CHT} of 0.03883 hours. In each iteration of simulation B, a "warmup" simulation of one hour is run before the delayed calls are

counted. This is to remove start up biases that would result from the queue and channels all being free. No statistics are recorded until this warmup hour completes.

D. Simulation Running Details

The computer used for simulation has an Intel Core i5-4690K quad core CPU overclocked to a base frequency of 3.8GHz, 16GB DDR3 memory, and runs 64-bit Windows 10. The simulations were created and run in Python 3.8.0 using a Jupyter notebook. To run to completion, simulations A and B took 8.02 and 28.56 minutes respectively.

IV. Results

A. Erlang-B & Simulation A

The GOS by Erlang-B and the mean GOS from 1000 iterations of simulation A were plotted against the offered traffic in fig. 1.

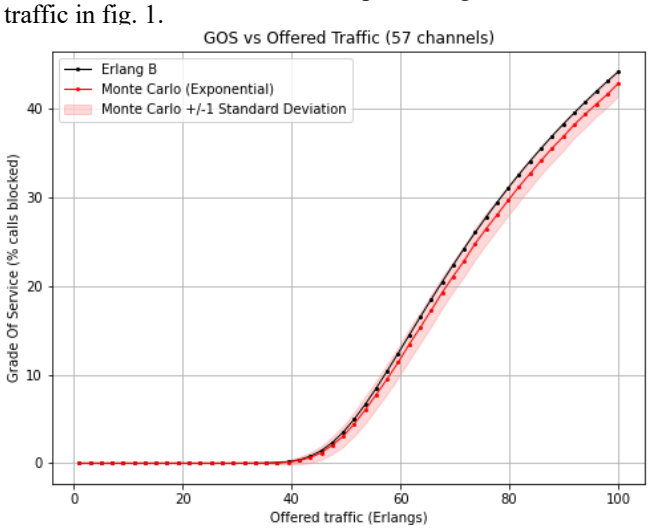

Fig. 1. GOS vs Offered Traffic by Erlang-B and simulation A.

Erlang-B estimations of the simulated GOS remained within one standard deviation of the simulated results over the full range of traffic values. Erlang-B tended to underestimate the real GOS, demonstrated by the mean error (calculated – simulated) and mean absolute error being equal at 0.645%. This underestimate may make Erlang-B more appealing for designing a system than if it were to overestimate, as it creates a safety net for maintaining desired GOS. The error plotted in fig. 2 shows how the accuracy worsens as the traffic increases, with the rate of change of the error peaking over 57 Erlangs (equal to n channels). With a GOS at this point of ~10%, it is expected that most applications would aim for lower, where we see the Erlang-B estimate is more accurate.

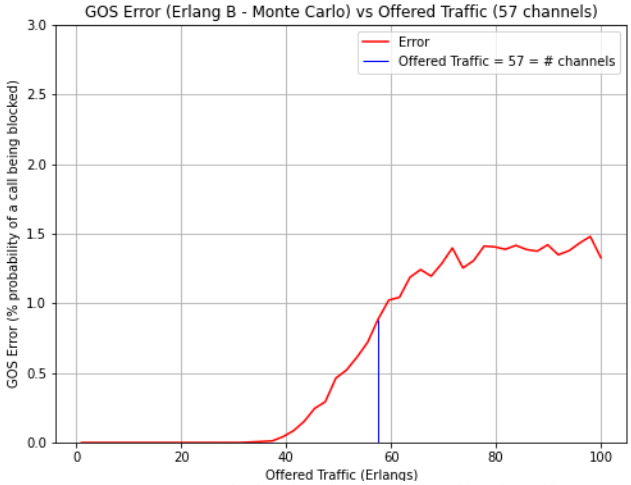

Fig. 2. GOS Error (Calculated – Simulated) vs Offered Traffic

B. Erlang-B, Erlang-C, & Simulation B

The results from simulation B were plotted with the calculated GOS by both Erlang-B and Erlang-C in fig. 3. As expected, the GOS rises rapidly as A_0 tends towards n. Similar to the Erlang-B results above, when comparing Erlang-C to the simulation with $d_{queue} = 0$, we get a mean error (calculated – simulated) equal to the mean absolute error at 0.341%. Again indicating that the calculated is underestimating performance, however this is under Erlang-C's very strict assumptions. When compared to the more realistic simulation with $d_{queue} = 6s$, a larger mean absolute error of 0.401% was calculated. Again the error increased rapidly for larger A_0, but mostly effecting quite poor GOS' of > 10%, likely undesirable when designing a system.

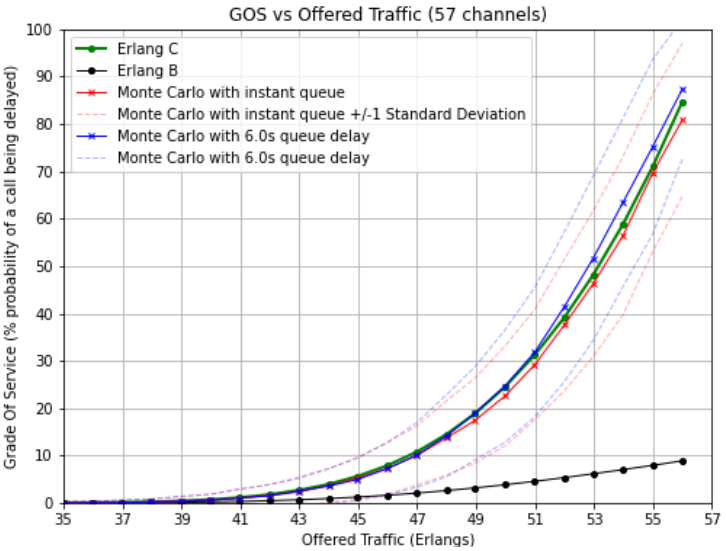

Fig. 3. GOS vs Offered Traffic by Erlang formulae and simulation B.

V. Conclusions

From these results we learn that Erlang-B is prone to being pessimistic when estimating the GOS of a trunked 1G systems with no queuing. However, the error may be suitably low, particularly when $A_0 < n$. Erlang-C makes more assumptions about the system, notably with no caller abandonment and instant handover. When simulating under these assumptions it is similarly pessimistic when estimating the simulated GOS, but a more realistic simulation with a call requeue delay finds it can be prone to error in either direction. Both equations are significantly more accurate when looking over smaller A_0 values, especially when aiming for a GOS $\leq$ 1%, which is quite typical in industry. As such they could still be used in suitable applications if there is enough room for error, but Monte Carlo simulations are a more desirable approach.

VI. References

[1] W. R. Young, "Advanced Mobile Phone Service: Introduction, Background, and Objectives," *Bell Syst. Tech. J.*, vol. 58, 1979.

[2] Y. Zhang, "The Cellular Concept - System Design Fundamentals," 2005.

[3] CTIA-The Wireless Association, "Background on CTIA's Semi-Annual Wireless Industry Survey," 2013. Accessed: Dec. 29, 2020. [Online]. Available: http://www.ctia.org/store/producttyperesults.cfm?group_id=1.

[4] M. Alwakeel, "Deriving Call Holding Time Distribution in Cellular Network from Empirical Data," 2009.

[5] C. P. Liu and A. T. Wang, "Performance Measures on a Customer Retrial Calls in Telephone Networks by Using Monte Carlo Simulation," 2011.

VII. Code

In [ ]:

```
from math import factorial, exp
import matplotlib.pyplot as plt
import numpy as np
import csv
import time

def offeredTraffic(calls_per_hour, hours_per_call): # A0
    return calls_per_hour*hours_per_call

def erlangB(n, A0):
    denom = 0
    for i in range(n+1):
        denom += (A0**i)/(factorial(i))
    E1 = ((A0**n)/factorial(n))/denom
    return E1

def erlangC(n, A0):
    denomSum = 0
    for i in range(n):
        denomSum += (A0**i)/factorial(i)
    denom = A0**n + factorial(n)*(1-A0/n)*denomSum
    return (A0**n)/denom

def meanCallDelay(n, A0, meanCallDuration, probDelay):
    return probDelay*meanCallDuration/(n-A0)

def meanQueuedDelay(n, A0, meanCallDuration):
    return meanCallDuration/(n-A0)

def erlangC_allowedDelayGOS(n, A0, meanCallDuration, allowedDelay=0):
    probDelay = erlangC(n,A0)
    return probDelay*exp(-(n-A0)*allowedDelay/meanCallDuration)
```

(bigger code cells sometimes start on new pages, see next page)

In [ ]:

```
def trafficSimulation(numChannels, numCalls, meanCallDuration, requeue=False, recheckChannelsDelay=0, allowedDelay=0, holdingDistrib='exp', repetitions=1, warmUpHours=None):

    callFailRates = np.zeros(repetitions)
    meanCallDelay = np.zeros(repetitions)

    for i in range(repetitions):
        time = 0
        channelFreeTimes = np.zeros(numChannels)

        if warmUpHours != None:
            # WARMUP SIMULATION
            # Get call durations
            if holdingDistrib == 'exp':
                callDurations = np.random.exponential(scale=meanCallDuration, size=int(numCalls*warmUpHours))

            elif holdingDistrib == 'gamma':
                callDurations = np.random.gamma(shape=1.2073, scale=0.03205, size=int(numCalls*warmUpHours)) # shape & scale by [4]: 0.03205 = muCHT*(mu/b) = muCHT*(29.6121/35.875)
            # Get call start times
            callStarts = np.random.uniform(size=int(numCalls*warmUpHours))
            callStarts.sort()
            calls = np.stack((callStarts, np.add(callStarts,callDurations)),axis=1) # Calls = Numpy array of [[call0_start call0_end]; [call1_start call1_end];...

            # Run warmup simulation

            callsFailed = 0
            totalDelay = 0
            for call in calls:
                time = call[0]
                channelFound = False

                for j in range(numChannels):
                    if not(channelFound):
                        if channelFreeTimes[j] <= time:
                            channelFreeTimes[j] = call[1]
                            channelFound = True

                if channelFound == False:
                    if requeue == False:
                        callsFailed += 1
                    elif recheckChannelsDelay > 0:
                        callsFailed += 1
                        while channelFound == False:
                            time += recheckChannelsDelay
                            totalDelay += recheckChannelsDelay
                            call[1] += recheckChannelsDelay # move forward call end time

                            for j in range(numChannels):
                                if not(channelFound): # Stop checking channels as soon as a free one is found
                                    if channelFreeTimes[j] <= time:
                                        channelFreeTimes[j] = call[1]
                                        channelFound = True

                    else:
```

```
                        nextCallFinishTime = channelFreeTimes.min()
                        totalDelay += nextCallFinishTime-call[0]
                        call[1] += nextCallFinishTime-call[0]
                        if nextCallFinishTime-call[0]>allowedDelay: # only count call a
s failed if above the allowed wait time (if allowance present)
                            callsFailed += 1
                        indices = np.where(channelFreeTimes == nextCallFinishTime)
                        index = indices[0][0]
                        channelFreeTimes[index] = call[1]

        # MEASURED SIMULATION
        # Get call durations
        if holdingDistrib == 'exp':
            callDurations = np.random.exponential(scale=meanCallDuration, size=numCalls
)

        elif holdingDistrib == 'gamma':
            callDurations = np.random.gamma(shape=1.2073, scale=0.03205, size=numCalls)

        # Get call start times
        callStarts = np.random.uniform(size=numCalls)
        callStarts.sort()
        calls = np.stack((callStarts, np.add(callStarts,callDurations)),axis=1)

        # Run measured simulation
        if warmUpHours != None:
            calls += warmUpHours
            time = warmUpHours

        callsFailed = 0
        totalDelay = 0
        for call in calls:
            time = call[0]
            channelFound = False

            for j in range(numChannels):
                if not(channelFound):
                    if channelFreeTimes[j] <= time:
                        channelFreeTimes[j] = call[1]
                        channelFound = True

            if channelFound == False:
                if requeue == False:
                    callsFailed += 1

                elif recheckChannelsDelay > 0:
                    callsFailed += 1
                    while channelFound == False:
                        time += recheckChannelsDelay
                        totalDelay += recheckChannelsDelay

                        call[1] += recheckChannelsDelay # move forward call end time
                        for j in range(numChannels):
                            if not(channelFound): # Stop checking channels as soon as a
free one is found
                                if channelFreeTimes[j] <= time:
                                    channelFreeTimes[j] = call[1]
                                    channelFound = True

                else:
                    nextCallFinishTime = channelFreeTimes.min()
```

```
                    totalDelay += nextCallFinishTime-call[0]
                    call[1] += nextCallFinishTime-call[0]
                    if nextCallFinishTime-call[0]>allowedDelay: # only count call as fa
iled if above the allowed wait time (if allowance present)
                        callsFailed += 1
                    indices = np.where(channelFreeTimes == nextCallFinishTime)
                    index = indices[0][0]
                    channelFreeTimes[index] = call[1]

        callFailRates[i] = callsFailed/numCalls
        meanCallDelay[i] = totalDelay/numCalls

    if requeue == False:
        return callFailRates.mean(), np.std(callFailRates, ddof=1)
    else:
        return callFailRates.mean(), np.std(callFailRates, ddof=1), meanCallDelay.mean
(), np.std(meanCallDelay, ddof=1)
```

In [ ]:

```
meanCallDuration = 2.33/60 # 2.33 mins in hrs
n = 57 # number of channels
```

In [ ]:

```
# CALLS BLOCKED - NO REQUEUING
startTime = time.time()

simulationCallAmounts = (np.linspace(1, 75, 15)/meanCallDuration).astype(np.int)
A0s = offeredTraffic(simulationCallAmounts, meanCallDuration)
repetitionsPerSimulation = 100

# Initialise arrays for results
erlangBs = np.zeros(simulationCallAmounts.shape)
GOS_exp = np.zeros(simulationCallAmounts.shape)
stdDevs_exp = np.zeros(simulationCallAmounts.shape)
# GOS_gamma = np.zeros(simulationCallAmounts.shape)
# stdDevs_gamma = np.zeros(simulationCallAmounts.shape)

# Run simulations
print("# Calls:\tTraffic:\tErlang B:\t\tSim GOS (exp):\t\tstdDev (exp):")
for i, numCalls in enumerate(simulationCallAmounts):
    A0 = A0s[i]
    erlangBs[i] = erlangB(n, A0)*100

    meanBlockingRate_exp, stdDev_exp = trafficSimulation(n, numCalls, meanCallDuration,
repetitions=repetitionsPerSimulation)
    GOS_exp[i] = meanBlockingRate_exp*100
    stdDevs_exp[i] = stdDev_exp*100

#     meanBlockingRate_gamma, stdDev_gamma = trafficSimulation(n, numCalls, meanCallDur
ation, holdingDistrib='gamma', repetitions=repetitionsPerSimulation)
#     GOS_gamma[i] = meanBlockingRate_gamma*100
#     stdDevs_gamma[i] = stdDev_gamma*100
    print("{}\t\t{:.4f}\t\t{:.4f}\t\t\t{:.4f}\t\t\t{:.4f}".format(numCalls, A0, erlangB
s[i], GOS_exp[i], stdDevs_exp[i]))

# PLOT
plt.figure(figsize=(8,6))
markerSize=2
lineWidth=1
plt.plot(A0s, erlangBs, label='Erlang B', color="black", marker='o', markersize=markerS
ize, linewidth=lineWidth)
plt.plot(A0s, GOS_exp, label="Monte Carlo (Exponential)", color="red", marker='o', mark
ersize=markerSize, linewidth=lineWidth)
plt.fill_between(A0s, GOS_exp+stdDevs_exp, GOS_exp-stdDevs_exp, interpolate=True, color
="red", alpha=0.15, label="Monte Carlo +/-1 Standard Deviation")
# plt.plot(A0s, GOS_gamma, label="Monte Carlo (Gamma)", color="green", marker='o', mark
ersize=markerSize, linewidth=lineWidth)
# plt.fill_between(A0s, GOS_gamma+stdDevs_gamma, GOS_gamma-stdDevs_gamma, interpolate=T
rue, color="green", alpha=0.15)

plt.xlabel('Offered traffic (Erlangs)')
plt.ylabel('Grade Of Service (% calls blocked)')
plt.title("GOS vs Offered Traffic ({} channels)".format(n))
plt.legend()
plt.grid()
plt.savefig("sim1_1.png")
plt.show()

timeTaken = time.time()-startTime
print(timeTaken)
```

In [ ]:

```
# ERROR
error = erlangBs-GOS_exp

plt.figure(figsize=(8,6))
plt.plot(A0s, error, label="Error", color='red')
# plt.vlines(x = A0s[n//2], ymin = 0, ymax = error[n//2], color = 'blue', label = 'Offe
red Traffic = 57 = # channels', linewidth=1)

plt.xlabel('Offered Traffic (Erlangs)')
plt.ylabel('GOS Error (% probability of a call being blocked)')
plt.title("GOS Error (Erlang B - Monte Carlo) vs Offered Traffic ({} channels)".format(
n))
plt.ylim([0,2])
plt.legend()
plt.grid()
plt.savefig('sim1_2.png')
plt.show()

plt.figure(figsize=(8,6))
plt.scatter(erlangBs, GOS_exp, 1, vmin=0, vmax=1)
plt.xlabel('Simulated GOS')
plt.ylabel('Calculated GOS (Erlang-B)')
plt.plot([0,50],[0,50])
plt.show()
```

In [ ]:

```
# Save to csv
with open('block.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    writer.writerow(["# Calls", "Offered Traffic", "Erlang B:", "Sim GOS (exponentia
l):", "Sim std dev (exponential):", "Repetitions per simulation = "+str(repetitionsPerS
imulation)])
    # (numCalls, A0, erlangBs[i], GOS_exp[i], stdDevs_exp[i],
    for i, numCalls in enumerate(simulationCallAmounts):
        writer.writerow([numCalls, A0s[i], erlangBs[i], GOS_exp[i], stdDevs_exp[i]])
```

In [ ]:

```
# WITH REQUEUING
startTime = time.time()

simulationCallAmounts = (np.linspace(1, n-1, n-1)/meanCallDuration).astype(np.int)
A0s = offeredTraffic(simulationCallAmounts, meanCallDuration)
repetitionsPerSimulation = 100

# Initialise arrays for results
erlangBs = np.zeros(simulationCallAmounts.shape)
erlangCs = np.zeros(simulationCallAmounts.shape)
meanCallDelays_calc = np.zeros(simulationCallAmounts.shape)

qGOS_exp = np.zeros(simulationCallAmounts.shape)
qStdDevs_exp = np.zeros(simulationCallAmounts.shape)
meanCallDelays_exp = np.zeros(simulationCallAmounts.shape)
callDelayStdDevs_exp = np.zeros(simulationCallAmounts.shape)

qGOS_exp_delayed = np.zeros(simulationCallAmounts.shape)
qStdDevs_exp_delayed = np.zeros(simulationCallAmounts.shape)
meanCallDelays_exp_delayed = np.zeros(simulationCallAmounts.shape)
callDelayStdDevs_exp_delayed = np.zeros(simulationCallAmounts.shape)

# GOS_gamma = np.zeros(simulationCallAmounts.shape)
# stdDevs_gamma = np.zeros(simulationCallAmounts.shape)
# meanCallDelays_gamma = np.zeros(simulationCallAmounts.shape)
# callDelayStdDevs_gamma = np.zeros(simulationCallAmounts.shape)

allowedDelay = 0 # (in hours)
recheckDelay = 0.1/60 # (in hours)

# Run simulations
print("# Calls:\tTraffic:\tErlang C:\t\tSim GOS (exp):\t\tSim GOS stdDev\t\tCalc. dela
y/call\tSim Mean delay/call\tdelay/call stdDev")
for i, numCalls in enumerate(simulationCallAmounts):

    A0 = A0s[i]
    erlangBs[i] = erlangB(n, A0)*100
    erlangCs[i] = erlangC_allowedDelayGOS(n, A0, meanCallDuration, allowedDelay=allowed
Delay)*100
    meanCallDelays_calc[i] = meanCallDelay(n, A0, meanCallDuration, np.clip(erlangCs/10
0,0,1)[i])

    meanDelayRate_exp, stdDev_exp,  meanCallDelay_exp, callDelayStdDev_exp = trafficSim
ulation(n, numCalls, meanCallDuration, requeue=True, recheckChannelsDelay=0, repetition
s=repetitionsPerSimulation, warmUpHours=1)
    qGOS_exp[i] = meanDelayRate_exp*100
    qStdDevs_exp[i] = stdDev_exp*100
    meanCallDelays_exp[i] = meanCallDelay_exp
    callDelayStdDevs_exp[i] = callDelayStdDev_exp

    meanDelayRate_exp_delayed, stdDev_exp_delayed,  meanCallDelay_exp_delayed, callDela
yStdDev_exp_delayed = trafficSimulation(n, numCalls, meanCallDuration, requeue=True, re
checkChannelsDelay=recheckDelay, repetitions=repetitionsPerSimulation, warmUpHours=1)
    qGOS_exp_delayed[i] = meanDelayRate_exp_delayed*100
    qStdDevs_exp_delayed[i] = stdDev_exp_delayed*100
    meanCallDelays_exp_delayed[i] = meanCallDelay_exp_delayed
    callDelayStdDevs_exp_delayed[i] = callDelayStdDev_exp_delayed

    # meanDelayRate_gamma, stdDev_gamma,  meanCallDelay_gamma, callDelayStdDev_gamma =
```

```
 trafficSimulation(n, numCalls, meanCallDuration, requeue=True, holdingDistrib='gamma r
echeckChannelsDelay=0, repetitions=repetitionsPerSimulation, warmUpHours=1)
    # GOS_gamma[i] = meanDelayRate_gamma*100
    # stdDevs_gamma[i] = stdDev_gamma*100
    # meanCallDelays_gamma[i] = meanCallDelay_gamma
    # callDelayStdDevs_gamma[i] = callDelayStdDev_gamma

    print("{}\t\t{:.4f}\t\t\t{:.3f}\t\t\t{:.3f}\t\t\t{:.4f}\t\t\t{:.4f}\t\t\t{:.4f}\t\t
\t{:.4f}".format(numCalls, A0, erlangCs[i], qGOS_exp[i], qStdDevs_exp[i], meanCallDelay
s_calc[i], meanCallDelays_exp[i], callDelayStdDevs_exp[i]))

# PLOT RESULTS
plt.figure(figsize=(8,6))
markerSize = 3
lineWidth = 1
plt.plot(A0s, erlangCs, label='Erlang C', color="orange", marker='o', markersize=marker
Size, linewidth=lineWidth)
plt.plot(A0s, erlangBs, label='Erlang B', color="black", marker='o', markersize=markerS
ize, linewidth=lineWidth)
plt.plot(A0s, qGOS_exp, label="Monte Carlo with instant queue", color="red", marker='x'
, markersize=markerSize, linewidth=lineWidth)
plt.fill_between(A0s, qGOS_exp+qStdDevs_exp,  qGOS_exp-qStdDevs_exp, interpolate=True,
color="red", alpha=0.15, label="Monte Carlo with instant queue +/-1 Standard Deviation"
)
# plt.errorbar(A0s, qGOS_exp, yerr=qStdDevs_exp, label="Monte Carlo with requeue (Expon
ential)", color="red")
plt.plot(A0s, qGOS_exp_delayed, label="Monte Carlo with {}s queue delay".format(recheck
Delay*3600), color="blue", marker='x', markersize=markerSize, linewidth=lineWidth)
plt.fill_between(A0s, qGOS_exp_delayed+qStdDevs_exp_delayed,  qGOS_exp_delayed-qStdDevs
_exp_delayed, interpolate=True, color="blue", alpha=0.15, label="Monte Carlo with {}s q
ueue delay +/-1 Standard Deviation".format(recheckDelay*3600))
# plt.errorbar(A0s, GOS_gamma, yerr=stdDevs_gamma, label="Monte Carlo with requeue (Gam
ma)", color="green")
# plt.fill_between(A0s, GOS_gamma+stdDevs_gamma, GOS_gamma-stdDevs_gamma, interpolate=T
rue, color="green", alpha=0.15)
# plt.vlines(x = n, ymin = 0, ymax = 100, color = 'blue', label = 'Offered Traffic = n
 channels')
plt.xlim([30,58])
plt.ylim([0,100])
plt.xlabel('Offered Traffic (Erlangs)')
plt.ylabel('Grade Of Service (% probability of a call being delayed)')
# plt.ylabel('Grade Of Service (% probability of a call being delayed for > {} sec)'.fo
rmat(allowedDelay*3600))
plt.title("GOS vs Offered Traffic ({} channels)".format(n))
plt.legend()
plt.savefig("sim2_1.png")
plt.yticks(np.linspace(0,100,11))
plt.grid()
plt.show()

timeTaken = time.time()-startTime
print(timeTaken)
```

In [ ]:

```
# ERROR
qError = erlangCs-qGOS_exp
qAbsError = np.abs(qError)
plt.figure(figsize=(8,5))
plt.plot(A0s, qAbsError)
plt.xlabel('Offered Traffic (Erlangs)')
plt.ylabel('GOS Absolute Error (% probability of a call being delayed)')
plt.title("GOS Absolute Error vs Offered Traffic ({} channels)".format(n))
plt.show()
print(qAbsError.mean())
```

In [ ]:

```
# DELAY DURATION PLOTS
plt.figure(figsize=(12,6))
plt.plot(A0s, meanCallDelays_calc*3600, label='Erlang C Calculated Delay', color="orang
e")
plt.plot(A0s, meanCallDelays_exp*3600, label="Monte Carlo with instant queue", color="r
ed")
plt.fill_between(A0s, meanCallDelays_exp*3600+callDelayStdDevs_exp*3600, meanCallDelays
_exp*3600-callDelayStdDevs_exp*3600, interpolate=True, color="red", alpha=0.15)
plt.plot(A0s, meanCallDelays_exp_delayed*3600, label="Monte Carlo with {}s queue delay"
.format(recheckDelay*3600), color="blue")
plt.fill_between(A0s, meanCallDelays_exp_delayed*3600+callDelayStdDevs_exp_delayed*3600
, meanCallDelays_exp_delayed*3600-callDelayStdDevs_exp_delayed*3600, interpolate=True,
color="blue", alpha=0.15)
# plt.xlim([30,60])
plt.xlabel('Offered Traffic (Erlangs)')
plt.ylabel('Mean Delay/Call (seconds)')
plt.title("Mean Delay per Call vs Offered Traffic ({} channels)".format(n))
plt.legend()
plt.show()
```