CS3205 Computer Networks Lab Report

(Jan - May 2021 - Prof. Siva Ram Murthy)

Assignment 2: TCP Congestion Control

Name: **Rushabh Nirdosh Lalwani**

Roll No: **CS18B046**

1. Aim/Objective

The objective of this project is to emulate the TCP congestion control algorithm.

2. Introduction

We are currently executing TCP protocol in the Transport Layer. It uses segments to transfer data. These segments are transferred in a window or buffer containing multiple segments which are currently in progress. The receiver window is usually fixed while the sender window size (or Congestion window size) can vary based on the network requirements, network failures, network traffic etc. It is very difficult to use the best possible values of congestion window size as determining the traffic and acknowledgements is difficult.

We use an iterative algorithm to determine the CW size called **AIMD** (Additive Increase Multiplicative Decrease). This algorithm maintains a threshold value for the CW size and keeps decreasing it as timeout occurs. If the CW size is more than threshold then we linearly increase the CW size and if it is less than threshold then we exponentially grow it.

In this way the CW size is modified corresponding to the network traffic and network failure.

3. Experimental details

3.1 Experimental/Simulation setup

The receiver window size is fixed to 1 MB while the sender's MSS (Maximum segment size) is fixed to 1KB. Our experiment involves modification of CW size at each of the following instances using the following parameters:

a) Initialization: CWnew = **Ki** $*$ MSS
b) Timeout situation: CWnew = max(1, **Kf** * CWold)
c) Exponential growth: CWnew = min(CWold + **Km** $*$ MSS, RWS)

d) Linear growth: CWnew = min(CWold + **KI** ∗ MSS ∗ MSS / CWold, RWS)

We use the Ps value to determine whether timeout has occurred or not explained below.

3.2 Entities involved and functions in each entity

We have a getTimeout() function which reports whether the given instance produces a timeout or not. For that we have used the rand() function and called it at each iteration of the CW size update.

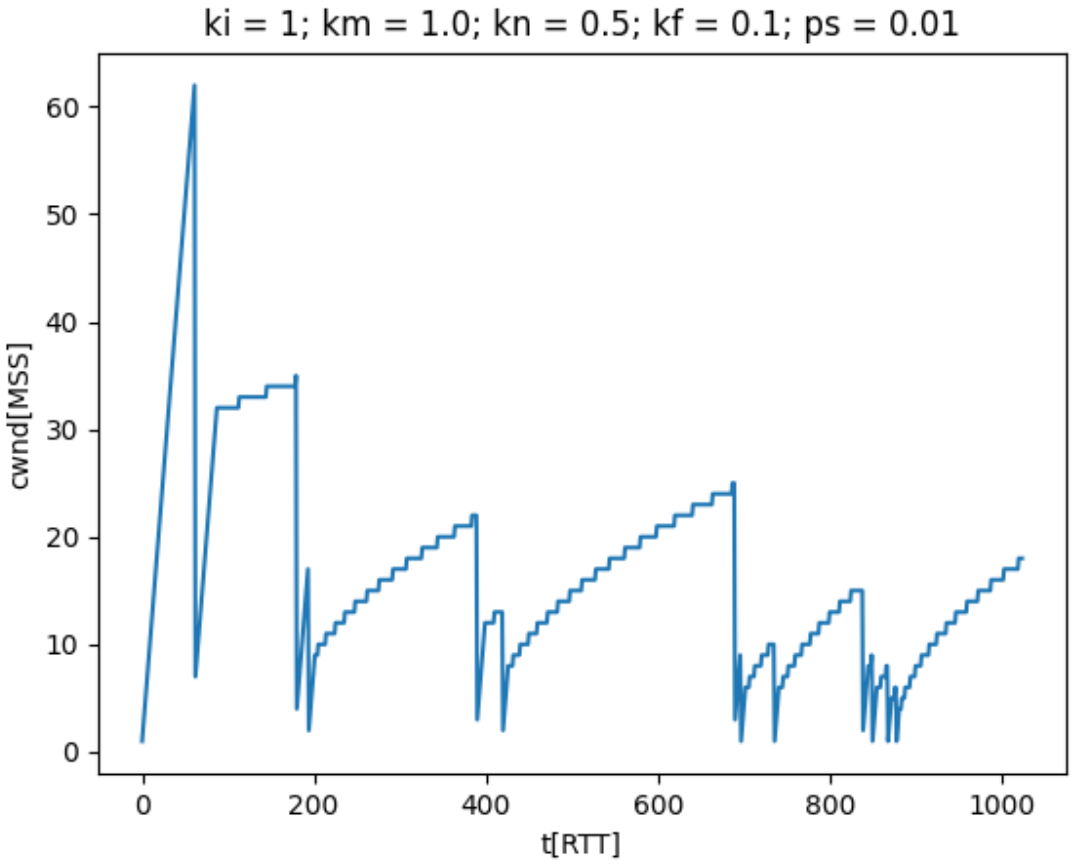
We have maintained some global definitions for MSS (Maximum Segment Size == 1) and RWS (Receiver Window Size == 1024). Apart from our usual 5 parameters to progress the algorithm for some iterations, we have declared the <T> value to be 1024. This means the algorithm will proceed to update the WS size for exactly T times.

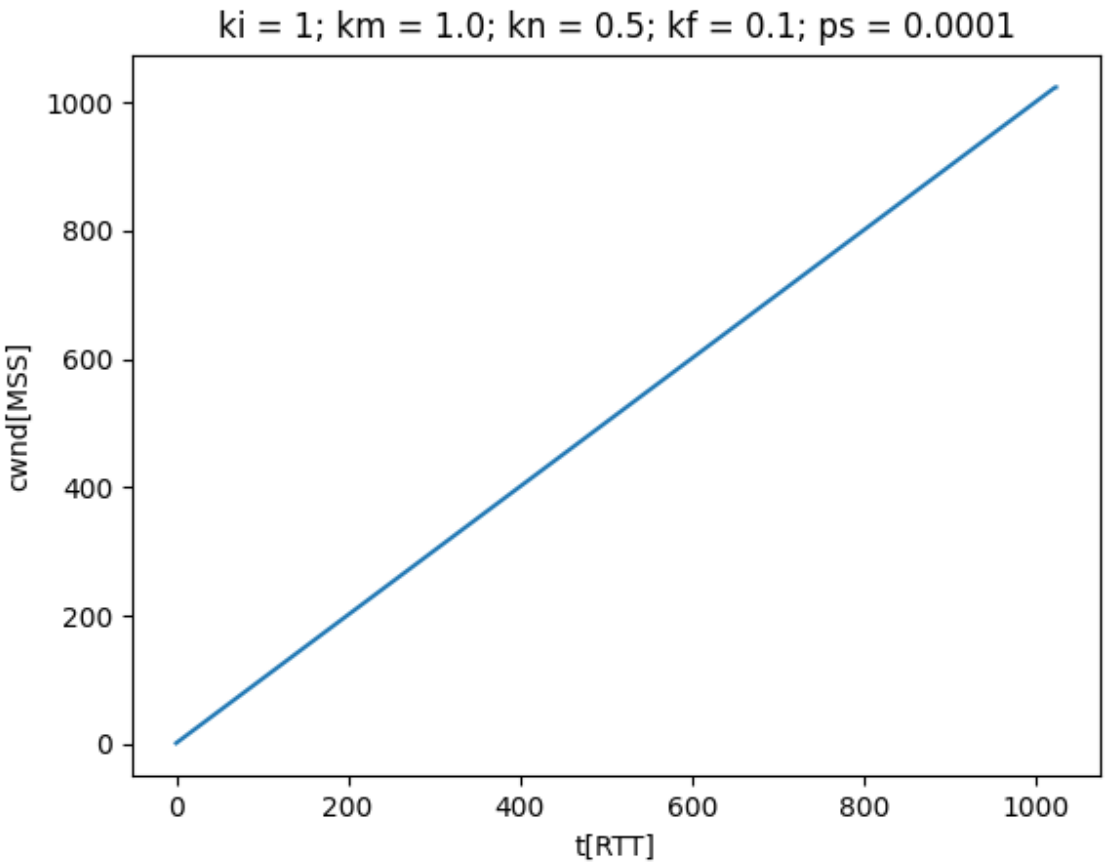
3.3 Additional details

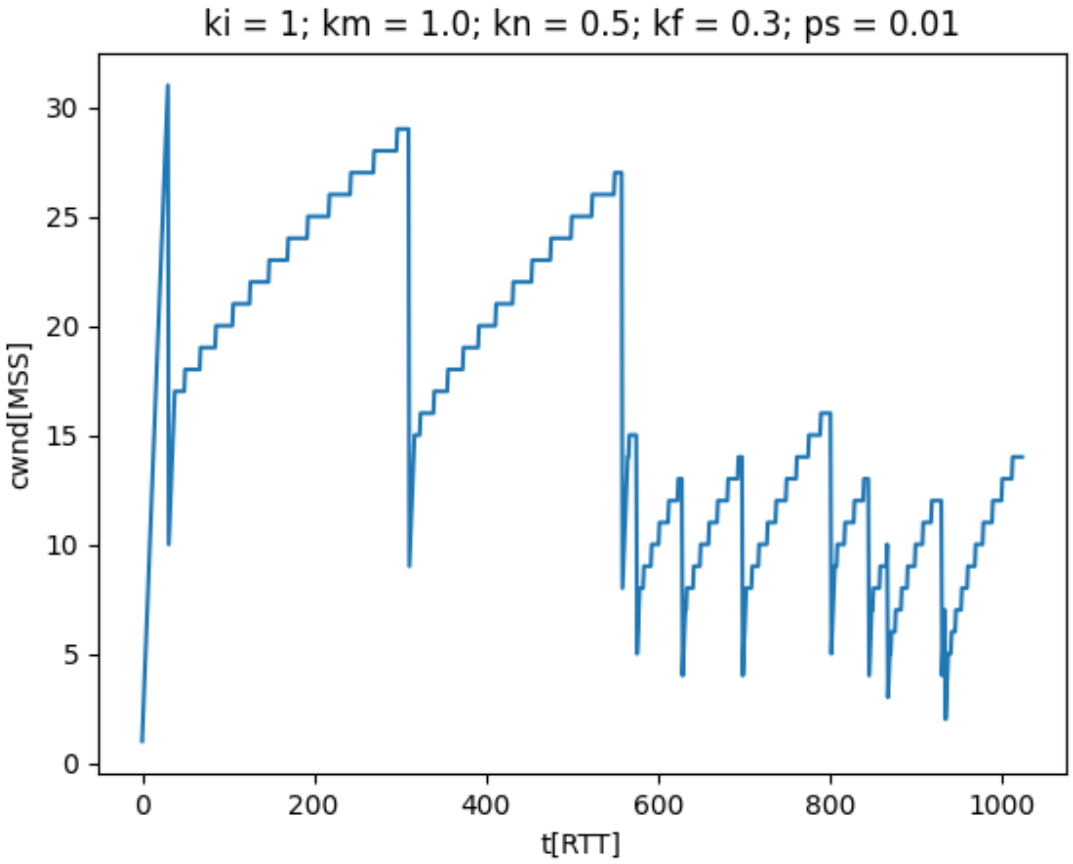
We maintained a bash script to pass these 5 parameters for 32 times to the .cpp file (cw.cpp). We also plot them using a python script plot.py after successful invocation of .cpp. This python script uses matplotlib.pyplot to plot the output file consisting of CW sze values.

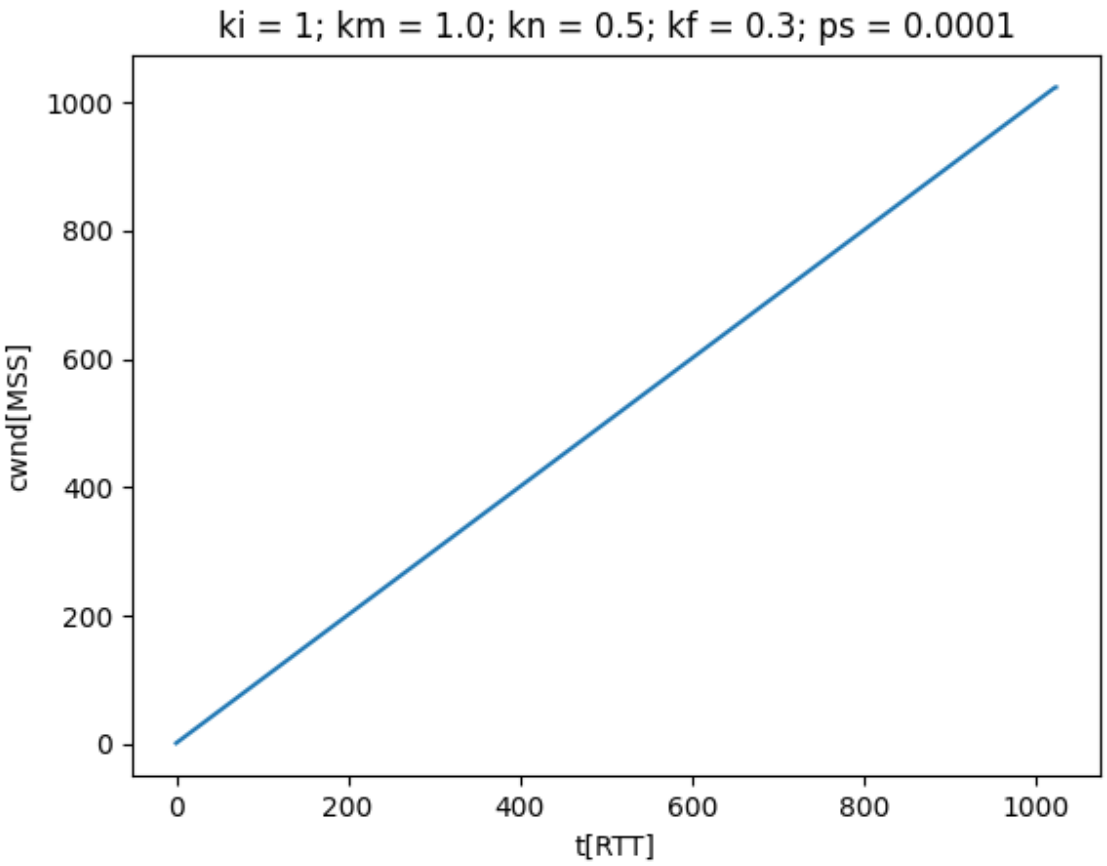
4. Results and Observations

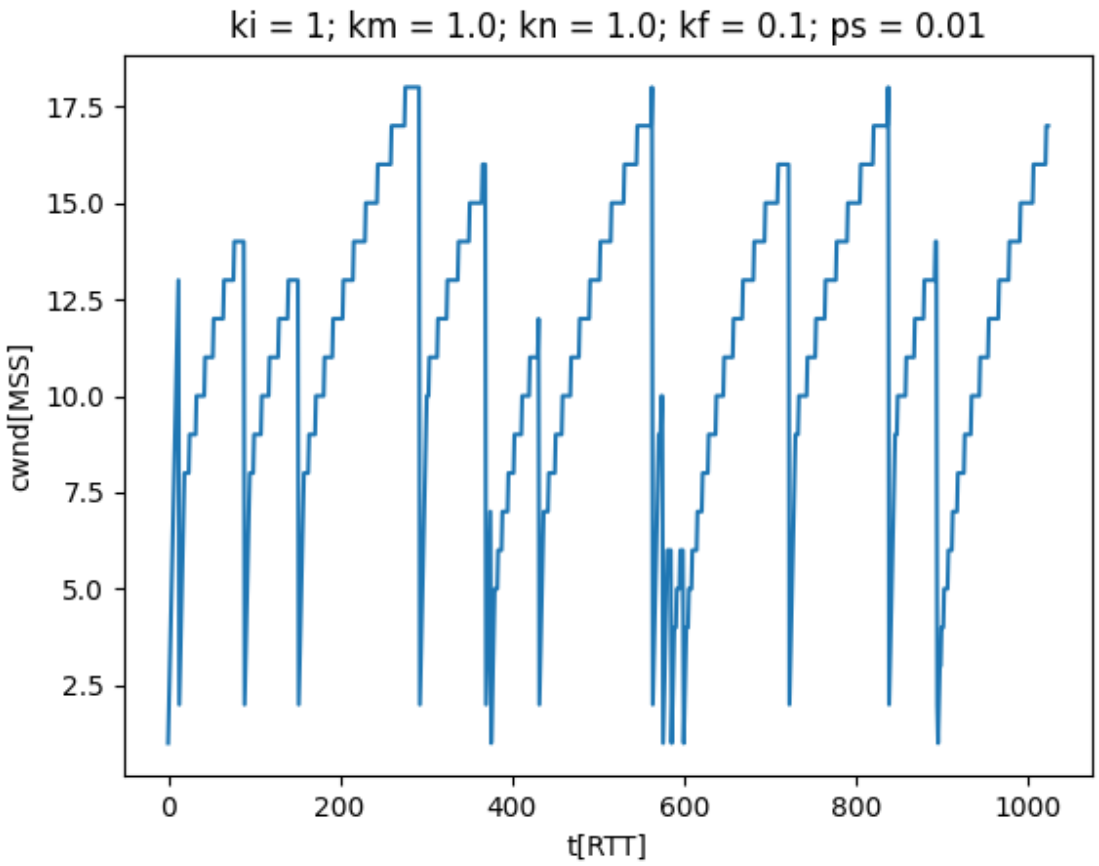
For 2^5 = 32 sets of input data we have got the following 32 plots. Please refer to README for running the program and getting similar outputs.

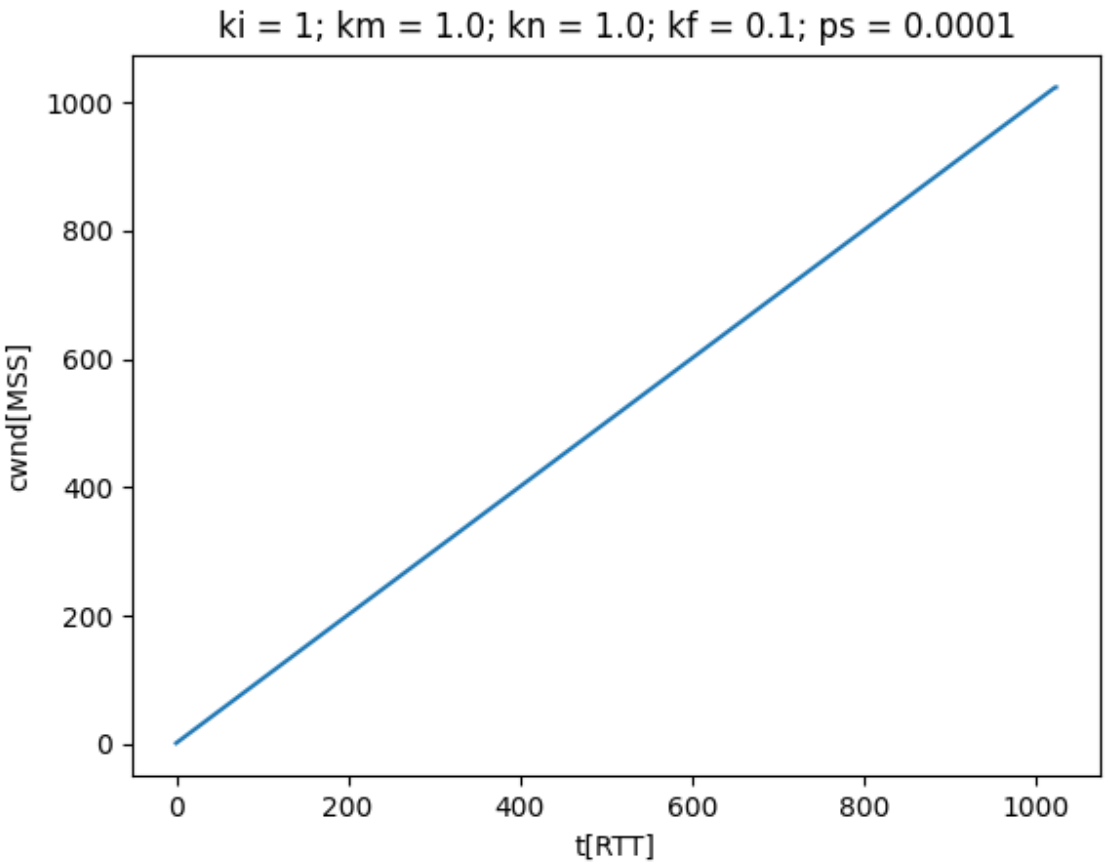

ki = 1; km = 1.0; kn = 0.5; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
10
20
30
40
50
60
0
200
400
600
800
1000

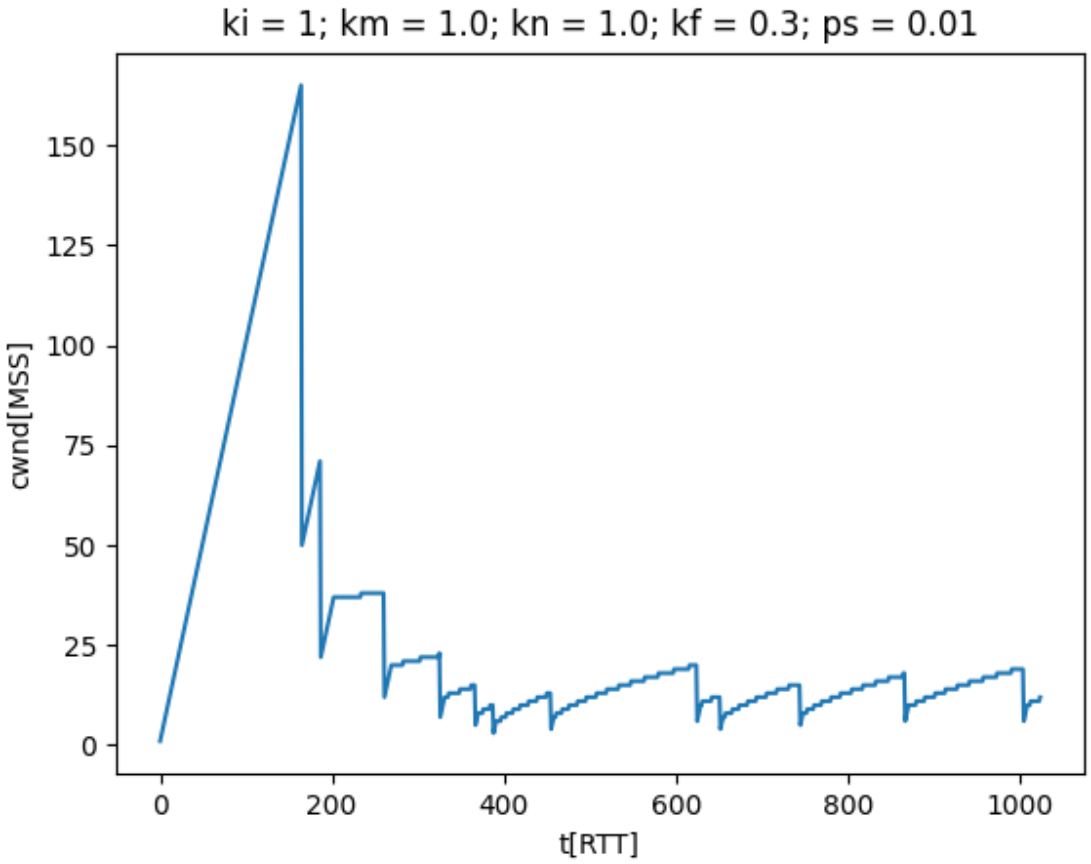

ki = 1; km = 1.0; kn = 0.5; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

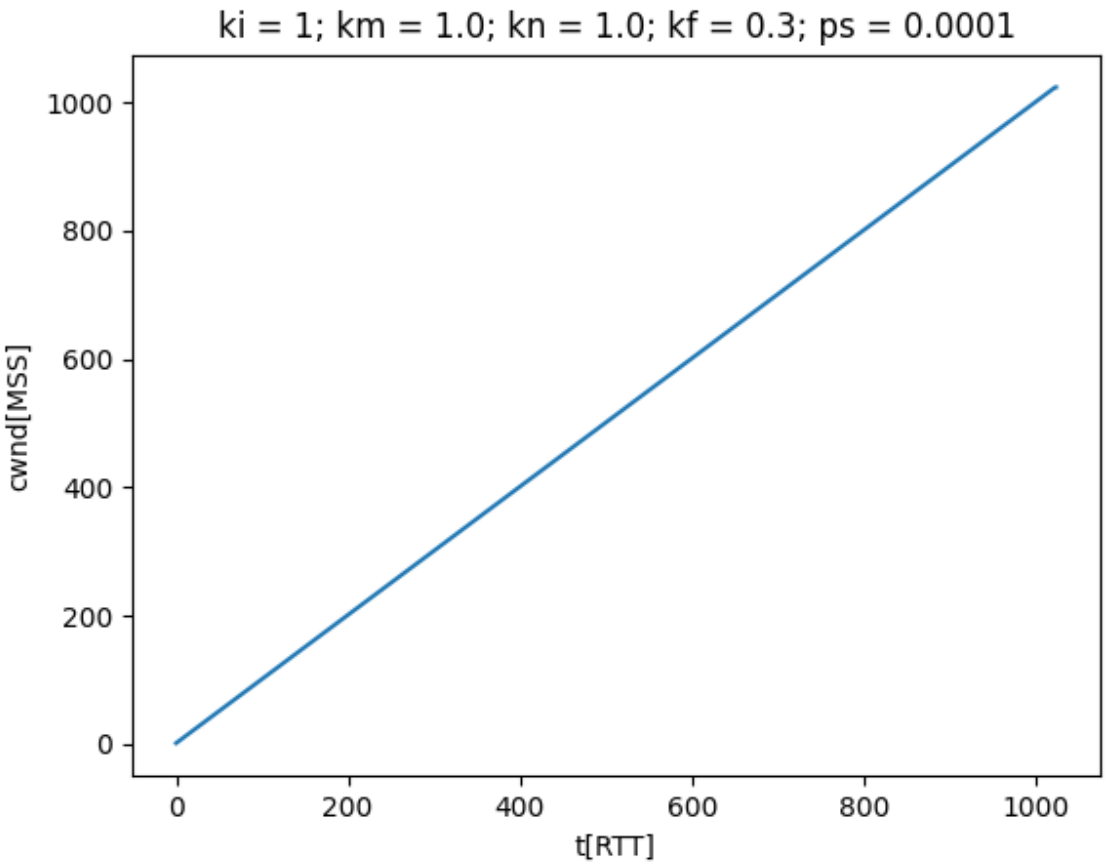

ki = 1; km = 1.0; kn = 0.5; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
0
5
10
15
20
25
30
0
200
400
600
800
1000

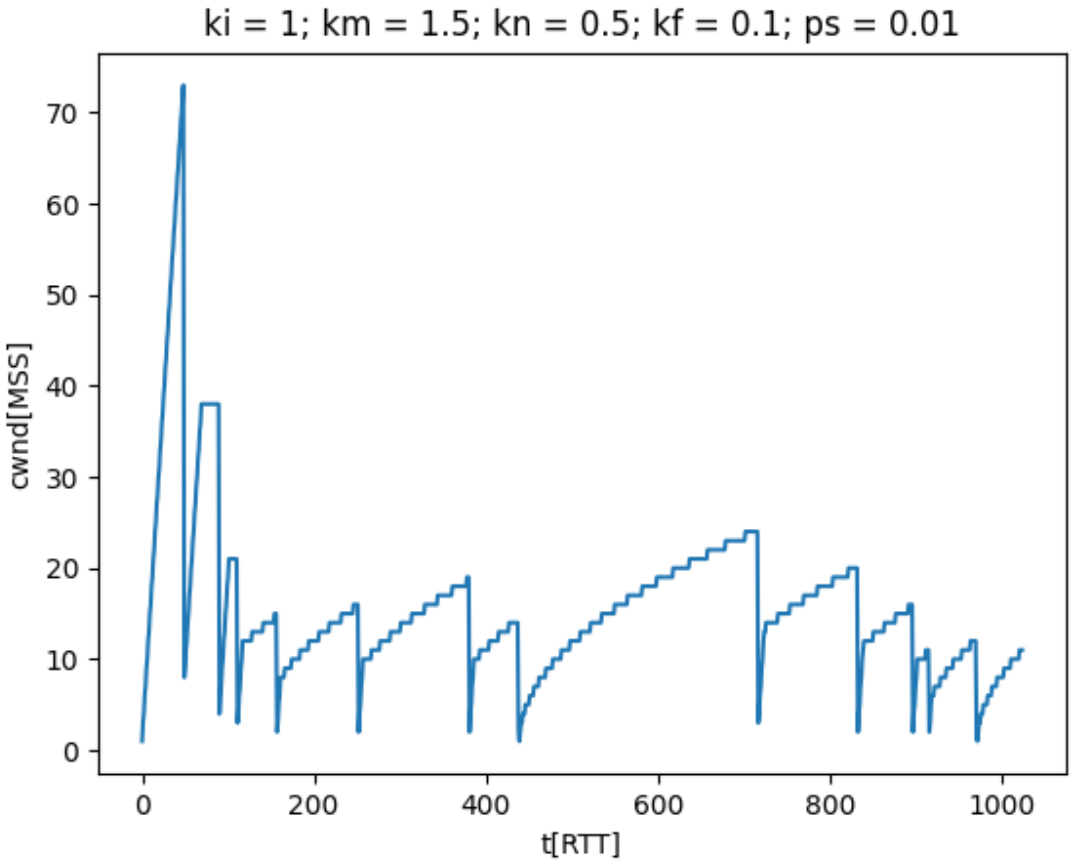

ki = 1; km = 1.0; kn = 0.5; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

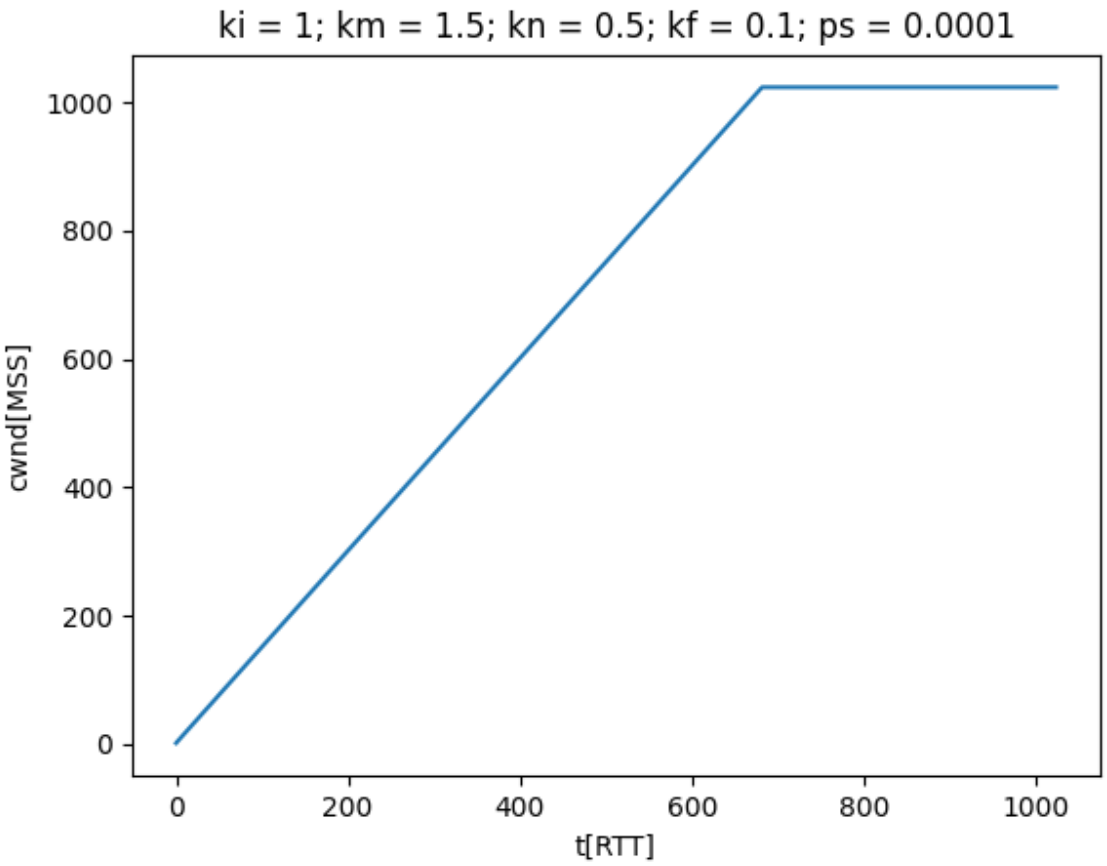

ki = 1; km = 1.0; kn = 1.0; kf = 0.1; ps = 0.01
cwnd[MSS]
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
200
400
600
800
1000
t[RTT]

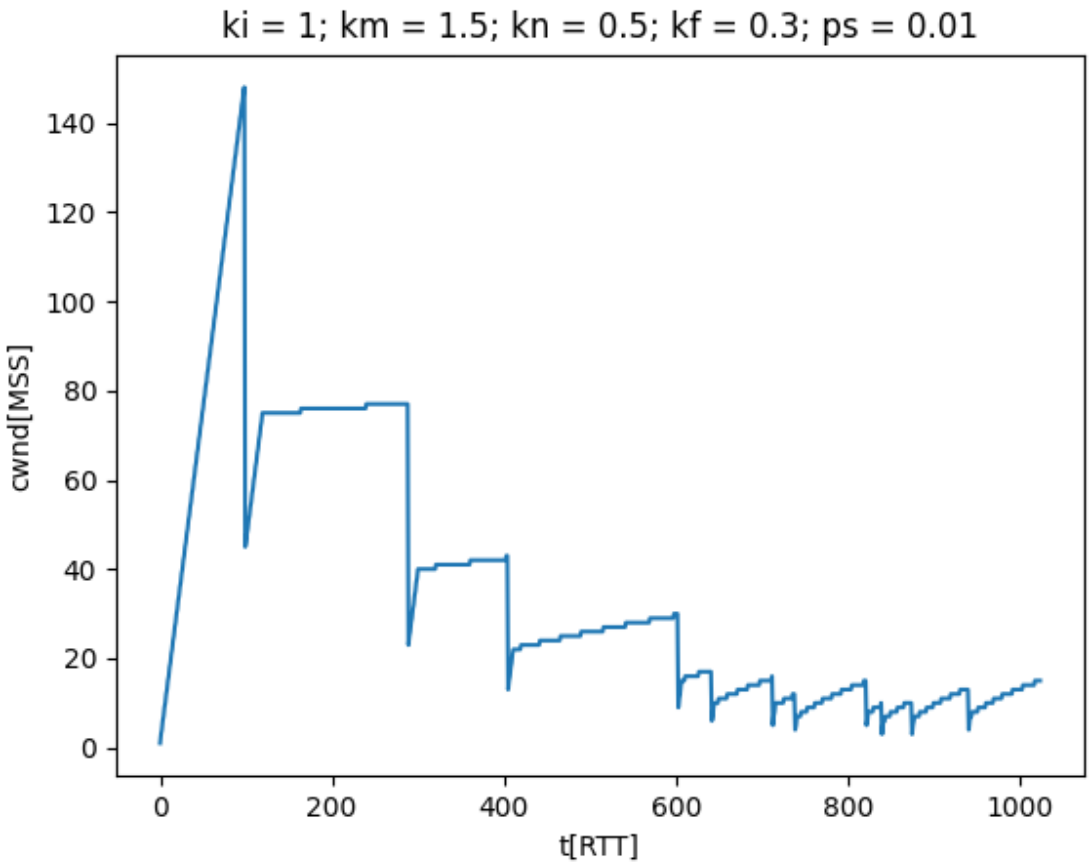

ki = 1; km = 1.0; kn = 1.0; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

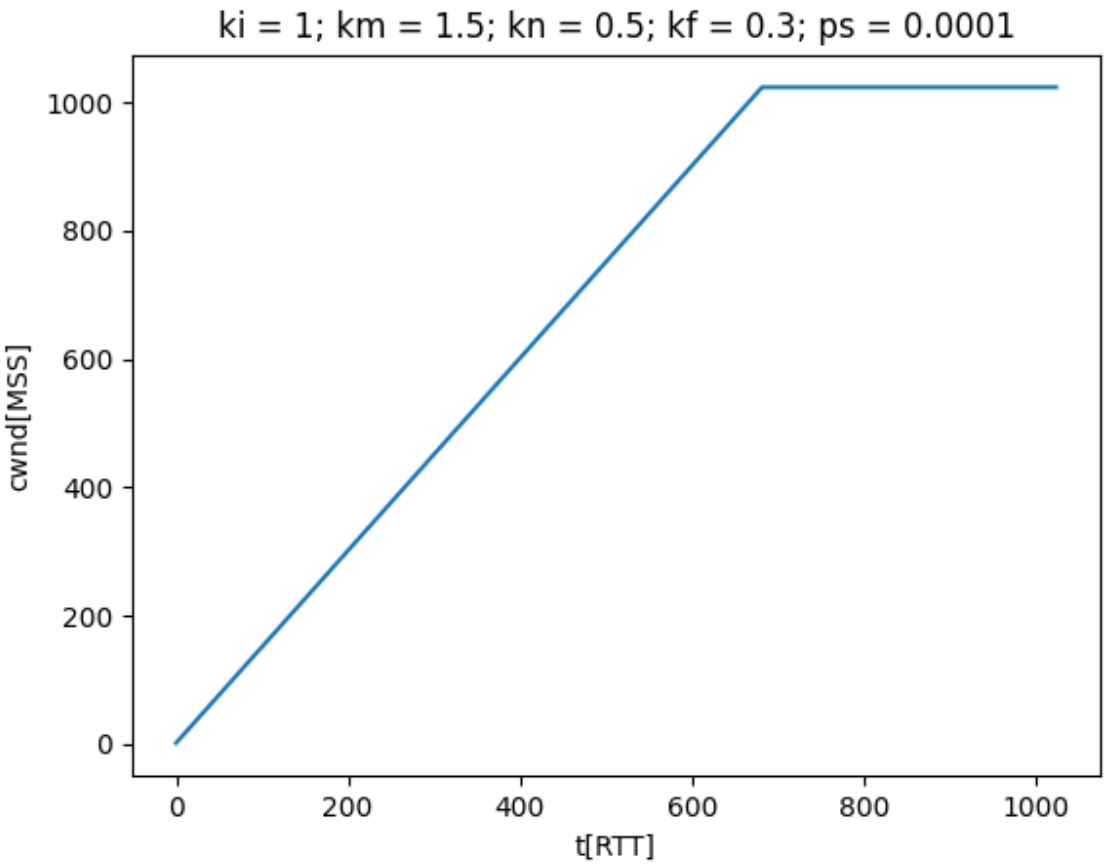

ki = 1; km = 1.0; kn = 1.0; kf = 0.3; ps = 0.01
cwnd[MSS]
150
125
100
75
50
25
0
0
200
400
600
800
1000
t[RTT]

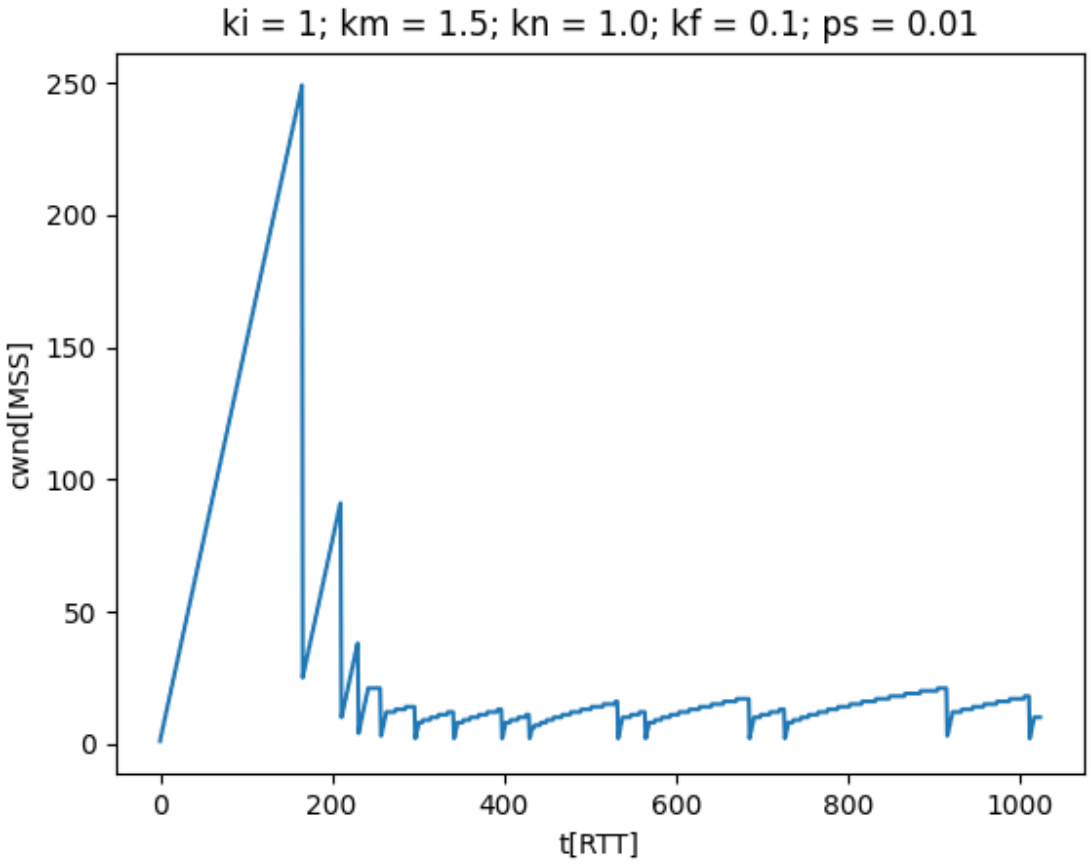

ki = 1; km = 1.0; kn = 1.0; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

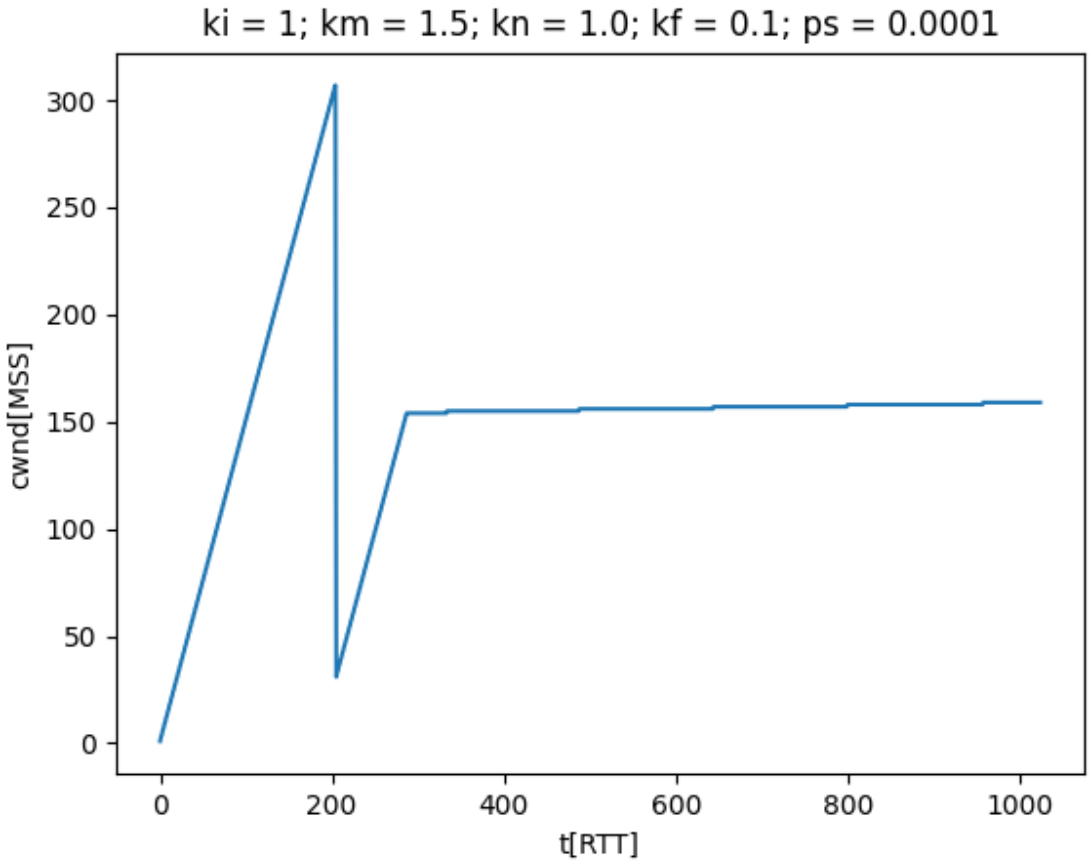

ki = 1; km = 1.5; kn = 0.5; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
10
20
30
40
50
60
70
0
200
400
600
800
1000

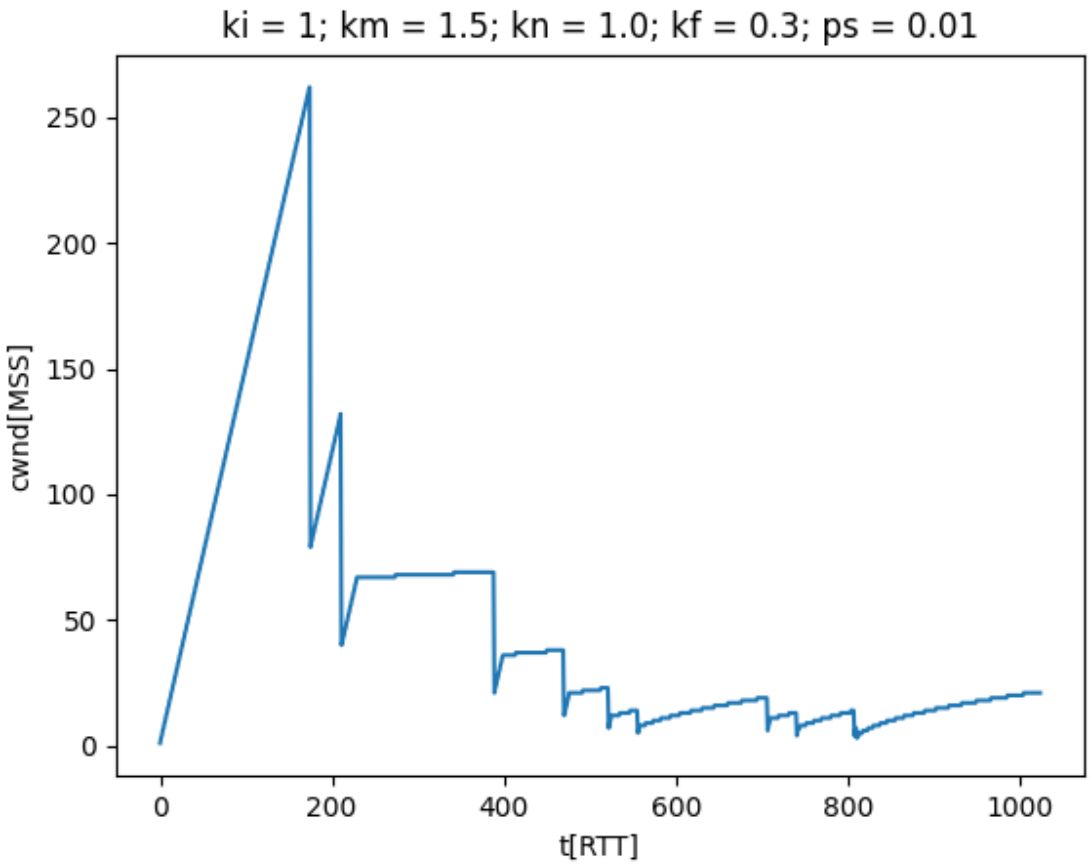

ki = 1; km = 1.5; kn = 0.5; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

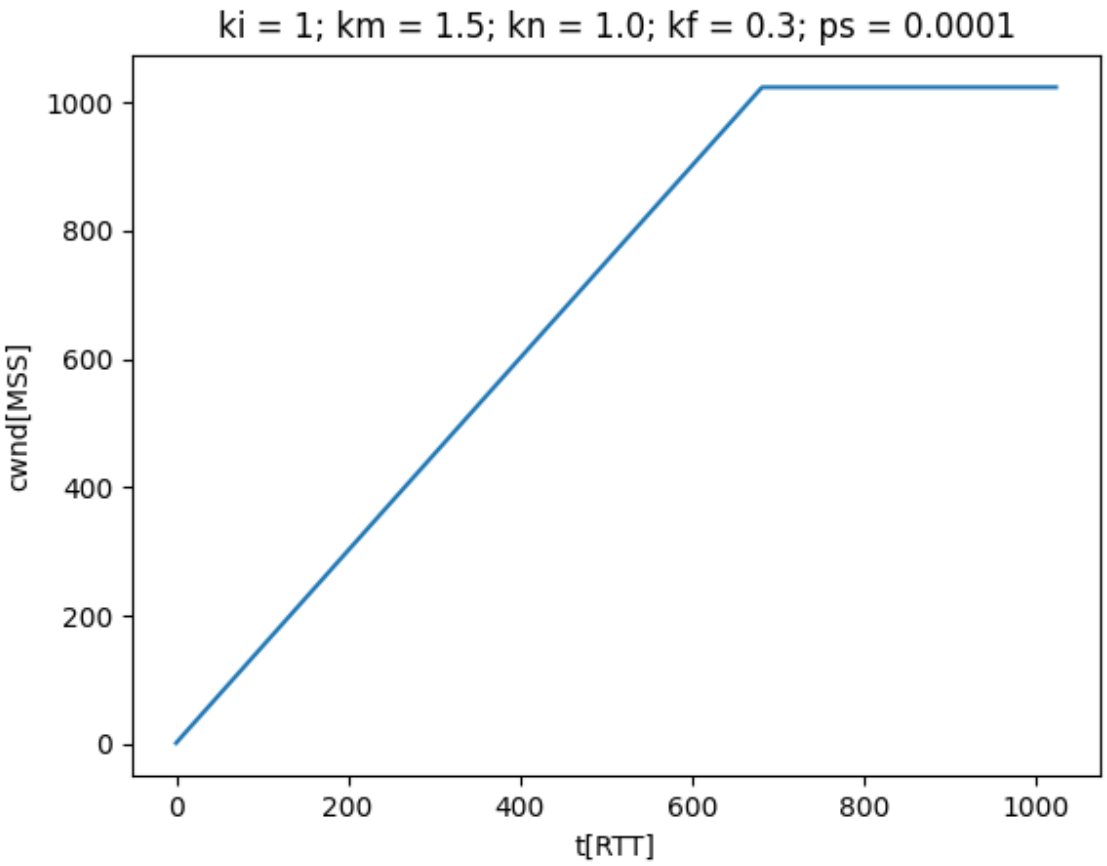

ki = 1; km = 1.5; kn = 0.5; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
0
20
40
60
80
100
120
140
0
200
400
600
800
1000

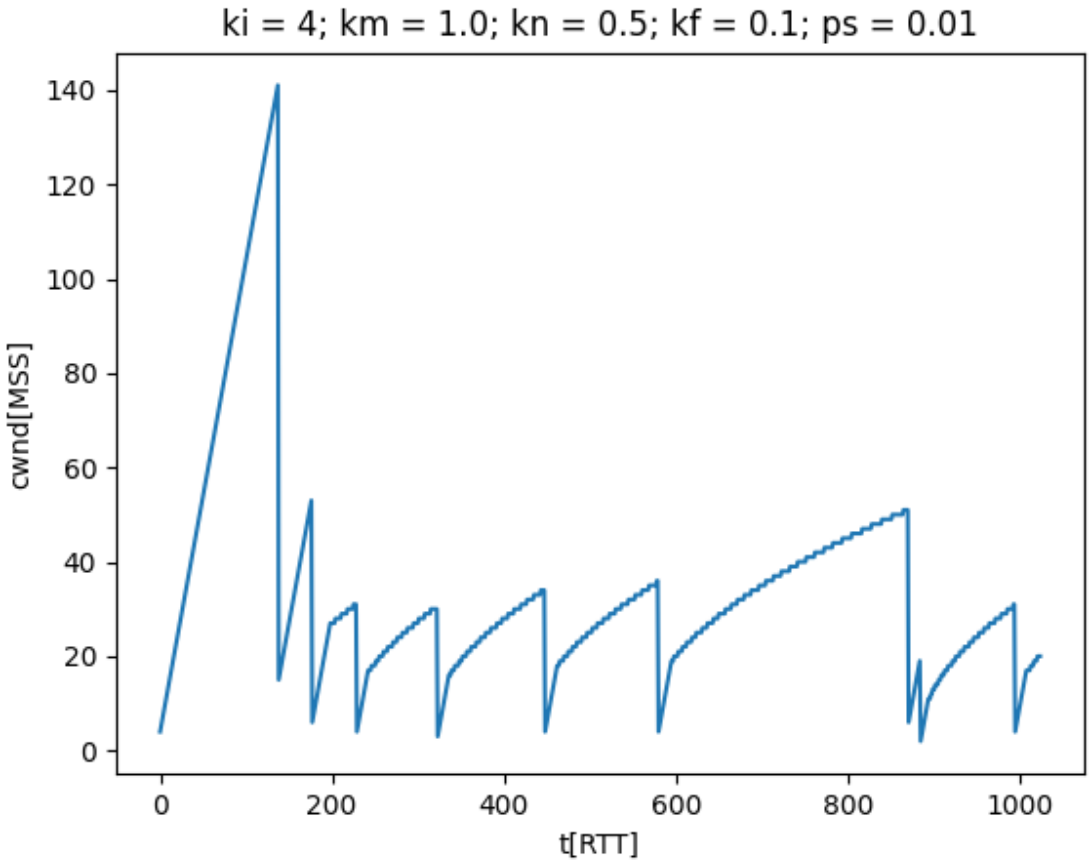

ki = 1; km = 1.5; kn = 0.5; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

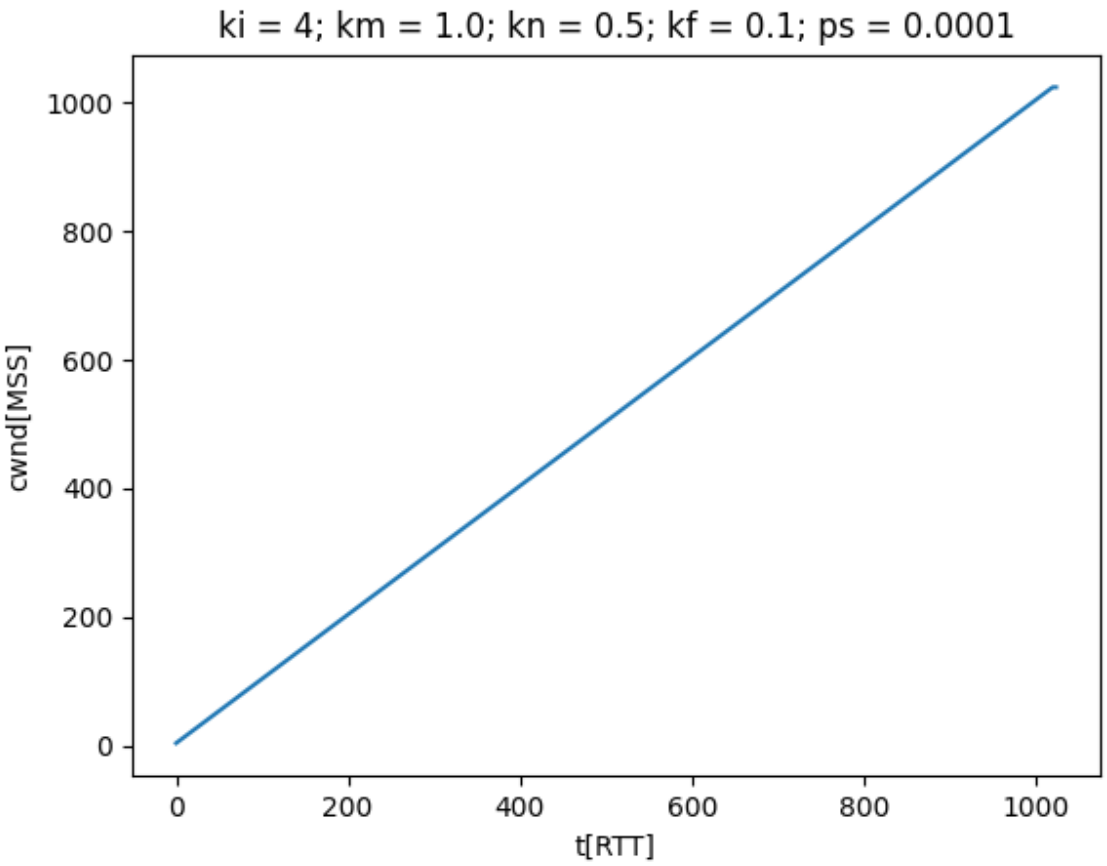

ki = 1; km = 1.5; kn = 1.0; kf = 0.1; ps = 0.01
cwnd[MSS]
250
200
150
100
50
0
0
200
400
600
800
1000
t[RTT]

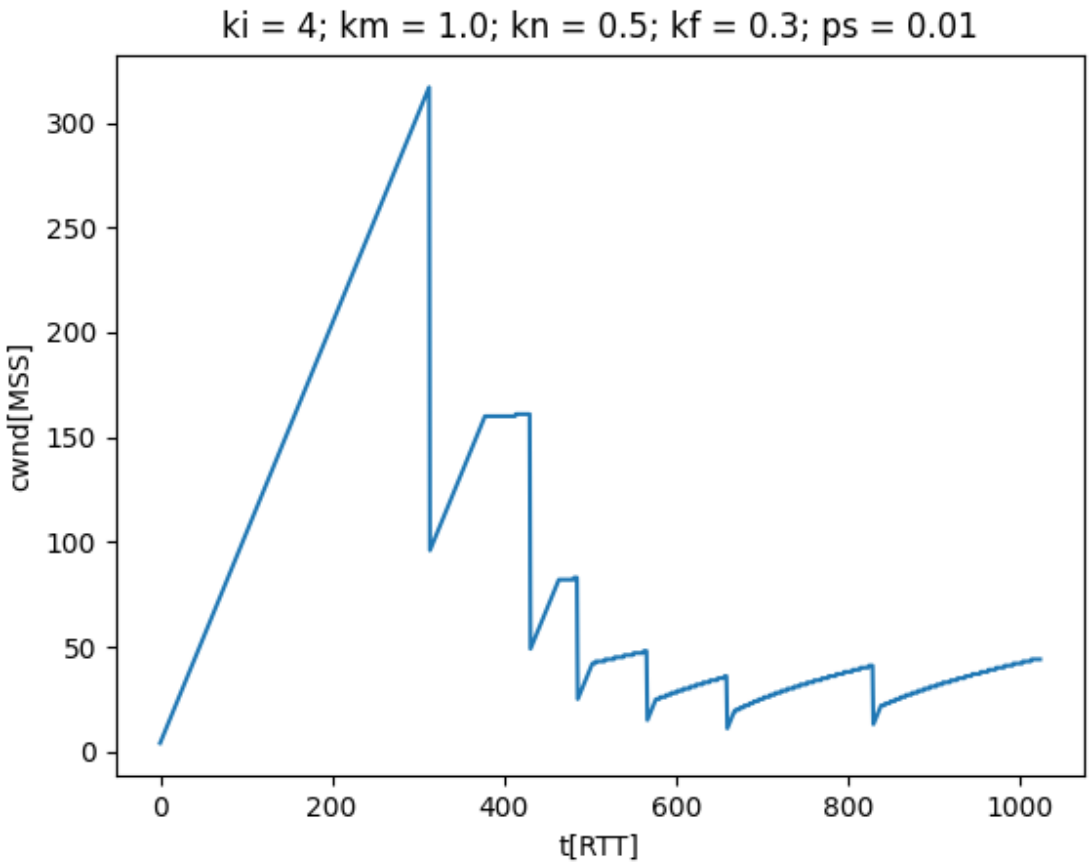

ki = 1; km = 1.5; kn = 1.0; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
50
100
150
200
250
300
0
200
400
600
800
1000

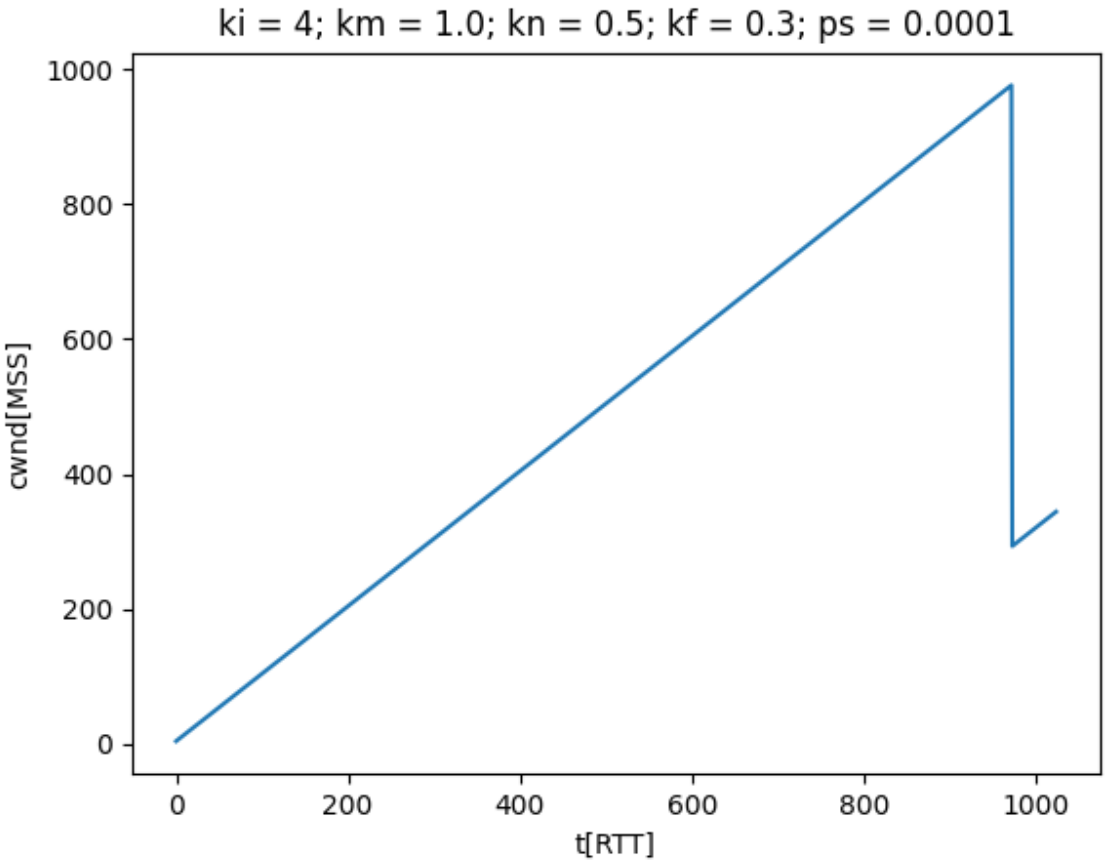

ki = 1; km = 1.5; kn = 1.0; kf = 0.3; ps = 0.01
cwnd[MSS]
250
200
150
100
50
0
0
200
400
600
800
1000
t[RTT]

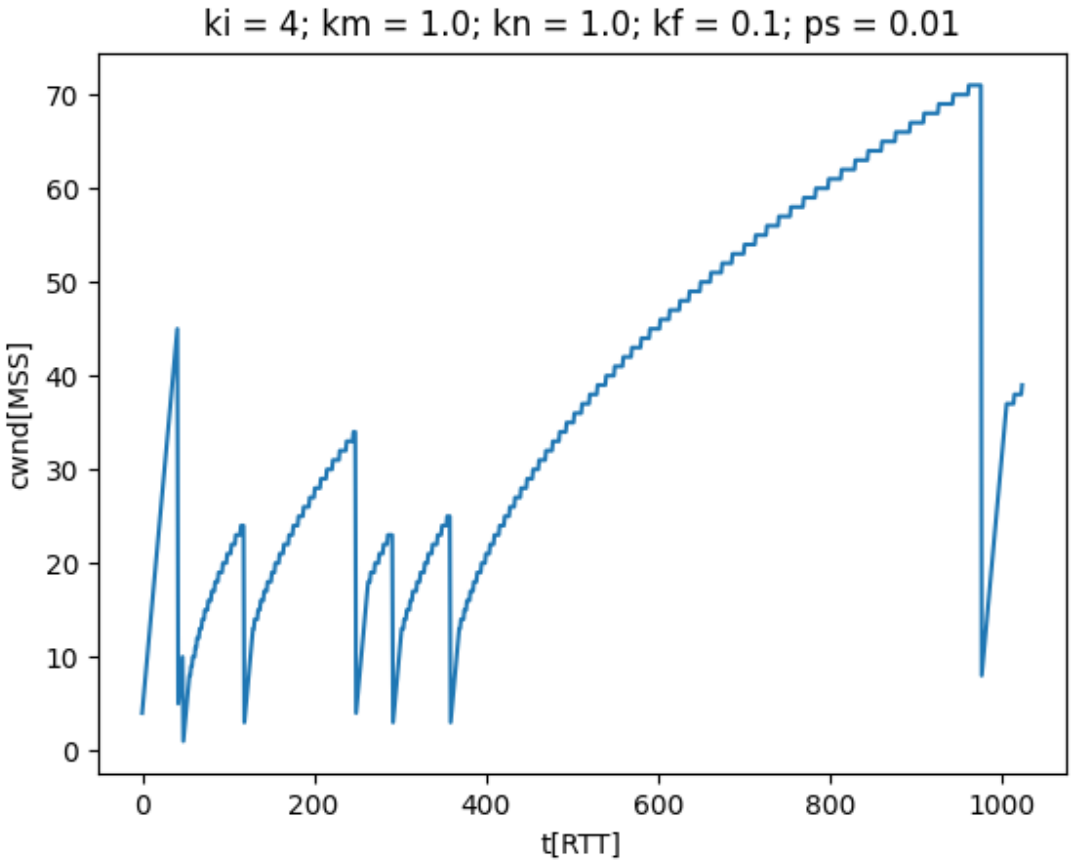

ki = 1; km = 1.5; kn = 1.0; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

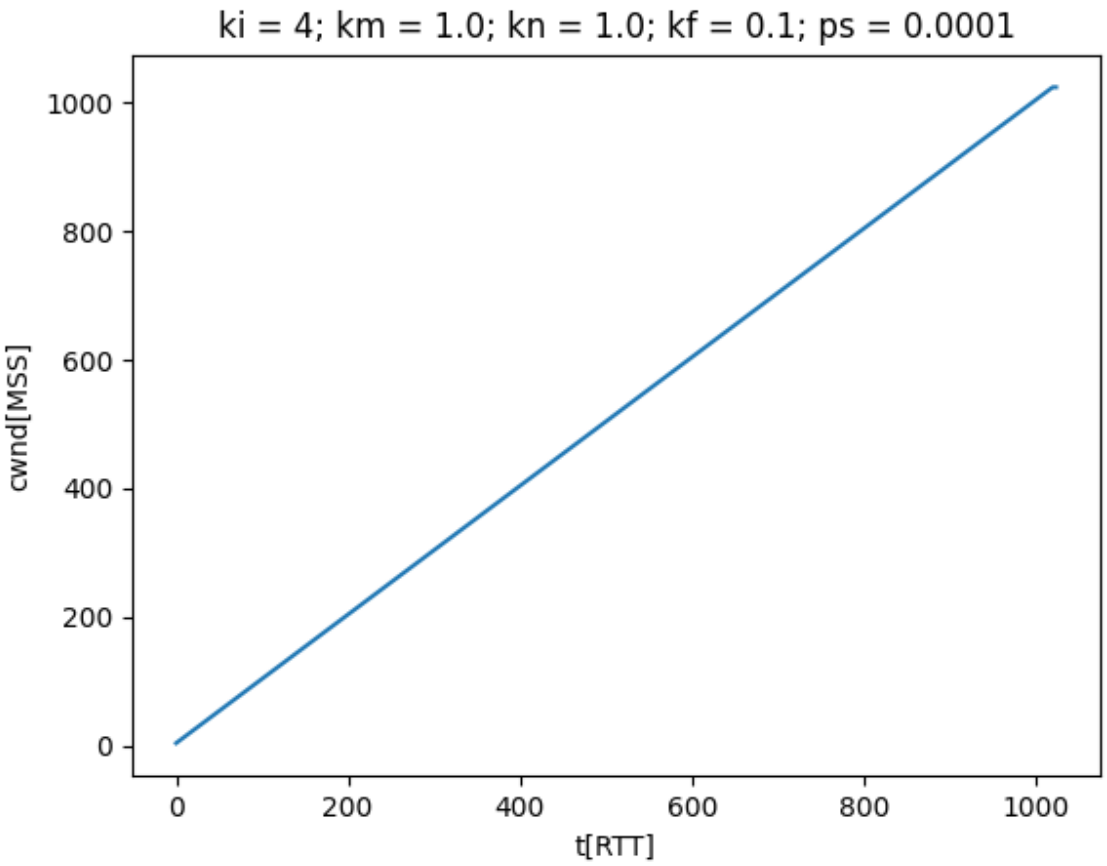

ki = 4; km = 1.0; kn = 0.5; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
20
40
60
80
100
120
140
0
200
400
600
800
1000

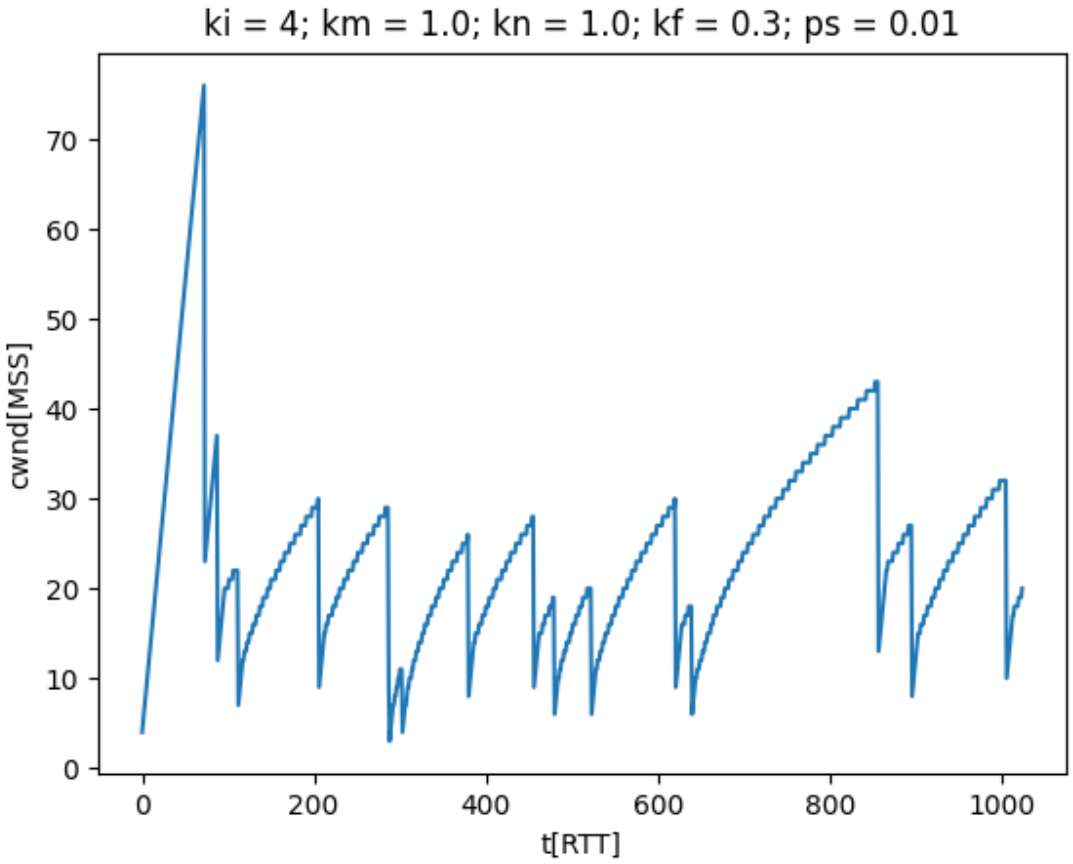

ki = 4; km = 1.0; kn = 0.5; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

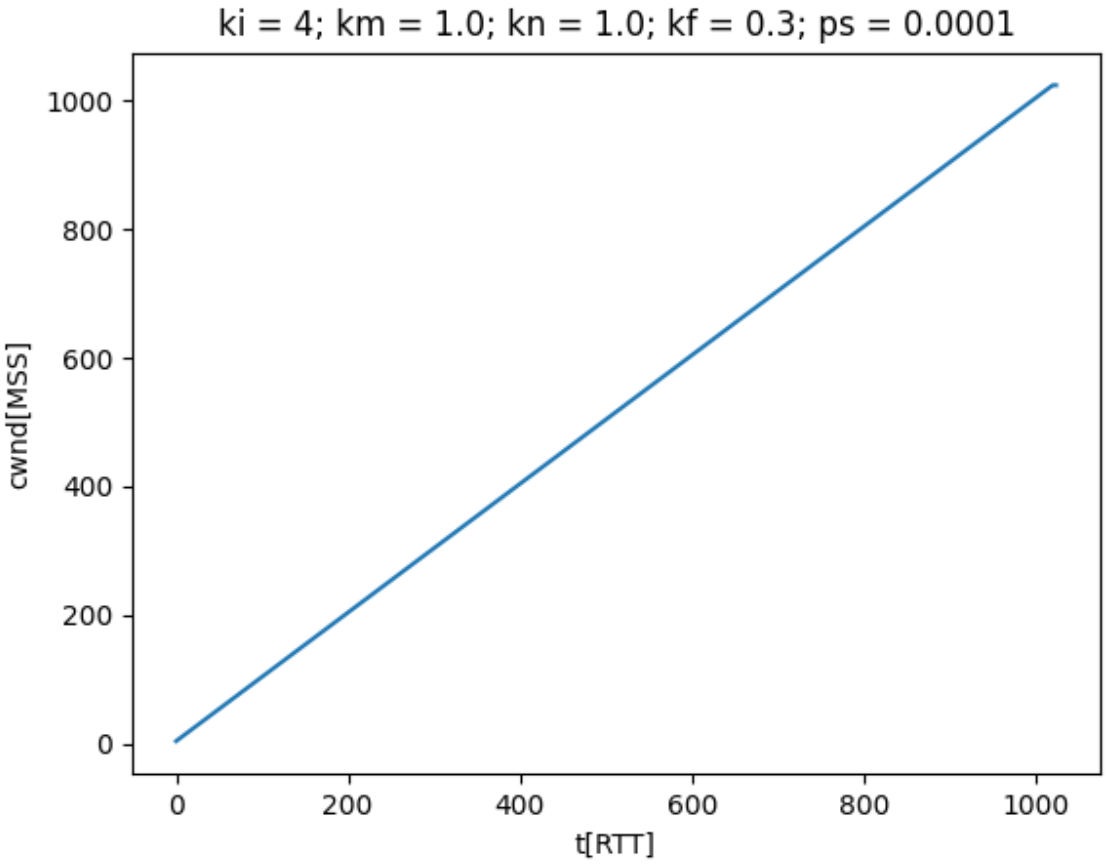

ki = 4; km = 1.0; kn = 0.5; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
0
50
100
150
200
250
300
0
200
400
600
800
1000

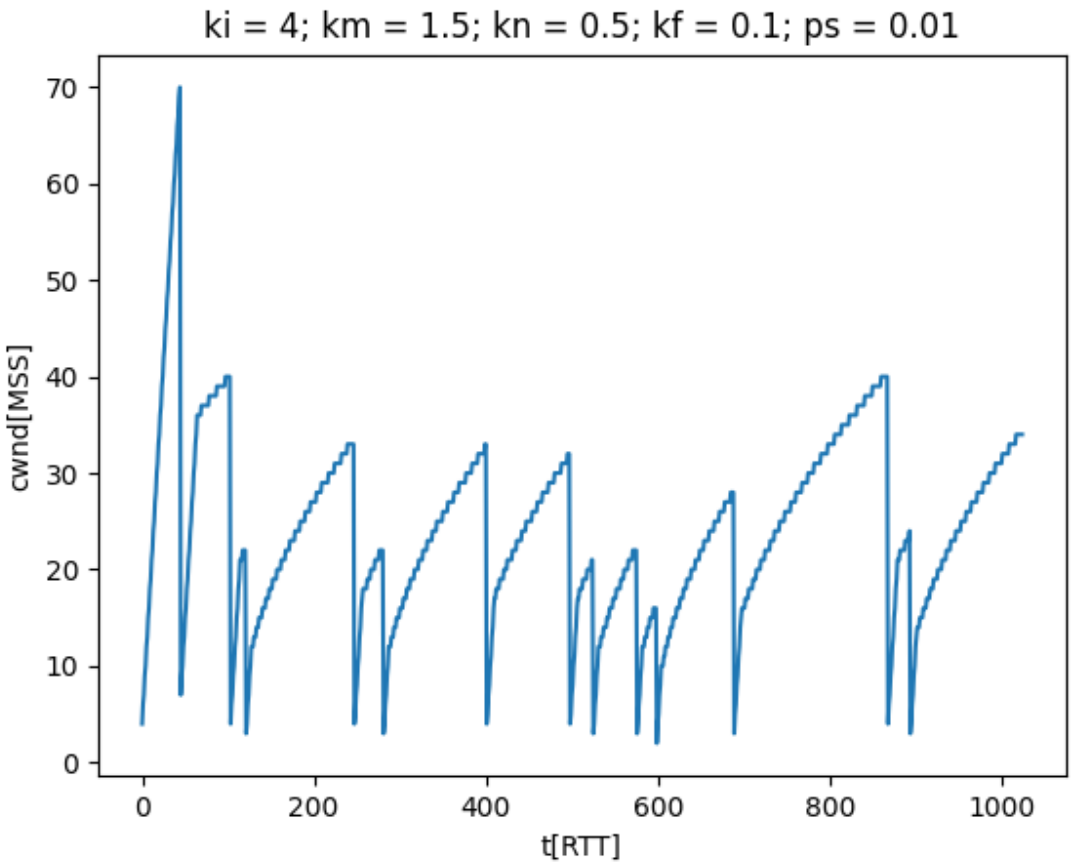

ki = 4; km = 1.0; kn = 0.5; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

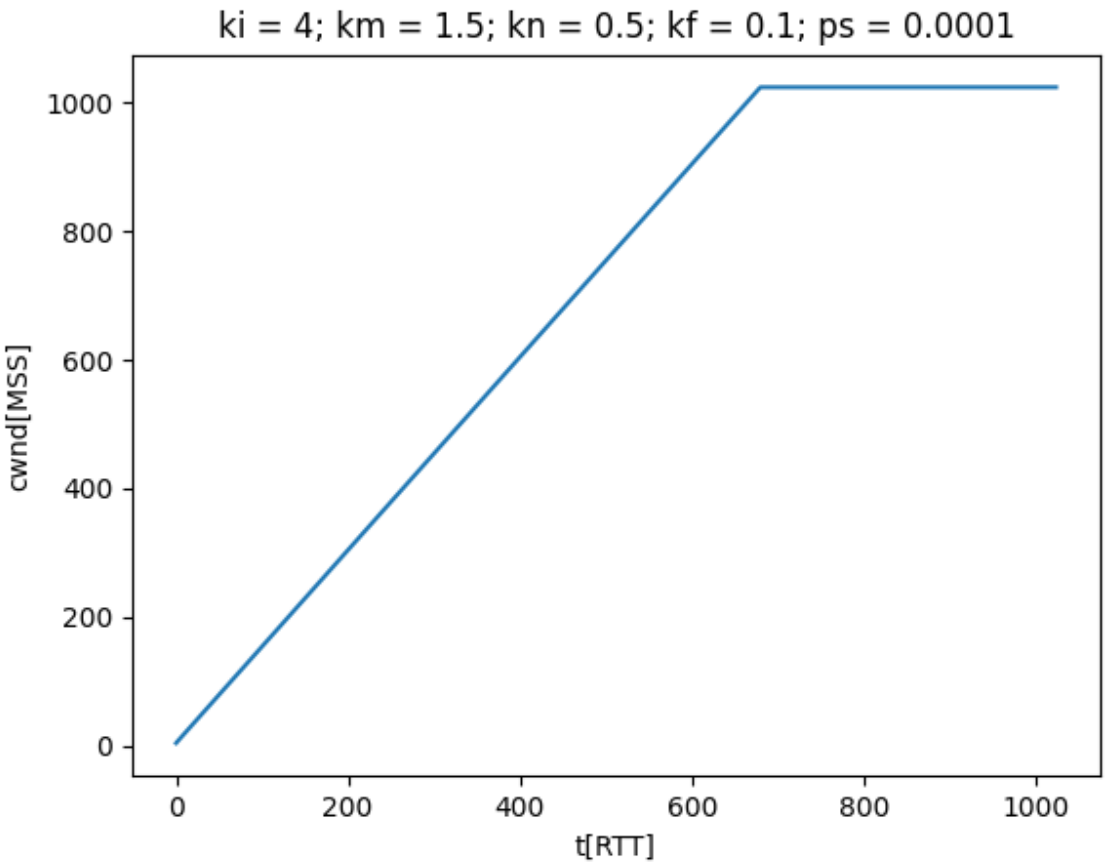

ki = 4; km = 1.0; kn = 1.0; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
10
20
30
40
50
60
70
0
200
400
600
800
1000

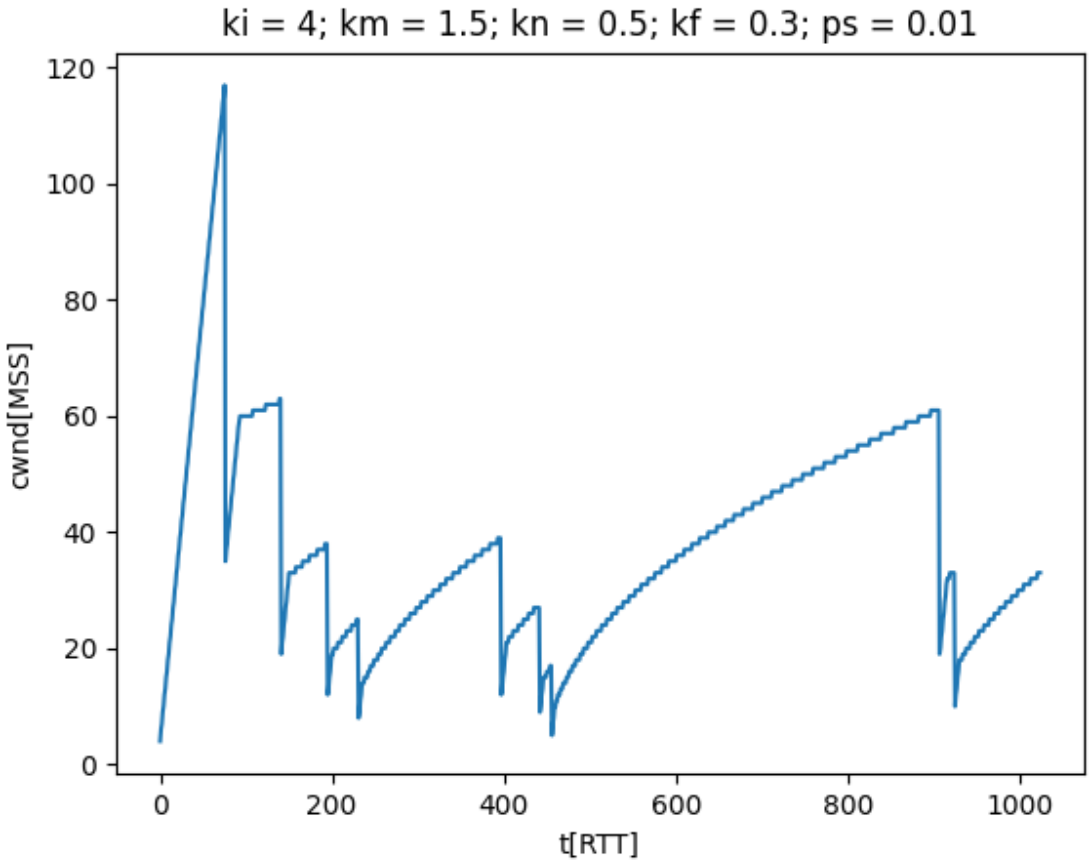

ki = 4; km = 1.0; kn = 1.0; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

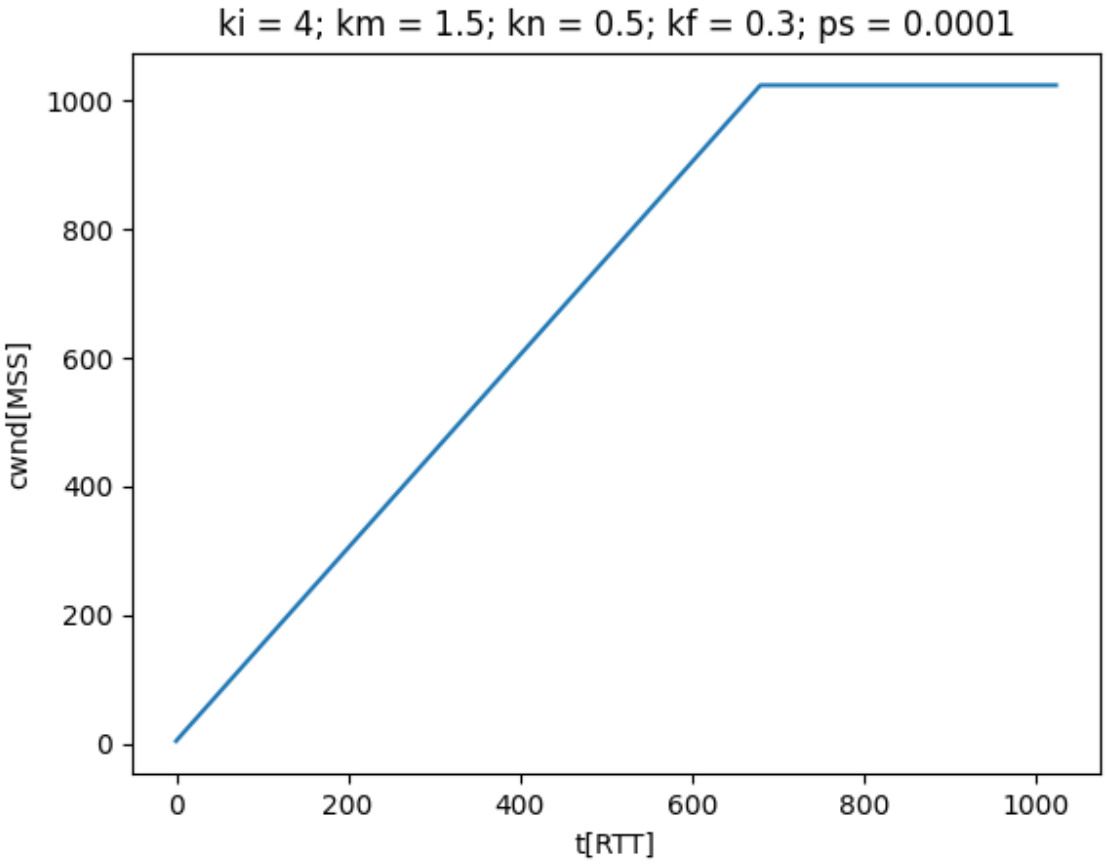

ki = 4; km = 1.0; kn = 1.0; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
0
10
20
30
40
50
60
70
0
200
400
600
800
1000

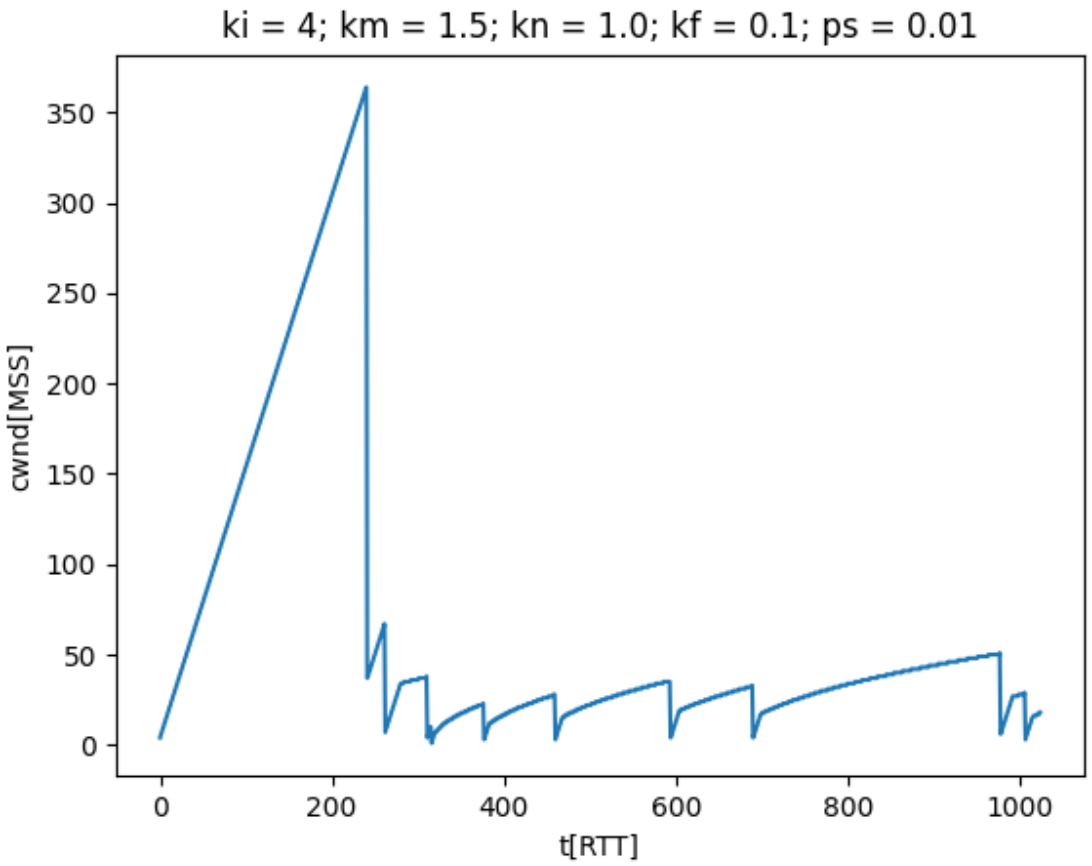

ki = 4; km = 1.0; kn = 1.0; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

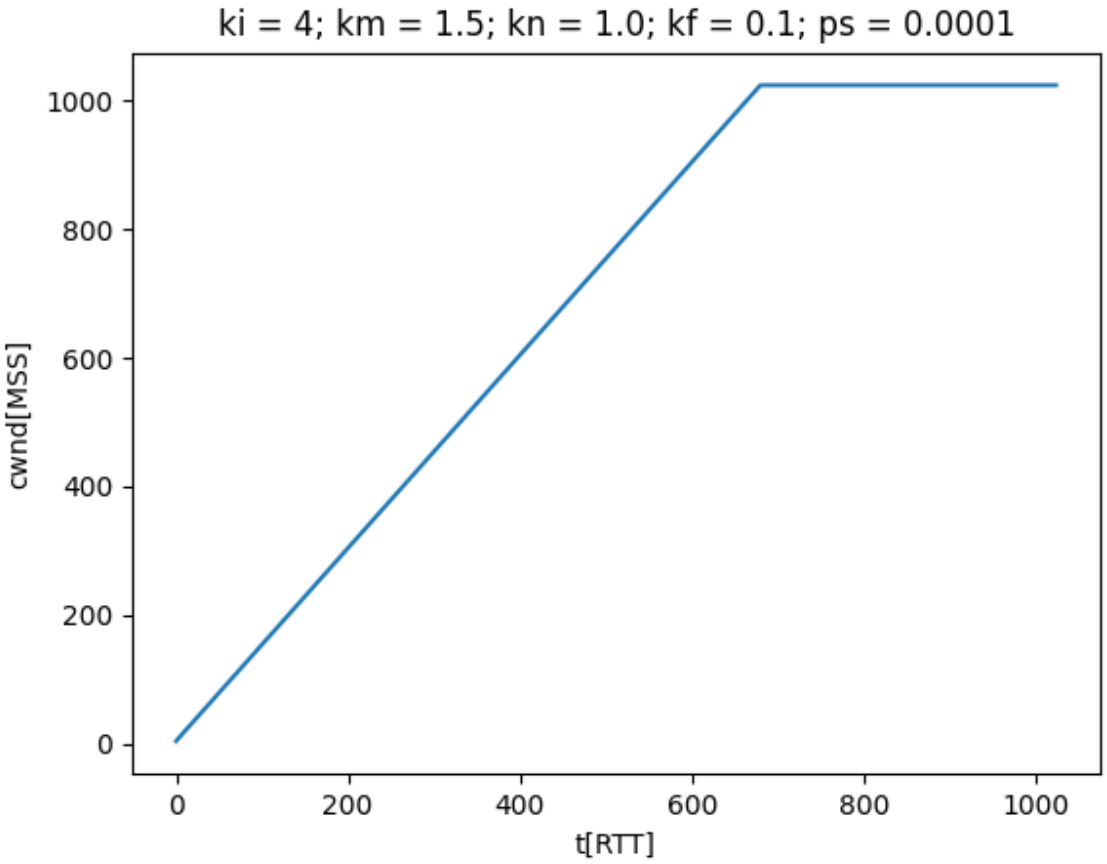

ki = 4; km = 1.5; kn = 0.5; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
10
20
30
40
50
60
70
0
200
400
600
800
1000

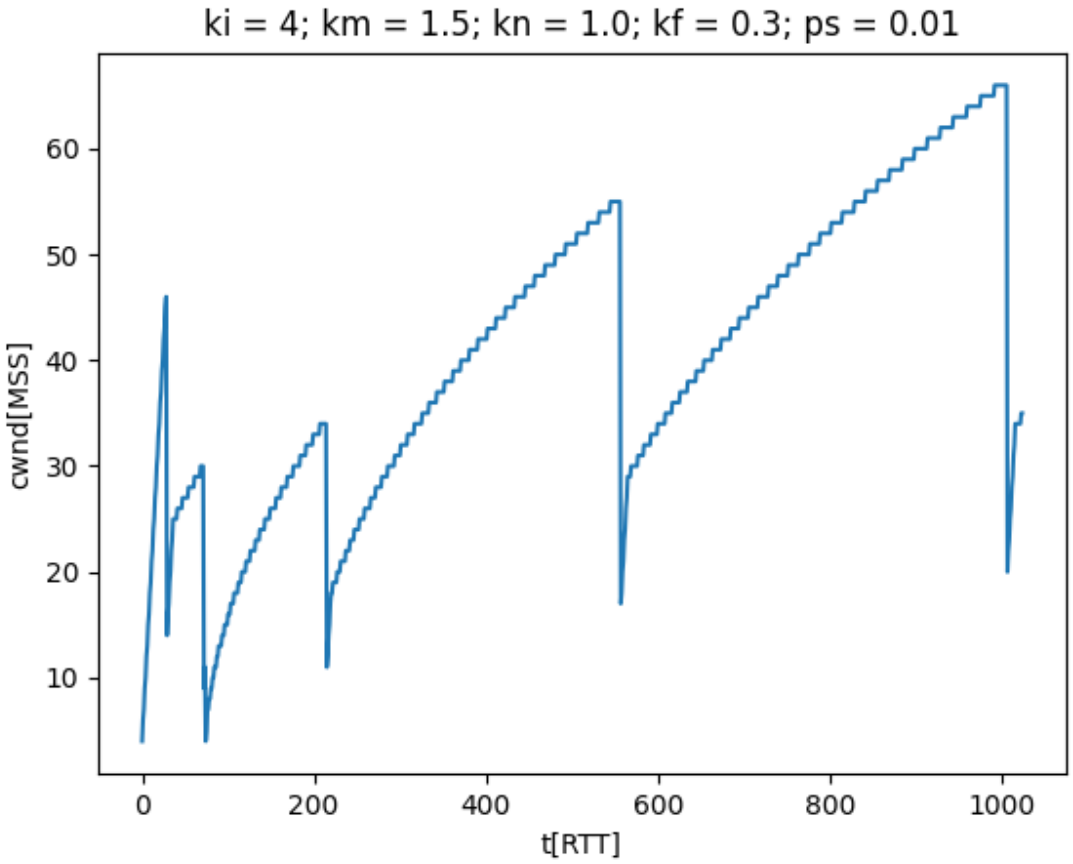

ki = 4; km = 1.5; kn = 0.5; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000

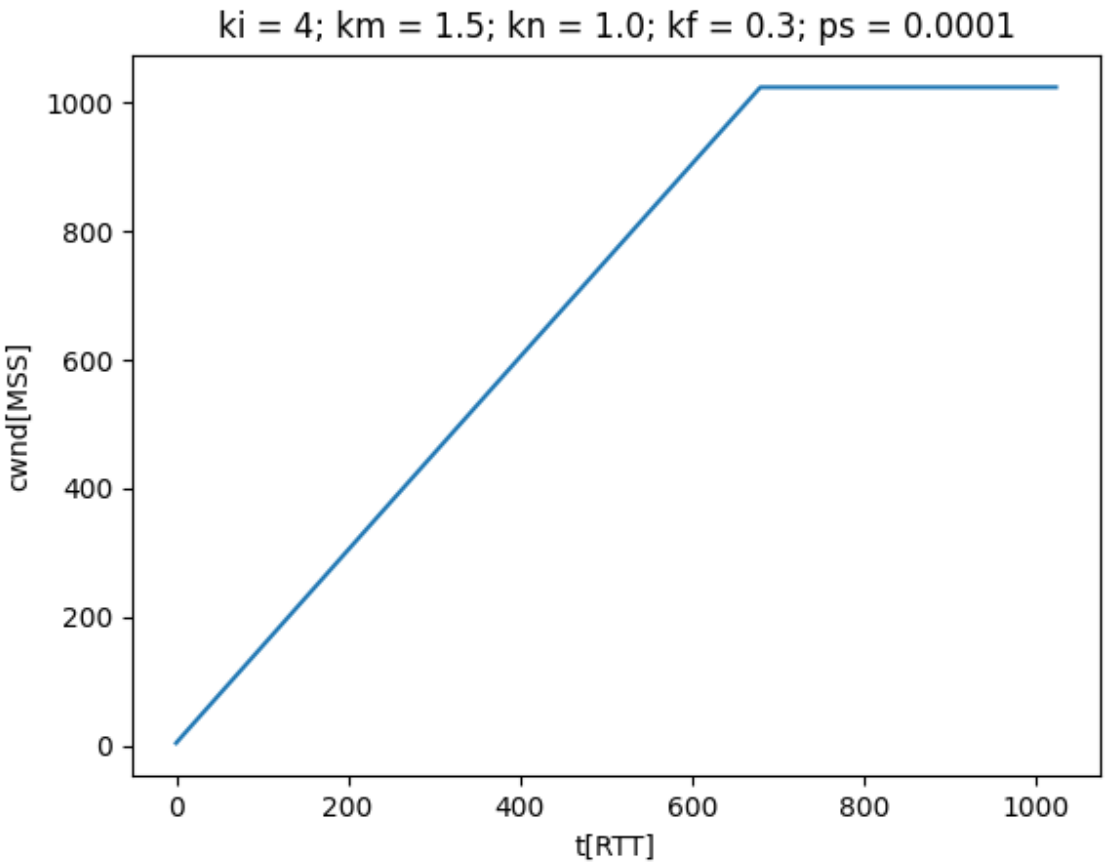

ki = 4; km = 1.5; kn = 0.5; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
0
20
40
60
80
100
120
0
200
400
600
800
1000


ki = 4; km = 1.5; kn = 0.5; kf = 0.3; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000


ki = 4; km = 1.5; kn = 1.0; kf = 0.1; ps = 0.01
cwnd[MSS]
t[RTT]
0
50
100
150
200
250
300
350
0
200
400
600
800
1000


ki = 4; km = 1.5; kn = 1.0; kf = 0.1; ps = 0.0001
cwnd[MSS]
t[RTT]
0
200
400
600
800
1000


ki = 4; km = 1.5; kn = 1.0; kf = 0.3; ps = 0.01
cwnd[MSS]
t[RTT]
10
20
30
40
50
60
0
200
400
600
800
1000



a) Usually the timeouts lead to a very conservative set of cw values.
b) The frequency of timeouts was very less during ps = 0.0001 as it should. Sometimes there was no timeout and just two increasing straight lines (exponential + linear growth).
c) Higher Km values makes the exponential growth faster leading to early timeout. This can be seen as there are lot spikes when Km is higher.
d) After timeout the new CW value is not initialised to the very starting CW value. Hence the left ends of a spike in the graph are not aligned.The threshold is also reduced whenever we have any sort of retransmission to potentially avoid reaching the current messed up state again.
e) Whenever a timeout occurs implying network traffic the threshold is reduced and CW size is reinitialized giving some time for traffic to resolve.
f) If the timeout occurs early (rare) then the threshold becomes very small not letting the CW size increase exponentially.
g) Reinitialization during timeout is considered to be a fraction of CW old size. This makes use of more bandwidths than compared to original method of reinitializing to the very beginning number of CW.

5. Learnings

a) The understanding and illustration of AIMD algorithm.
b) Plotting of data using python and matplotlib.pyplot library.
c) Using a bash script to perform multiple programs and combine them and work on them.
d) Using the script to record the terminal.

6. Additional thoughts if any (like how it could be handled in a better way or limitations)

a) Halfing of threshold value at each retransmission might cause a loss in network bandwidth. This limitation can lead to very slow growth of CW size.

7. Conclusion

a) This emulation gives a broader perspective of how TCP Congestion works at the transport layer and how we can modify the window size to avoid network issues.
b) AIMD algorithm surely helps resolve timeout and acknowledgment failure and helps to avoid coagulation of network traffic.

8. References

1. https://www.taniarascia.com/how-to-create-and-use-bash-scripts/
2. https://www.tutorialspoint.com/unix_commands/script.htm
3. The lecture slides for this assignment for understanding AIMD algorithm.
4. https://www.geeksforgeeks.org/how-to-use-sys-argv-in-python/
5. https://www.geeksforgeeks.org/matplotlib-pyplot-savefig-in-python/
6. https://stackoverflow.com/questions/38532298/how-can-you-plot-data-from-a-txt-file-using-matplotlib