

# **ZEIT8219**

## **Satellite Communications**

*Assignment 2*

NINA AVERILL  
z3531215

UNSW Canberra  
Apr 2022

# Contents

<b>1</b>	<b>Mobile Satellite Service (MSS) Analysis</b>	<b>3</b>
1.1	Assumptions . . . . .	3
1.2	Mobile Earth Station A . . . . .	4
1.2.1	Upstream Channel . . . . .	4
1.2.2	Downstream Channel . . . . .	6
1.3	Mobile Earth Station B . . . . .	8
1.3.1	Upstream Channel . . . . .	8
1.3.2	Downstream Channel . . . . .	10
1.4	Evaluation . . . . .	12
1.4.1	Transmit Capabilities . . . . .	12
1.4.2	Receive Capabilities . . . . .	12
1.4.3	Comparison . . . . .	13
<b>2</b>	<b>Channel Coding Analysis</b>	<b>14</b>
2.0.1	Uplink . . . . .	14
2.0.2	Overall . . . . .	16
2.0.3	Downlink . . . . .	16
2.1	Discussion . . . . .	17
<b>3</b>	<b>Satellite Internet Service Analysis</b>	<b>18</b>
3.1	Mission Summary . . . . .	18
3.2	Network Design . . . . .	18
3.2.1	MQ-4C Triton . . . . .	18
3.2.1.1	Requirements . . . . .	19
3.2.1.2	Antenna Subsystem Parameters . . . . .	19
3.2.2	Satellite Internet Constellation . . . . .	20
3.2.2.1	Comparison of Available Services . . . . .	20
3.2.2.2	Constellation . . . . .	21
3.2.2.3	Antenna Subsystem Parameters . . . . .	21
3.3	Link Analysis . . . . .	21
3.3.1	Assumptions . . . . .	22
3.3.2	Case 1: Near-Ground Operations . . . . .	23
3.3.3	Case 2: High-Altitude Operations . . . . .	25
3.3.4	Analysis . . . . .	27
<b>4</b>	<b>Appendix</b>	<b>28</b>
4.1	Equations . . . . .	28
4.1.1	Link Budget Equations . . . . .	28
4.1.2	Channel Coding Equations . . . . .	28
	<b>References</b>	<b>29</b>
4.2	Python Code Samples . . . . .	30

# 1 Mobile Satellite Service (MSS) Analysis

The following report will analyse a communications service network involving a fixed Earth station and two mobile Earth stations via a single geostationary (GEO) communications satellite. The analysis will consider both the upstream and downstream communications channels to determine the most suitable channel for a user of the service. The geographic coordinates of the Earth stations and the satellite are outlined in Table 1

Table 1: Geographic Coordinates of Network Node

Station	Latitude (deg)	Longitude (deg)
GEO Satellite	00°00'N	149°48'E
Fixed Earth Station	31°10'S	147°16'E
Mobile Earth Station A	19°05'S	178°05'E
Mobile Earth Station B	19°05'S	178°05'E

The GEO communications satellite has an uses the Ka frequency band for both uplink (30.5 GHz) and downlink (20.5 GHz) channels, and has a bandwidth of 50 MHz. This frequency band is appropriate for both FSS and MSS capabilities, as it has a large bandwidth ideal for FSS systems, while reducing the degree of path loss associated with smaller low-gain mobile antennas.

Both uplink and downlink communication on the satellite is done using a single parabolic antenna with a circular diameter of 1m and an efficiency of 60%. The fixed Earth station is fitted with a large parabolic antenna that has a circular diameter of 10.5m, an efficiency of 65% and emits 500W of power. The associated feeder loss of the system is 2.5 dB.

The code and equations used to generate the link budgets is defined in Appendix 1 (lines 29:265) and in Appendix 4.1.1 respectively.

## 1.1 Assumptions

A number of assumptions were made for the purposes of this analysis.

All antennas are assumed to have parabolic reflectors with varying degrees of efficiency and feeder losses.

All communication devices are assumed to be fitted with Low Noise Amplifiers (LNA), and any gain and back-off losses associated with these devices is already included in the stated power levels. Other system losses between the LNAs and parabolic reflectors, including coupling loss and branching losses, are assumed to be negligible for the purposes of this analysis and have been ignored.

The total atmospheric signal attenuation is described by the atmospheric loss that is assumed to be the same for all links, regardless of the slant range or elevation. This atmospheric loss therefore accounts for all beam-spreading, absorption, weather (rain), scintillation, and polarisation losses. The attenuation due to rain fade has also been omitted from the link budget analysis.

The environmental temperature for all Earth stations is defined to be 25°K.

Furthermore, all nodes in the network are assumed to use 8-PSK modulation with no channel coding.

## 1.2 Mobile Earth Station A

Mobile Earth Station A is a parabolic antenna with a circular diameter of 1.2m and an efficiency of 55%. The Earth station is fitted with an LNA that provides 40W of power. The associated feeder loss of the system is 1dB.

### 1.2.1 Upstream Channel

The upstream channel for Earth station A involves communication from the mobile station to the fixed Earth station, via the satellite. A preliminary link budget has been provided in Table 2 using the system definition and assumptions outlined above.

Table 2: Earth Station A Upstream Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	13.888	13.917	35.656
Carrier Power Density (dBW)		-155.405	-163.125
Free-Space Path Loss (dB)		-213.505	-210.079
Atmospheric Loss (dB)		-6.000	-6.000
C/No Ratio (dB)		94.539	116.277
Bandwidth to Bit Rate Ratio (dB)		-3.632	-3.632
C/N Ratio (dB)		17.549	39.288
Central Angle (°)		33.749	35.249
Transmitter Noise Figure ( )		2.800	2.500
Transmitter Equivalent Noise Temperature (K)		45.000	450.000
Transmitter Noise Temperature (K)		25.000	300.000
Transmitter Combined Gain (dB)		45.629	47.874
Transmitter G/Te Ratio (dBK-1)		29.097	21.342
Transmitter Diameter (m)		1.200	1.000
Transmitter Gain (dB)		49.080	44.423
Transmitter Half Beamwidth (dBW)		64.100	52.954

Table 2: Earth Station A Upstream Link Budget

Name	Overall	Uplink	Downlink
Transmitter Frequency (GHz)		30.500	20.500
Transmitter Wavelength (m)		0.010	0.015
Transmitter Feeder Loss (dB)		-1.000	-0.500
Transmitter EIRP (dBW)		64.100	52.954
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		16.021	9.031
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		0.000	0.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		115.385	115.385
Transmitter Modulation Data Rate (mbps)		115.385	115.385
Transmitter Modulation Bandwidth (GHz)		0.050	0.050
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Transmitter Modulation C/N Ratio (bits/s/GHz)			
Transmitter Modulation Eb/No Ratio (dB)			
Transmitter Modulation Bit Error Rate ( )			
Transmitter Modulation Roll-Off Factor ( )		0.300	0.300
Transmitter Earth Station Latitude (°)		-19.080	
Transmitter Earth Station Longitude (°)		178.180	
Transmitter Earth Station Altitude (°)		0.000	
Receiver Noise Figure ( )		2.500	2.100
Receiver Equivalent Noise Temperature (K)		450.000	27.500
Receiver Noise Temperature (K)		300.000	25.000
Receiver Combined Gain (dB)		47.874	65.195
Receiver G/Te Ratio (dBK-1)		21.342	50.801
Receiver Diameter (m)		1.000	10.500
Receiver Gain (dB)		47.874	65.195
Receiver Half Beamwidth (dBW)		56.405	89.684
Receiver Frequency (GHz)		30.500	20.500
Receiver Wavelength (m)		0.010	0.015
Receiver Feeder Loss (dB)		-0.500	-2.500
Receiver EIRP (dBW)		56.405	89.684
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		9.031	26.990
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		0.000	0.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		115.385	115.385
Receiver Modulation Data Rate (mbps)		115.385	115.385
Receiver Modulation Bandwidth (GHz)		0.050	0.050
Receiver Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Receiver Modulation C/N Ratio (bits/s/GHz)			

Table 2: Earth Station A Upstream Link Budget

Name	Overall	Uplink	Downlink
Receiver Modulation Eb/No Ratio (dB)			
Receiver Modulation Bit Error Rate ()			
Receiver Modulation Roll-Off Factor ()		0.300	0.300
Receiver Sub-Satellite Latitude (°)		0.000	
Receiver Sub-Satellite Longitude (°)		149.800	
Receiver Sub-Satellite Altitude (°)		0.000	
Transmitter Sub-Satellite Latitude (°)			0.000
Transmitter Sub-Satellite Longitude (°)			149.800
Transmitter Sub-Satellite Altitude (°)			0.000
Receiver Earth Station Latitude (°)			-35.170
Receiver Earth Station Longitude (°)			147.270
Receiver Earth Station Altitude (°)			0.000

### 1.2.2 Downstream Channel

The downstream channel for Earth station A involves communication from the fixed Earth station to the mobile station, via the satellite. The link budget for both uplink and downlink channels is outlined in Table 3.

Table 3: Earth Station A Downstream Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	13.971	42.927	13.977
Carrier Power Density (dBW)		-126.395	-163.100
Free-Space Path Loss (dB)		-213.530	-210.054
Atmospheric Loss (dB)		-6.000	-6.000
C/No Ratio (dB)		123.548	94.598
Bandwidth to Bit Rate Ratio (dB)		-3.632	-3.632
C/N Ratio (dB)		46.558	17.608
Central Angle (°)		35.249	33.749
Transmitter Noise Figure ()		2.100	2.500
Transmitter Equivalent Noise Temperature (K)		27.500	450.000
Transmitter Noise Temperature (K)		25.000	300.000
Transmitter Combined Gain (dB)		65.195	47.874
Transmitter G/T <sub>e</sub> Ratio (dBK-1)		50.801	21.342
Transmitter Diameter (m)		10.500	1.000
Transmitter Gain (dB)		68.645	44.423
Transmitter Half Beamwidth (dBW)		93.135	52.954
Transmitter Frequency (GHz)		30.500	20.500
Transmitter Wavelength (m)		0.010	0.015

Table 3: Earth Station A Downstream Link Budget

Name	Overall	Uplink	Downlink
Transmitter Feeder Loss (dB)		-2.500	-0.500
Transmitter EIRP (dBW)		93.135	52.954
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		26.990	9.031
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		0.000	0.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		115.385	115.385
Transmitter Modulation Data Rate (mbps)		115.385	115.385
Transmitter Modulation Bandwidth (GHz)		0.050	0.050
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Transmitter Modulation C/N Ratio (bits/s/GHz)			
Transmitter Modulation Eb/No Ratio (dB)			
Transmitter Modulation Bit Error Rate ()			
Transmitter Modulation Roll-Off Factor ()		0.300	0.300
Transmitter Earth Station Latitude (°)		-35.170	
Transmitter Earth Station Longitude (°)		147.270	
Transmitter Earth Station Altitude (°)		0.000	
Receiver Noise Figure ()		2.500	2.800
Receiver Equivalent Noise Temperature (K)		450.000	45.000
Receiver Noise Temperature (K)		300.000	25.000
Receiver Combined Gain (dB)		47.874	45.629
Receiver G/Te Ratio (dBK-1)		21.342	29.097
Receiver Diameter (m)		1.000	1.200
Receiver Gain (dB)		47.874	45.629
Receiver Half Beamwidth (dBW)		56.405	60.650
Receiver Frequency (GHz)		30.500	20.500
Receiver Wavelength (m)		0.010	0.015
Receiver Feeder Loss (dB)		-0.500	-1.000
Receiver EIRP (dBW)		56.405	60.650
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		9.031	16.021
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		0.000	0.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		115.385	115.385
Receiver Modulation Data Rate (mbps)		115.385	115.385
Receiver Modulation Bandwidth (GHz)		0.050	0.050
Receiver Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Receiver Modulation C/N Ratio (bits/s/GHz)			
Receiver Modulation Eb/No Ratio (dB)			
Receiver Modulation Bit Error Rate ()			

Table 3: Earth Station A Downstream Link Budget

Name	Overall	Uplink	Downlink
Receiver Modulation Roll-Off Factor ()		0.300	0.300
Receiver Sub-Satellite Latitude (°)		0.000	
Receiver Sub-Satellite Longitude (°)		149.800	
Receiver Sub-Satellite Altitude (°)		0.000	
Transmitter Sub-Satellite Latitude (°)			0.000
Transmitter Sub-Satellite Longitude (°)			149.800
Transmitter Sub-Satellite Altitude (°)			0.000
Receiver Earth Station Latitude (°)			-19.080
Receiver Earth Station Longitude (°)			178.180
Receiver Earth Station Altitude (°)			0.000

### 1.3 Mobile Earth Station B

Mobile Earth Station A is a parabolic antenna with a circular diameter of 0.8m and an efficiency of 60%. The Earth station is fitted with an LNA that provides 80W of power. The associated feeder loss of the system is 1dB.

#### 1.3.1 Upstream Channel

The link budget for the upstream channel for Earth station B is outlined in Table .

Table 4: Earth Station B Upstream Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	13.755	13.783	35.656
Carrier Power Density (dBW)		-155.538	-163.125
Free-Space Path Loss (dB)		-213.505	-210.079
Atmospheric Loss (dB)		-6.000	-6.000
C/No Ratio (dB)		94.405	116.277
Bandwidth to Bit Rate Ratio (dB)		-3.632	-3.632
C/N Ratio (dB)		17.415	39.288
Central Angle (°)		33.749	35.249
Transmitter Noise Figure ()		2.500	2.500
Transmitter Equivalent Noise Temperature (K)		37.500	450.000
Transmitter Noise Temperature (K)		25.000	300.000
Transmitter Combined Gain (dB)		42.485	47.874
Transmitter G/Te Ratio (dBK-1)		26.745	21.342
Transmitter Diameter (m)		0.800	1.000



Table 4: Earth Station B Upstream Link Budget

Name	Overall	Uplink	Downlink
Transmitter Gain (dB)		45.936	44.423
Transmitter Half Beamwidth (dBW)		63.967	52.954
Transmitter Frequency (GHz)		30.500	20.500
Transmitter Wavelength (m)		0.010	0.015
Transmitter Feeder Loss (dB)		-1.000	-0.500
Transmitter EIRP (dBW)		63.967	52.954
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		19.031	9.031
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		0.000	0.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		115.385	115.385
Transmitter Modulation Data Rate (mbps)		115.385	115.385
Transmitter Modulation Bandwidth (GHz)		0.050	0.050
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Transmitter Modulation C/N Ratio (bits/s/GHz)			
Transmitter Modulation Eb/No Ratio (dB)			
Transmitter Modulation Bit Error Rate ()			
Transmitter Modulation Roll-Off Factor ()		0.300	0.300
Transmitter Earth Station Latitude (°)		-19.080	
Transmitter Earth Station Longitude (°)		178.180	
Transmitter Earth Station Altitude (°)		0.000	
Receiver Noise Figure ()		2.500	2.100
Receiver Equivalent Noise Temperature (K)		450.000	27.500
Receiver Noise Temperature (K)		300.000	25.000
Receiver Combined Gain (dB)		47.874	65.195
Receiver G/Te Ratio (dBK-1)		21.342	50.801
Receiver Diameter (m)		1.000	10.500
Receiver Gain (dB)		47.874	65.195
Receiver Half Beamwidth (dBW)		56.405	89.684
Receiver Frequency (GHz)		30.500	20.500
Receiver Wavelength (m)		0.010	0.015
Receiver Feeder Loss (dB)		-0.500	-2.500
Receiver EIRP (dBW)		56.405	89.684
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		9.031	26.990
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		0.000	0.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		115.385	115.385
Receiver Modulation Data Rate (mbps)		115.385	115.385
Receiver Modulation Bandwidth (GHz)		0.050	0.050

Table 4: Earth Station B Upstream Link Budget

Name	Overall	Uplink	Downlink
Receiver Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Receiver Modulation C/N Ratio (bits/s/GHz)			
Receiver Modulation Eb/No Ratio (dB)			
Receiver Modulation Bit Error Rate ()			
Receiver Modulation Roll-Off Factor ()		0.300	0.300
Receiver Sub-Satellite Latitude (°)		0.000	
Receiver Sub-Satellite Longitude (°)		149.800	
Receiver Sub-Satellite Altitude (°)		0.000	
Transmitter Sub-Satellite Latitude (°)			0.000
Transmitter Sub-Satellite Longitude (°)			149.800
Transmitter Sub-Satellite Altitude (°)			0.000
Receiver Earth Station Latitude (°)			-35.170
Receiver Earth Station Longitude (°)			147.270
Receiver Earth Station Altitude (°)			0.000

### 1.3.2 Downstream Channel

The link budget for the upstream channel for Earth station B is outlined in Table .

Table 5: Earth Station B Downstream Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	11.621	42.927	11.624
Carrier Power Density (dBW)		-126.395	-163.100
Free-Space Path Loss (dB)		-213.530	-210.054
Atmospheric Loss (dB)		-6.000	-6.000
C/No Ratio (dB)		123.548	92.246
Bandwidth to Bit Rate Ratio (dB)		-3.632	-3.632
C/N Ratio (dB)		46.558	15.256
Central Angle (°)		35.249	33.749
Transmitter Noise Figure ()		2.100	2.500
Transmitter Equivalent Noise Temperature (K)		27.500	450.000
Transmitter Noise Temperature (K)		25.000	300.000
Transmitter Combined Gain (dB)		65.195	47.874
Transmitter G/T <sub>e</sub> Ratio (dBK-1)		50.801	21.342
Transmitter Diameter (m)		10.500	1.000
Transmitter Gain (dB)		68.645	44.423
Transmitter Half Beamwidth (dBW)		93.135	52.954
Transmitter Frequency (GHz)		30.500	20.500
Transmitter Wavelength (m)		0.010	0.015

Table 5: Earth Station B Downstream Link Budget

Name	Overall	Uplink	Downlink
Transmitter Feeder Loss (dB)		-2.500	-0.500
Transmitter EIRP (dBW)		93.135	52.954
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		26.990	9.031
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		0.000	0.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		115.385	115.385
Transmitter Modulation Data Rate (mbps)		115.385	115.385
Transmitter Modulation Bandwidth (GHz)		0.050	0.050
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Transmitter Modulation C/N Ratio (bits/s/GHz)			
Transmitter Modulation Eb/No Ratio (dB)			
Transmitter Modulation Bit Error Rate ()			
Transmitter Modulation Roll-Off Factor ()		0.300	0.300
Transmitter Earth Station Latitude (°)		-35.170	
Transmitter Earth Station Longitude (°)		147.270	
Transmitter Earth Station Altitude (°)		0.000	
Receiver Noise Figure ()		2.500	2.500
Receiver Equivalent Noise Temperature (K)		450.000	37.500
Receiver Noise Temperature (K)		300.000	25.000
Receiver Combined Gain (dB)		47.874	42.485
Receiver G/Te Ratio (dBK-1)		21.342	26.745
Receiver Diameter (m)		1.000	0.800
Receiver Gain (dB)		47.874	42.485
Receiver Half Beamwidth (dBW)		56.405	60.516
Receiver Frequency (GHz)		30.500	20.500
Receiver Wavelength (m)		0.010	0.015
Receiver Feeder Loss (dB)		-0.500	-1.000
Receiver EIRP (dBW)		56.405	60.516
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		9.031	19.031
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		0.000	0.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		115.385	115.385
Receiver Modulation Data Rate (mbps)		115.385	115.385
Receiver Modulation Bandwidth (GHz)		0.050	0.050
Receiver Modulation Spectral Efficiency (bits/s/Hz)		2.308	2.308
Receiver Modulation C/N Ratio (bits/s/GHz)			
Receiver Modulation Eb/No Ratio (dB)			
Receiver Modulation Bit Error Rate ()			

Table 5: Earth Station B Downstream Link Budget

Name	Overall	Uplink	Downlink
Receiver Modulation Roll-Off Factor ()		0.300	0.300
Receiver Sub-Satellite Latitude (°)		0.000	
Receiver Sub-Satellite Longitude (°)		149.800	
Receiver Sub-Satellite Altitude (°)		0.000	
Transmitter Sub-Satellite Latitude (°)			0.000
Transmitter Sub-Satellite Longitude (°)			149.800
Transmitter Sub-Satellite Altitude (°)			0.000
Receiver Earth Station Latitude (°)			-19.080
Receiver Earth Station Longitude (°)			178.180
Receiver Earth Station Altitude (°)			0.000

## 1.4 Evaluation

Both mobile Earth stations are co-located, with roughly equivalent slant ranges, free-space path losses, and feeder losses. The difference in performance between the services provided by each mobile station can therefore be fully accounted for by the antenna and amplifier configuration. The performance of both stations will be compared in terms of their transmit and receive capabilities.

### 1.4.1 Transmit Capabilities

Earth station B has both double the power output and a higher efficiency than Earth station A. This suggests that the station is a more efficient transmitter of power and, seeing as both antennas have equal losses, would be the dominant factor affecting the EIRP of equivalent mobile Earth stations. However, Earth station B has a significantly worse transmitter gain. This can be accounted for by the difference of 0.4 metres in the circular diameter between the two parabolic antennas, as the gain for parabolic antennas is a function of the efficiency and the cross-sectional areas of the reflector. A higher gain for Earth station A means that it has much better transmission directivity. For the overall uplink, this translates to an  $\frac{E_b}{N_0}$  difference of 0.133 dB, indicating that Earth station A has more efficient use of energy per bit transmitted.

### 1.4.2 Receive Capabilities

The positive characteristics of Earth station A for transmit also applies to its receive capabilities. On the downlink of the downstream channels, Earth station A exhibits both a 3.15 dB higher receive gain, and 0.13 dB higher EIRP. The relative performance is lower when compared to the antenna's transmit capabilities as a result of Earth station A's higher receive

noise figure, which results in amplified signal noise on the downlink, and a less efficient use of energy per bit. The resulting effect of this is a lower  $\frac{E_b}{N_0}$  ratio, though the ratio is still higher in comparison to Earth station B.

### 1.4.3 Comparison

Overall, Earth station A performs better in terms of both its transmit capabilities for upstream services and receive capabilities, and is the preferred provider of service for a user in the specified location.

## 2 Channel Coding Analysis

The following will analyse the channel coding characteristics of a broadcast television service. The coding channel uses 8-PSK modulation, with a bandwidth of 50 MHz and a filter roll-off factor of 0.3. The service has defined a set of minimum standards required of the communication channel. The service must have a data rate of at least 60 Mbps and a Bit Error Rate (BER) of less than  $1e^{-9}$ . The three code rates that are being considered are outlined in Table 6. The code and equations used to generate the following tables in defined in Appendix 1 (lines 258-35) and Appendix 4.1.2.

Table 6: Convolutional Code Definitions

Convolutional Code Rate	Coding Gain (dB)
7/8	2.5
3/4	3
1/2	3.5

### 2.0.1 Uplink

The minimum  $\frac{E_b}{N_0}$  ratio defined for uplink communication is defined as 33.2dB. Applying each of the coding rates to the communications channel defined above, we get the parameters defined in Tables 7, 8, and 9.

Table 7: 7/8 Channel Summary

Name	Value
Maximum Bit Rate (mbps)	115.385
Data Rate (mbps)	100.962
Bandwidth (GHz)	0.050
Spectral Efficiency (bits/s/Hz)	2.308
C/N Ratio (bits/s/GHz)	36.832
Coded C/N Ratio (bits/s/GHz)	39.332
Eb/No Ratio (dB)	33.200
Coded Eb/No Ratio (dB)	35.700
Bit Error Rate ()	0.000
Coded Bit Error Rate ()	0.000
Roll-Off Factor ()	0.300
Coding Rate (mbps)	0.875
Coding Gain (dB)	2.500

Table 8: 3/4 Channel Summary

Name	Value
Maximum Bit Rate (mbps)	115.385
Data Rate (mbps)	86.538
Bandwidth (GHz)	0.050
Spectral Efficiency (bits/s/Hz)	2.308
C/N Ratio (bits/s/GHz)	36.832
Coded C/N Ratio (bits/s/GHz)	39.832
Eb/No Ratio (dB)	33.200
Coded Eb/No Ratio (dB)	36.200
Bit Error Rate ()	0.000
Coded Bit Error Rate ()	0.000
Roll-Off Factor ()	0.300
Coding Rate (mbps)	0.750
Coding Gain (dB)	3.000

Table 9: 1/2 Channel Summary

Name	Value
Maximum Bit Rate (mbps)	115.385
Data Rate (mbps)	57.692
Bandwidth (GHz)	0.050
Spectral Efficiency (bits/s/Hz)	2.308
C/N Ratio (bits/s/GHz)	36.832
Coded C/N Ratio (bits/s/GHz)	40.332
Eb/No Ratio (dB)	33.200
Coded Eb/No Ratio (dB)	36.700
Bit Error Rate ()	0.000
Coded Bit Error Rate ()	0.000
Roll-Off Factor ()	0.300
Coding Rate (mbps)	0.500
Coding Gain (dB)	3.500

To achieve the most robust communication channels to errors while still maintaining the minimum data rate, the 3/4 convolutional code must be applied. Though the 1/2 channel provides the highest degree of redundancy, the data rate of 57.69 mbps is below the minimum requirement and therefore cannot be used.

## 2.0.2 Overall

The 3/4 coding rate applies an overall coding gain of 3 dB to the communications channel. Assuming a minimum bit error rate of  $1e^{-9}$ , we can then calculate the overall communication channel characteristics outlined in Table 10.

Table 10: Overall Channel Summary

Name	Value
Maximum Bit Rate (mbps)	115.385
Data Rate (mbps)	86.538
Bandwidth (GHz)	0.050
Spectral Efficiency (bits/s/Hz)	2.308
C/N Ratio (bits/s/GHz)	19.657
Coded C/N Ratio (bits/s/GHz)	22.657
Eb/No Ratio (dB)	16.025
Coded Eb/No Ratio (dB)	19.025
Bit Error Rate ()	0.000
Coded Bit Error Rate ()	0.000
Roll-Off Factor ()	0.300
Coding Rate (mbps)	0.750
Coding Gain (dB)	3.000

## 2.0.3 Downlink

To calculate the minimum  $\frac{C}{N}$  ratio for the downlink channel, the uncoded  $\frac{E_b}{N_0}$  ratio must be calculated for the downlink channels. Using Equation 9-28 (Ryan, 2004) rearranged, this can be calculated as 16.11 dB, with a coded  $\frac{E_b}{N_0}$  of 19.11 dB. The coded and uncoded  $\frac{C}{N}$  ratios for the downlink can then be calculated from Equation 6-55 (Ryan, 2004) to be 22.741 dB and 19.741 dB respectively. The resultant coded and uncoded BERs can then be found to be  $7.026 \times 10^{-10}$  and  $8.907 \times 10^{-18}$ .

Table 11: Downlink Modulation Summary

Name	Value
Maximum Bit Rate (mbps)	115.385
Data Rate (mbps)	86.538
Bandwidth (GHz)	0.050
Spectral Efficiency (bits/s/Hz)	2.308
C/N Ratio (bits/s/GHz)	19.741
Coded C/N Ratio (bits/s/GHz)	22.741
Eb/No Ratio (dB)	16.109



Table 11: Downlink Modulation Summary

Name	Value
Coded Eb/No Ratio (dB)	19.109
Bit Error Rate ()	0.000
Coded Bit Error Rate ()	0.000
Roll-Off Factor ()	0.300
Coding Rate (mbps)	0.750
Coding Gain (dB)	3.000

## 2.1 Discussion

The coded system therefore performs significantly better than the uncoded system in most performance benchmarks. Though both the coded and uncoded signal have a BER above the minimum required for the service, this represents the optimal performance achievable by the system. Adverse transmission conditions can degrade the signal significantly and Broadcast Satellite Services (BSS) tend to require a high degree of reliability. The convolutional code therefore provides a significant safety margin above the minimum BER of  $1 \times 10^{-9}$ .

However, though the convolutional code is optimal for conditions where the dominant source of errors is uniformly random errors, it may not be effective for a channel that commonly experiences burst errors. Coding concatenation can then be considered, though this comes at the cost of a lower data rate. Switching to a higher convolutional coding rate may then be necessary to ensure the data rate remains above 60 mbps.

## 3 Satellite Internet Service Analysis

The following report will analyse the feasibility of global LEO satellite constellations to provide service to the fleet of MQ-4C Triton Unmanned Aircraft Systems (UAS) for high altitude, long endurance operations. A low-orbit internet service was selected as the primary communication provider due to the one-way propagation delay of up to 15 ms, which is comparable to that of terrestrial links (He, Gao, Sun, & Zhang, 2021).

### 3.1 Mission Summary

The MQ-4C Triton is a Remotely Piloted Aerial System that is design to perform extended maritime patrol and surveillance (RAAF, 2020). The 2016 Defence White Paper stated that up to seven Tritan UAVs will be the Royal Australian Air Force (ANAO, n.d.), with the intent to be operational by 2025-2026 (DoD, n.d.).

The operations centre for the fleet will be headquartered at RAAF Base Edinbrugh in South Australia, while the majority of flight operations will be conducted out of RAAF Base Tindal in the Northern Territory (DoD, n.d.). The Triton system is able to conduct missions for over 30 hours, with an operational range of roughly 15000 km (Northrop, 2020). The primary region of operations will therefore likely be the maritime border that Australia shares with Indonesia and other Pacific countries.

The extensive range of operations means that the Triton systems will not always be in sight of the ground segment. The system will then conduct Beyond Line of Sight (BLOS) operations, transmitting sensor data via a broadband satellite link to ensure continuous situational awareness of the maritime environment.

### 3.2 Network Design

#### 3.2.1 MQ-4C Triton

The MQ-4C is High-Altitude Long Endurance (HALE) surveillance UAV designed for maritime operations. The vehicle is capable of fully autonomous operations, but can be supported by land-based command and control mission planners and sensor operators. The surveillance operations are enabled by a suite of 360° field of regard (FOR) sensors on the vehicle, including Synthetic Aperture Radar, Electro-Optical / Infrared (EO/IR) sensor, and an automatic identification system (AIS) receiver (TealGroup, 2019).

The Triton vehicles is a maritime derivative on the RQ-4B Global Hawk UAS system (NavyRecognition, 2020), incorporating requirements from the US Navy. A number of assumptions about the Triton vehicles have therefore been derived from publicly available

information on the Global Hawk system. The Global Hawk system has a similar mission goal and surveillance sensor suite to the MQ-4C Tritons, employing both EO/IR and SAR sensors to generate both wide area and spot imagery (Stephenson, 1999). The Global Hawk system used Ku-band satellite communication wideband payload data downlink to provide data rates of 40-360 Mbit/s. (Kramer, 2010) to support a data volume of 274 Mbps (Griethe, 2011). This figure aligns with a paper written by Griethe that estimated the uncompressed data rates for stream EO/IR to be 200-500 mbps, and SAR spot imagery to be 5-10 mbps. We can therefore assume the Triton to have similar data throughput requirements, with some additional margin to account for technological advances in the sensors and onboard computational ability (Griethe, 2011).

The Triton system uses Ka band for both uplink and downlink channels (ThinKom, 2018), allowing for smaller antennas and greater data throughput at the cost of greater susceptibility to atmospheric attenuation.

### 3.2.1.1 Requirements

The intent of fleet the provide real-time intelligence, surveillance and reconnaissance missions (ISR) over vast ocean and coastal regions. This will require rapid, continuous transmission of collected data. The vehicle should be in continuous contact with either the ground segment or the relay satellite. The satellite internet constellation must therefore provide continuous coverage over the area of operations. The antenna subsystem should also be designed to maintain a Bit Error Rate (BER) sufficient to ensure reliable transmission with minimal risk of data corruption from random errors or signal fading. The throughput of the satellite communications channel should also be sufficient for the transmission of both payload and TT&C data.

### 3.2.1.2 Antenna Subsystem Parameters

The prototype Flying Test Bed (FTB) for the Triton vehicle used a ThinAir Ka2517 phased-array satellite antenna to provide BLOS connectivity (Group, 2022). This is a commercial off-the-shelf antenna, and link parameters can therefore be derived from the publicly available data sheet. These antenna parameters are outlined in Table 13

Table 12: ThinAir Ka2517 Phased-Array Antenna Parameters

Parameter	Value
G/T (dB/K)	18.5
EIRP (dBW)	55.5
Tx band (GHz)	27.5 to 31
Rx band (GHz)	17.7 to 21.2
Uplink Throughput (mbps)	800
Downlink Throughput (mbps)	200
Spectral Efficiency (bits/s/Hz)	4
Tx/Rx Power (W)	35
Cross-Sectional Area ( $m^2$ )	542.08
Uplink Center Frequency (GHz)	29

With the uplink and downlink frequency bands centered around a major absorption band, the signal will be heavily attenuated by hydrometers, risking absorption and scattering by the atmosphere. Block coding will therefore required to protect against burst errors. The uplink throughput is more than sufficient to cover the estimated data rates of the system.

### 3.2.2 Satellite Internet Constellation

#### 3.2.2.1 Comparison of Available Services

A number of LEO broadband internet services are proposed or operational, the primary constellations being Telesat, OneWeb, Starlink (SpaceX) and Kuiper (Amazon). A paper by Pachler et. al estimated the maximum system throughput of all four constellation, taking into account publicly available information on the antenna configuration and orbit characteristics (Pachler, del Portillo Barrios, Crawley, & Cameron, 2021). Of the four providers, SpaceX and Amazon allowed for the highest average data rate per satellite of 6.16 Gbps and 16.5 Gbps respectively.

For each user terminal, the Starlink constellation has a data throughput of 100 mbps, and uplink and downlink frequency bands of 14.0-14.5 and 10.7-12.7 GHz respectively (Starlink Services, 2021). This is incompatible with the Triton system and the constellation is therefore not feasible as a service provider.

In comparison, the Kuiper constellation allows for a maximum throughput of 400 mbps, a value that is sufficient to cover the data rate requirements of the UAV with a large margin. The communication channels also have compatible uplink and downlink frequency bands of 27.5-30.0 GHz and 18.8-20.2 GHz respectively. This constellation has therefore been chosen as the primary communications service for the Triton fleet.

### 3.2.2.2 Constellation

The Kuiper constellation is a proposed constellation of 3,236 satellites in 98 orbital planes at altitudes of 590 km, 610 km, and 630 km. The proposed configuration allows for constant coverage over the latitudes of  $\pm 56^\circ\text{N}$ , a range that covers the RAAF fleet's full range of operations (Kuiper Systems, 2020a). The company intends the at least half of the constellation to be fully operational by 2026.

### 3.2.2.3 Antenna Subsystem Parameters

Based on technical documents released by Amazon (Kuiper Systems, 2020b), the Kuiper constellation will have the following antenna parameters:

Table 13: ThinAir Ka2517 Phased-Array Antenna Parameters

Parameter	Value
Gain (dBi)	37
EIRP (dBW)	35.8
Tx band (GHz)	18.8 to 20.2
Rx band (GHz)	27.5 to 31
Uplink Throughput (mbps)	400
Downlink Throughput (mbps)	400
Tx/Rx Power (W)	38.7
Minimum Elevation ( $^\circ$ )	20
Downlink Center Frequency (GHz)	19

The size of the antenna diameter is not specified but can be estimated to be in the range of 1-2.4 m based on similar systems (Pachler et al., 2021). For the purposes of this analysis we have assumed the average antenna size of 1.6m.

## 3.3 Link Analysis

For this analysis, we will assume the Triton is located at a point 1500km from the aircraft base at RAAF Base Tindal, a distance that is near the extent of its operational range. At this location, the Triton will likely be conducting Beyond Line of Sight (BLOS) operations, and therefore relying on the SATCOM constellation for payload and TT&C datalinks. We will consider two scenarios. The first considers operations from near sea-level, covering conditions of operations over launch and low-altitude operations. The second scenario will consider operations at the UAV's maximum operational altitude of 17 km (NavyRecognition, 2020). This analysis therefore covers the full range of operations for the Triton fleet and will consider a worst-case transmission environment to ensure the internet constellation is feasible for the mission.

The Kuiper constellation requires a transmission bandwidth of 50 MHz for uplink, and 100 MHz for downlink, and constrains the available frequency ranges to 27.5-30.0 GHz and 18.8-20.2 GHz. The maximum data rate of the uplink channel is also constrained from 800 mbps allowed by the Triton antenna to 400 mbps.

### 3.3.1 Assumptions

Assuming the fleet will conduct operations within the bounds of Australia's maritime borders, the area of operations for the UAV fleet is within the range of continuous coverage from the Kuiper constellation. It is therefore safe to assume that there will be a satellite within line of sight of the vehicle at all times. For the purposes of this analysis, we will assume the satellite is at the zenith above the UAV, at the maximum orbit altitude of 630 km.

The most significant atmospheric attenuation affects occurs close to the surface of the Earth, below an altitude of 2 km (Notes, n.d.). The increased slant range for the low altitude operations will also increase the free-space path loss. However, for the purposes of this analysis, we will assume the difference to be negligible and will consider the atmospheric loss to be constant for both scenarios. Within the Ka band, we can estimate the rain attenuation at 0.01% of the time to be 83 dB, cloud attenuation to be 1 dB and specific attenuation to be negligible (Al-Saegh et al., 2014) for a total atmospheric attenuation of 84 dB. This attenuation is consistent for both uplink and downlink channels. This high attenuation from hydrometers is consistent with the high humidity and frequent rainfall during the wet season that is associated with the equatorial region of operations.

We will further assume the phase shifter losses of both parabolic antennas to be 6 dB, and the LNA connector loss to be 1 dB, for a total antenna loss of 7 dB (Ahn et al., 2019).

For defence applications, the reliability of the communications channel is critical, therefore the system will target an uplink bit error rate of  $1e^{-6}$  (Li, Lin, & Ma, 2019) to ensure rapid dissemination of the surveillance data. The downlink channel, primarily used for command and control, can tolerate a slightly lower BER due to the reliance on onboard autonomy. The downlink channel will therefore target a BER of  $1e^{-5}$ .

The user terminals for the Kuiper constellation use OFDM and QPSK modulation (Kuiper Systems, 2021). To provide additional error resiliency, the communications channel will use 3/4 convolution coding, resulting in a coding gain of approximately 6.5 dB (Heiskala & Terry, 2001). This will reduce the uplink data rate to 300 mbps but, as outlined in Section 3.2.1, this still leaves an acceptable margin above the minimum required. The analysis will further assume a standard roll-off rate of 0.4.

The code and equations used to generate the following tables are defined in Appendix 1 (lines 348-387) and Appendix 4.1.1.

### 3.3.2 Case 1: Near-Ground Operations

The link analysis for the MQ-4C Tritin UAV during low-altitude operations is outlined in Table 14.

Table 14: Low-Altitude Operations Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	7.023	10.530	9.588
Coded Eb/No Ratio (dB)	13.523	17.030	16.088
Carrier Power Density (dBW)		-206.183	-222.210
Free-Space Path Loss (dB)		-177.683	-174.010
Atmospheric Loss (dB)		-84.000	-84.000
C/No Ratio (dB)			
Bandwidth to Bit Rate Ratio (dB)		-9.031	-3.010
C/N Ratio (dB)		26.061	19.098
Central Angle (°)		0.000	0.000
Slant Range (km)		630.000	630.000
Coded C/N Ratio (dB)		26.061	19.098
Transmitter Noise Figure ()			
Transmitter Equivalent Noise Temperature (K)			
Transmitter Noise Temperature (K)			
Transmitter Combined Gain (dB)		47.059	37.000
Transmitter G/Te Ratio (dBK-1)			
Transmitter Frequency (GHz)		29.000	19.000
Transmitter Wavelength (m)		0.010	0.016
Transmitter Feeder Loss (dB)		-6.000	-6.000
Transmitter Gain (dB)		47.059	37.000
Transmitter EIRP (dBW)		55.500	35.800
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		15.441	15.877
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		-1.000	-1.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		400.000	200.000
Transmitter Modulation Data Rate (mbps)		300.000	150.000
Transmitter Modulation Bandwidth (GHz)		0.050	0.100
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		4.000	2.000
Transmitter Modulation C/N Ratio (bits/s/GHz)		19.561	12.598
Transmitter Modulation Coded C/N Ratio (bits/s/GHz)		26.061	19.098
Transmitter Modulation Eb/No Ratio (dB)		10.530	9.588
Transmitter Modulation Coded Eb/No Ratio (dB)		17.030	16.088
Transmitter Modulation Bit Error Rate ()		0.000	0.000
Transmitter Modulation Coded Bit Error Rate ()		0.000	0.000
Transmitter Modulation Roll-Off Factor ()		0.400	0.400

Table 14: Low-Altitude Operations Link Budget

Name	Overall	Uplink	Downlink
Transmitter Modulation Coding Rate (mbps)		0.750	0.750
Transmitter Modulation Coding Gain (dB)		6.500	6.500
Transmitter Earth Station Latitude (°)		-3.746	
Transmitter Earth Station Longitude (°)		124.401	
Transmitter Earth Station Altitude (°)		0.000	
Receiver Noise Figure (°)			
Receiver Equivalent Noise Temperature (K)			
Receiver Noise Temperature (K)			
Receiver Combined Gain (dB)		37.000	47.059
Receiver G/T <sub>e</sub> Ratio (dBK-1)			
Receiver Frequency (GHz)		29.000	19.000
Receiver Wavelength (m)		0.010	0.016
Receiver Feeder Loss (dB)		-6.000	-6.000
Receiver Gain (dB)		37.000	47.059
Receiver EIRP (dBW)		46.000	55.500
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		15.877	15.441
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		-1.000	-1.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		400.000	200.000
Receiver Modulation Data Rate (mbps)		300.000	150.000
Receiver Modulation Bandwidth (GHz)		0.050	0.100
Receiver Modulation Spectral Efficiency (bits/s/Hz)		8.000	4.000
Receiver Modulation C/N Ratio (bits/s/GHz)		19.561	12.598
Receiver Modulation Coded C/N Ratio (bits/s/GHz)		26.061	19.098
Receiver Modulation Eb/No Ratio (dB)		10.530	9.588
Receiver Modulation Coded Eb/No Ratio (dB)		17.030	16.088
Receiver Modulation Bit Error Rate (°)		0.000	0.000
Receiver Modulation Coded Bit Error Rate (°)		0.000	0.000
Receiver Modulation Roll-Off Factor (°)		0.400	0.400
Receiver Modulation Coding Rate (mbps)		0.750	0.750
Receiver Modulation Coding Gain (dB)		6.500	6.500
Receiver Sub-Satellite Latitude (°)		-3.746	
Receiver Sub-Satellite Longitude (°)		124.401	
Receiver Sub-Satellite Altitude (°)		16.000	
Transmitter Sub-Satellite Latitude (°)			-3.746
Transmitter Sub-Satellite Longitude (°)			124.401
Transmitter Sub-Satellite Altitude (°)			16.000
Receiver Earth Station Latitude (°)			-3.746
Receiver Earth Station Longitude (°)			124.401



Receiver Earth Station Altitude (°)	0.000
-------------------------------------	-------

This uplink  $\frac{E_b}{N_0}$  aligns with the esimated value of 10.5 dB for OFDM modulation using QPSK, given a bit error rate of  $10e^{-6}$  (Ajose, Imoize, & Obiukwu, 2018).

### 3.3.3 Case 2: High-Altitude Operations

The link analysis for the MQ-4C Tritin UAV during high-altitude operations is outlined in Table 15.

Table 15: High-Altitude Operations Link Budget

Name	Overall	Uplink	Downlink
Eb/No Ratio (dB)	7.023	10.530	9.588
Coded Eb/No Ratio (dB)	13.523	17.030	16.088
Carrier Power Density (dBW)		-205.945	-221.972
Free-Space Path Loss (dB)		-177.445	-173.772
Atmospheric Loss (dB)		-84.000	-84.000
C/No Ratio (dB)			
Bandwidth to Bit Rate Ratio (dB)		-9.031	-3.010
C/N Ratio (dB)		26.061	19.098
Central Angle (°)		0.000	0.000
Slant Range (km)		613.000	613.000
Coded C/N Ratio (dB)		26.061	19.098
Transmitter Noise Figure ()			
Transmitter Equivalent Noise Temperature (K)			
Transmitter Noise Temperature (K)			
Transmitter Combined Gain (dB)		47.059	37.000
Transmitter G/Te Ratio (dBK-1)			
Transmitter Frequency (GHz)		29.000	19.000
Transmitter Wavelength (m)		0.010	0.016
Transmitter Feeder Loss (dB)		-6.000	-6.000
Transmitter Gain (dB)		47.059	37.000
Transmitter EIRP (dBW)		55.500	35.800
Transmitter S/N (dBW)			
Transmitter Amplifier Power (dBW)		15.441	15.877
Transmitter Amplifier Gain (dB)		0.000	0.000
Transmitter Amplifier Back-Off Loss (dB)		-1.000	-1.000
Transmitter Amplifier Noise Power (dB)			
Transmitter Modulation Maximum Bit Rate (mbps)		400.000	200.000
Transmitter Modulation Data Rate (mbps)		300.000	150.000
Transmitter Modulation Bandwidth (GHz)		0.050	0.100

Table 15: High-Altitude Operations Link Budget

Name	Overall	Uplink	Downlink
Transmitter Modulation Spectral Efficiency (bits/s/Hz)		4.000	2.000
Transmitter Modulation C/N Ratio (bits/s/GHz)		19.561	12.598
Transmitter Modulation Coded C/N Ratio (bits/s/GHz)		26.061	19.098
Transmitter Modulation Eb/No Ratio (dB)		10.530	9.588
Transmitter Modulation Coded Eb/No Ratio (dB)		17.030	16.088
Transmitter Modulation Bit Error Rate ()		0.000	0.000
Transmitter Modulation Coded Bit Error Rate ()		0.000	0.000
Transmitter Modulation Roll-Off Factor ()		0.400	0.400
Transmitter Modulation Coding Rate (mbps)		0.750	0.750
Transmitter Modulation Coding Gain (dB)		6.500	6.500
Transmitter Earth Station Latitude (°)		-3.746	
Transmitter Earth Station Longitude (°)		124.401	
Transmitter Earth Station Altitude (°)		17.000	
Receiver Noise Figure ()			
Receiver Equivalent Noise Temperature (K)			
Receiver Noise Temperature (K)			
Receiver Combined Gain (dB)		37.000	47.059
Receiver G/Te Ratio (dBK-1)			
Receiver Frequency (GHz)		29.000	19.000
Receiver Wavelength (m)		0.010	0.016
Receiver Feeder Loss (dB)		-6.000	-6.000
Receiver Gain (dB)		37.000	47.059
Receiver EIRP (dBW)		46.000	55.500
Receiver S/N (dBW)			
Receiver Amplifier Power (dBW)		15.877	15.441
Receiver Amplifier Gain (dB)		0.000	0.000
Receiver Amplifier Back-Off Loss (dB)		-1.000	-1.000
Receiver Amplifier Noise Power (dB)			
Receiver Modulation Maximum Bit Rate (mbps)		400.000	200.000
Receiver Modulation Data Rate (mbps)		300.000	150.000
Receiver Modulation Bandwidth (GHz)		0.050	0.100
Receiver Modulation Spectral Efficiency (bits/s/Hz)		8.000	4.000
Receiver Modulation C/N Ratio (bits/s/GHz)		19.561	12.598
Receiver Modulation Coded C/N Ratio (bits/s/GHz)		26.061	19.098
Receiver Modulation Eb/No Ratio (dB)		10.530	9.588
Receiver Modulation Coded Eb/No Ratio (dB)		17.030	16.088
Receiver Modulation Bit Error Rate ()		0.000	0.000
Receiver Modulation Coded Bit Error Rate ()		0.000	0.000
Receiver Modulation Roll-Off Factor ()		0.400	0.400
Receiver Modulation Coding Rate (mbps)		0.750	0.750
Receiver Modulation Coding Gain (dB)		6.500	6.500
Receiver Sub-Satellite Latitude (°)		-3.746	

Table 15: High-Altitude Operations Link Budget

Name	Overall	Uplink	Downlink
Receiver Sub-Satellite Longitude (°)		124.401	
Receiver Sub-Satellite Altitude (°)		16.000	
Transmitter Sub-Satellite Latitude (°)			-3.746
Transmitter Sub-Satellite Longitude (°)			124.401
Transmitter Sub-Satellite Altitude (°)			16.000
Receiver Earth Station Latitude (°)			-3.746
Receiver Earth Station Longitude (°)			124.401
Receiver Earth Station Altitude (°)			17.000

### 3.3.4 Analysis

For both operational conditions, the link budget confirms that the proposed Kuiper constellation is feasible service provider. For low operational altitudes the system allows for a reasonable uplink  $\frac{E_b}{N_0}$  of 17 dB for a bit error rate of  $10e^{-6}$  and downlink  $\frac{E_b}{N_0}$  of 16 dB for a bit error rate of  $10e^{-5}$ . For high operational altitudes the system has a similar  $\frac{E_b}{N_0}$  profile. The system allows for constant, reliable transmission of payload and TT&C data over the full range of operations. This holds for even worst-case transmission conditions of atmospheric attenuation. Further analysis at the extent of the possible slant range is recommended, to further evaluate the feasibility of the service in worst-case conditions. The use of the Kuiper constellation satisfies all of the requirement of the system with the current antenna configurations and will therefore be fast and cheap to integrate with the RAAF's existing ground segment. Overall, it is recommended that the RAAF explore the Kuiper constellation as a internet service provider.

Word Count: 1895

## 4 Appendix

### 4.1 Equations

#### 4.1.1 Link Budget Equations

All equations referenced in this and the following section can be assumed to be sourced from Principles of Satellite Communications ([Ryan, 2004](#)) unless stated otherwise.

Table 16: Link Budget Equations

Name	Equation Number
EIRP	9-3
Carrier Power Density	9-1
Carrier-to-Noise Density	9-12
Energy per Bit-to-Noise	9-27

#### 4.1.2 Channel Coding Equations

Table 17: Link Budget Equations

Name	Equation Number
BER	6-59
Energy-per-Bit to Noise	9-28
Carrier to Noise	6-55
Bit Rate	6-34

## References

- Ahn, B., Hwang, I.-J., Kim, K.-S., Chae, S.-C., Yu, J.-W., & Lee, H. L. (2019, Dec 05). Wide-angle scanning phased array antenna using high gain pattern reconfigurable antenna elements. *Scientific Reports*, 9(1), 18391. Retrieved from <https://doi.org/10.1038/s41598-019-54120-2> doi: 10.1038/s41598-019-54120-2
- Ajose, S., Imoize, A., & Obiukwu, O. (2018, 07). Bit error rate analysis of different digital modulation schemes in orthogonal frequency division multiplexing systems. *Nigerian Journal of Technology*, 37. doi: 10.4314/njt.v37i3.23
- Al-Saegh, A., Sali, A., Mandeep, J., Ismail, A., Al-Jumaily, A., & Gomes, C. (2014, 09). Atmospheric propagation model for satellite communications. In (p. 28). doi: 10.5772/58238
- ANAO. (n.d.).
- DoD. (n.d.).
- Griethe, W. (2011, 05). Advanced broadband links for tier iii uav data communication..
- Group, M. P. (2022). Thinkom antenna selected for triton surrogate. Retrieved from <https://monch.com/thinkom-antenna-selected-for-triton-surrogate/>
- He, G., Gao, X., Sun, L., & Zhang, R. (2021, 10). A review of multibeam phased array antennas as leo satellite constellation ground station. *IEEE Access*, PP, 1-1. doi: 10.1109/ACCESS.2021.3124318
- Heiskala, J., & Terry, J. (2001). *Ofdm wireless lans: A theoretical and practical guide*. USA: Sams.
- Kramer, H. (2010). Global hawk uas (unmanned aerial system) of nasa. *EOPortal Directory*.
- Kuiper Systems, L. (2020a). Application for authority to deploy and operate a ka-band non-geostationary satellite orbit system. Retrieved from <https://docs.fcc.gov/public/attachments/FCC-20-102A1.txt>
- Kuiper Systems, L. (2020b). Application of kuiper systems llc for authority to launch and operate a non-geostationary satellite orbit system in ka-band frequencies: Technical appendix. Retrieved from <https://arstechnica.com/wp-content/uploads/2019/07/amazon-Technical-Appendix.pdf>
- Kuiper Systems, L. (2021). Request for experimental authorization: Narrative statement. Retrieved from <https://apps.fcc.gov/els/GetAtt.html?id=285359>
- Li, K., Lin, B., & Ma, J. (2019). Bit-error rate investigation of satellite-to-ground downlink optical communication employing spatial diversity and modulation techniques. *Optics Communications*, 442, 123-131. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0030401819302020> doi: <https://doi.org/10.1016/j.optcom.2019.03.012>
- NavyRecognition. (2020).
- Northrop. (2020). Mq-4c triton: Making the world's oceans smaller. *Northrop Grumman*. Notes, E. (n.d.).

- Pachler, N., del Portillo Barrios, I., Crawley, E., & Cameron, B. (2021, 06). An updated comparison of four low earth orbit satellite constellation systems to provide global broadband. In (p. 1-7). doi: 10.1109/ICCWorkshops50388.2021.9473799
- RAAF. (2020). Mq-4c triton unmanned aircraft system. *Royal Australian Air Force*. Retrieved from <https://www.airforce.gov.au/technology/aircraft/intelligence-surveillance-and-reconnaissance/mq-4c-triton-unmanned-aircraft>
- Ryan, M. (2004). *Principles of satellite communications*.
- Starlink Services, L. (2021). Petition of starlink services, llc for designation as an eligible telecommunications carrier. Retrieved from <https://www.mass.gov/doc/dtc-21-1-starlink-final-order/download>
- Stephenson, J. L. (1999). *The air refueling receiver that does not complain* (Tech. Rep.). Air University Press. Retrieved 2022-06-15, from <http://www.jstor.org/stable/resrep13763.8>
- TealGroup. (2019). Global hawk triton future next-gen hale eo/ir radars. *Military Electronics Briefing*.
- ThinKom. (2018). Ka-band satcom antenna for global broadband: Technical specifications. Retrieved from <https://www.thinkom.com/wp-content/uploads/2022/03/thinair-ka2517-datasheet.pdf>

## 4.2 Python Code Samples

```

1 import pathlib
2 from math import inf, pi
3
4 import numpy as np
5 import pandas as pd
6
7 from link_calculator.components.antennas import Amplifier, Antenna,
  ↪ ParabolicAntenna
8 from link_calculator.components.communicators import GroundStation,
  ↪ Satellite
9 from link_calculator.constants import BOLTZMANN_CONSTANT, EARTH_RADIUS
10 from link_calculator.link_budget import Link, LinkBudget
11 from link_calculator.orbits.utils import GeodeticCoordinate, Orbit
12 from link_calculator.propagation.conversions import decibel_to_watt,
  ↪ watt_to_decibel
13 from link_calculator.signal_processing.conversions import (
14     Hz_to_GHz,
15     MHz_to_GHz,
16     MHz_to_Hz,
17     mbit_to_bit,
18 )
19 from link_calculator.signal_processing.modulation import (
20     ConvolutionalCode,
21     MPhaseShiftKeying,
22 )
23
24 print("\n\n")

```

```

25
26 ABS_PATH = pathlib.Path(__file__).parent.resolve()
27
28
29 def q1():
30     sat_mod = MPhaseShiftKeying(
31         levels=8,
32         bandwidth=MHz_to_GHz(50),
33         rolloff_rate=0.3,
34     )
35
36     # Ground Station A
37     gsA_point = GeodeticCoordinate(latitude=-19.08, longitude=178.18)
38     gsA_amp = Amplifier(
39         power=40,
40     )
41     gsA_transmit = ParabolicAntenna(
42         frequency=30.5,
43         efficiency=0.55,
44         circular_diameter=1.2,
45         loss=decibel_to_watt(-1),
46         amplifier=gsA_amp,
47         modulation=sat_mod,
48     )
49     gsA_receive = ParabolicAntenna(
50         frequency=20.5,
51         efficiency=0.55,
52         circular_diameter=1.2,
53         loss=decibel_to_watt(-1),
54         amplifier=gsA_amp,
55         modulation=sat_mod,
56     )
57     gsA = GroundStation(
58         name="EarthStationA",
59         noise_figure=2.8,
60         noise_temperature=25.0,
61         ground_coordinate=gsA_point,
62         transmit=gsA_transmit,
63         receive=gsA_receive,
64     )
65
66     # Ground Station B
67     gsB_point = GeodeticCoordinate(latitude=-19.08, longitude=178.18)
68     gsB_amp = Amplifier(
69         power=80,
70     )
71     gsB_transmit = ParabolicAntenna(
72         frequency=30.5,
73         circular_diameter=0.8,
74         efficiency=0.6,
75         loss=decibel_to_watt(-1.0),
76         amplifier=gsB_amp,
77         modulation=sat_mod,
78     )
79     gsB_receive = ParabolicAntenna(

```

```

80     frequency=20.5,
81     circular_diameter=0.8,
82     efficiency=0.6,
83     loss=decibel_to_watt(-1.0),
84     amplifier=gsB_amp,
85     modulation=sat_mod,
86 )
87 gsB = GroundStation(
88     name="EarthStationB",
89     noise_figure=2.5,
90     noise_temperature=25.0,
91     ground_coordinate=gsB_point,
92     transmit=gsB_transmit,
93     receive=gsB_receive,
94 )
95
96 # Ground Station F
97 gsF_point = GeodeticCoordinate(latitude=-35.17, longitude=147.27)
98 gsF_amp = Amplifier(
99     power=500.0,
100 )
101 gsF_transmit = ParabolicAntenna(
102     frequency=30.5,
103     circular_diameter=10.5,
104     efficiency=0.65,
105     loss=decibel_to_watt(-2.5),
106     amplifier=gsF_amp,
107     modulation=sat_mod,
108 )
109 gsF_receive = ParabolicAntenna(
110     frequency=20.5,
111     circular_diameter=10.5,
112     efficiency=0.65,
113     loss=decibel_to_watt(-2.5),
114     amplifier=gsF_amp,
115     modulation=sat_mod,
116 )
117 gsF = GroundStation(
118     name="EarthStationFixed",
119     noise_figure=2.1,
120     noise_temperature=25.0,
121     transmit=gsF_transmit,
122     receive=gsF_receive,
123     ground_coordinate=gsF_point,
124 )
125
126 # Satellite
127 orbit = Orbit(orbital_radius=42164)
128 ss_point = GeodeticCoordinate(latitude=0, longitude=149.8)
129 sat_amp = Amplifier(
130     power=8.0,
131 )
132 sat_transmit = ParabolicAntenna(
133     frequency=20.5, # GHz
134     modulation=sat_mod,

```



```

135         amplifier=sat_amp,
136         circular_diameter=1.0,
137         efficiency=0.6,
138         loss=decibel_to_watt(-0.5),
139     )
140     sat_receive = ParabolicAntenna(
141         frequency=30.5, # GHz
142         modulation=sat_mod,
143         amplifier=sat_amp,
144         circular_diameter=1.0,
145         efficiency=0.6,
146         loss=decibel_to_watt(-0.5),
147     )
148     sat = Satellite(
149         name="sat",
150         noise_temperature=300,
151         noise_figure=2.5,
152         transmit=sat_transmit,
153         receive=sat_receive,
154         ground_coordinate=ss_point,
155         orbit=orbit,
156     )
157
158     gsA_uplink_upstream = Link(
159         transmitter=gsA,
160         receiver=sat,
161         atmospheric_loss=decibel_to_watt(-6.0),
162         slant_range=Link.distance(sat, gsA),
163     )
164     gsA_downlink_upstream = Link(
165         slant_range=Link.distance(sat, gsF),
166         transmitter=sat,
167         receiver=gsF,
168         atmospheric_loss=decibel_to_watt(-6.0),
169     )
170     gsA_upstream_budget = LinkBudget(
171         uplink=gsA_uplink_upstream, downlink=gsA_downlink_upstream
172     )
173     gsA_upstream_summary = gsA_upstream_budget.summary()
174     gsA_upstream_summary.rename(
175         columns={"name": "Earth Station A Upstream"}, inplace=True
176     )
177     gsA_upstream_summary.index = (
178         gsA_upstream_summary.index + " (" + gsA_upstream_summary.pop("unit"
179         ↪ ) + ")"
180     )
181     gsA_upstream_summary.to_csv(
182         f"{ABS_PATH}/output/Q1EarthStationAUpstream.csv", float_format="
183         ↪ {:.3f}"
184     ).format
185     )
186     print(gsA_upstream_summary)
187
188     gsA_uplink_downstream = Link(
189         transmitter=gsF,
190         receiver=sat,

```

```

188     atmospheric_loss=decibel_to_watt(-6.0),
189     slant_range=Link.distance(sat, gsF),
190 )
191 gsA_downlink_downstream = Link(
192     slant_range=Link.distance(sat, gsA),
193     transmitter=sat,
194     receiver=gsA,
195     atmospheric_loss=decibel_to_watt(-6.0),
196 )
197 gsA_downstream_budget = LinkBudget(
198     uplink=gsA_uplink_downstream, downlink=gsA_downlink_downstream
199 )
200 gsA_downstream_summary = gsA_downstream_budget.summary()
201 gsA_downstream_summary.rename(
202     columns={"name": "Earth Station A Downstream"}, inplace=True
203 )
204 gsA_downstream_summary.index = (
205     gsA_downstream_summary.index + " (" + gsA_downstream_summary.pop("
↪ unit") + ")"
206 )
207 gsA_downstream_summary.to_csv(
208     f"{ABS_PATH}/output/Q1EarthStationADownstream.csv",
209     float_format="{:,.3f}".format,
210 )
211 print(gsA_downstream_summary)
212
213 gsB_uplink_upstream = Link(
214     transmitter=gsB,
215     receiver=sat,
216     atmospheric_loss=decibel_to_watt(-6.0),
217     slant_range=Link.distance(sat, gsB),
218 )
219 gsB_downlink_upstream = Link(
220     slant_range=Link.distance(sat, gsF),
221     transmitter=sat,
222     receiver=gsF,
223     atmospheric_loss=decibel_to_watt(-6.0),
224 )
225 gsB_upstream_budget = LinkBudget(
226     uplink=gsB_uplink_upstream, downlink=gsB_downlink_upstream
227 )
228 gsB_upstream_summary = gsB_upstream_budget.summary()
229 gsB_upstream_summary.rename(
230     columns={"name": "Earth Station B Upstream"}, inplace=True
231 )
232 gsB_upstream_summary.index = (
233     gsB_upstream_summary.index + " (" + gsB_upstream_summary.pop("unit"
↪ ) + ")"
234 )
235 gsB_upstream_summary.to_csv(
236     f"{ABS_PATH}/output/Q1EarthStationBUpstream.csv", float_format="
↪ {:,.3f}".format
237 )
238 print(gsB_upstream_summary)
239

```

```

240     gsB_uplink_downstream = Link(
241         transmitter=gsF,
242         receiver=sat,
243         atmospheric_loss=decibel_to_watt(-6.0),
244         slant_range=Link.distance(sat, gsF),
245     )
246     gsB_downlink_downstream = Link(
247         slant_range=Link.distance(sat, gsB),
248         transmitter=sat,
249         receiver=gsB,
250         atmospheric_loss=decibel_to_watt(-6.0),
251     )
252     gsB_downstream_budget = LinkBudget(
253         uplink=gsB_uplink_downstream, downlink=gsB_downlink_downstream
254     )
255     gsB_downstream_summary = gsB_downstream_budget.summary()
256     gsB_downstream_summary.rename(
257         columns={"name": "Earth Station B Downstream"}, inplace=True
258     )
259     gsB_downstream_summary.index = (
260         gsB_downstream_summary.index + " (" + gsB_downstream_summary.pop("
↪ unit") + ")"
261     )
262     gsB_downstream_summary.to_csv(
263         f"{ABS_PATH}/output/Q1EarthStationBDownstream.csv",
264         float_format="{:,.3f}".format,
265     )
266
267
268 def q2():
269     def link_eb_no(overall_eb_no: float, link_eb_no) -> float:
270         return (overall_eb_no * link_eb_no) / (link_eb_no - overall_eb_no)
271
272     uplink_eb_no = decibel_to_watt(33.2)
273     min_bit_error_rate = 1e-9
274     min_data_rate = mbit_to_bit(60)
275
276     codes = {
277         "7-8": {"code_rate": 7 / 8, "code_gain": decibel_to_watt(2.5)},
278         "3-4": {"code_rate": 3 / 4, "code_gain": decibel_to_watt(3)},
279         "1-2": {"code_rate": 1 / 2, "code_gain": decibel_to_watt(3.5)},
280     }
281     results = {}
282     for name, params in codes.items():
283         print(decibel_to_watt(params["code_gain"]))
284         code = ConvolutionalCode(
285             coding_rate=params["code_rate"], coding_gain=params["code_gain"]
↪ ]
286         )
287         mod = MPhaseShiftKeying(
288             levels=8,
289             bandwidth=MHz_to_GHz(50),
290             rolloff_rate=0.3,
291             code=code,
292             eb_no=uplink_eb_no,

```

```

293         )
294         results[name] = {"data_rate": mod.data_rate, "mod": mod}
295         summary = mod.summary()
296         summary.rename(columns={"name": f"{name} Modulation Summary"},
↳ inplace=True)
297         print(summary)
298         summary.index = summary.index + " (" + summary.pop("unit") + ")"
299         summary.to_csv(
300             f"{ABS_PATH}/output/Q2{name}Modulation.csv", float_format="
↳ {:, .3f}" .format
301         )
302
303     best_code = min(
304         results,
305         key=lambda x: results[x]["data_rate"]
306         if results[x]["data_rate"] > min_data_rate
307         else inf,
308     )
309     best_mod = results[best_code]["mod"]
310
311     overall_mod = MPhaseShiftKeying(
312         levels=8,
313         bit_error_rate=min_bit_error_rate,
314         bandwidth=MHz_to_GHz(50),
315         rolloff_rate=0.3,
316         code=best_mod.code,
317     )
318     overall_summary = overall_mod.summary()
319     overall_summary.rename(
320         columns={"name": "Overall Link Modulation Summary"}, inplace=True
321     )
322     overall_summary.index = (
323         overall_summary.index + " (" + overall_summary.pop("unit") + ")"
324     )
325     print(overall_summary)
326     overall_summary.to_csv(
327         f"{ABS_PATH}/output/Q2OverallModulation.csv", float_format="{:, .3f}
↳ " .format
328     )
329
330     downlink_eb_no = link_eb_no(overall_mod.eb_no, uplink_eb_no)
331     downlink_mod = MPhaseShiftKeying(
332         levels=8,
333         bandwidth=MHz_to_GHz(50),
334         eb_no=downlink_eb_no,
335         rolloff_rate=0.3,
336         code=best_mod.code,
337     )
338     downlink_summary = downlink_mod.summary()
339     downlink_summary.index = (
340         downlink_summary.index + " (" + downlink_summary.pop("unit") + ")"
341     )
342     print(downlink_summary)
343     downlink_summary.to_csv(
344         f"{ABS_PATH}/output/Q2DownlinkModulation.csv", float_format="{:, .3f

```

```

345     ↪ }".format
346     )
347
348 def inches_to_m(value) -> float:
349     return 0.0254 * value
350
351
352 def q3():
353     """
354     TODO
355     - finalise parameters
356     - bit error rate
357     - add signal to noise to output
358     """
359     locs = {
360         "LowAltitudeOperations": GeodeticCoordinate(
361             latitude=-3.74603, longitude=124.401, altitude=0
362         ), # 15000km outside of RAAF Base Tindal
363         "HighAltitudeOperations": GeodeticCoordinate(
364             latitude=-3.74603, longitude=124.401, altitude=17
365         ),
366     }
367
368     for scenario, uav_loc in locs.items():
369         code = ConvolutionalCode(coding_rate=3 / 4, coding_gain=
370     ↪ decibel_to_watt(6.5))
371         triton_transmit_mod = MPhaseShiftKeying(
372             levels=4,
373             bandwidth=MHz_to_GHz(50),
374             bit_rate=mbit_to_bit(400),
375             spectral_efficiency=4, # bit/s/Hz
376             bit_error_rate=1e-6,
377             code=code,
378             rolloff_rate=0.4,
379         )
380         triton_transmit_amp = Amplifier(power=35, loss=decibel_to_watt(-1))
381         triton_transmit = Antenna(
382             eirp=decibel_to_watt(55.5),
383             cross_sect_area=inches_to_m(24.2) * inches_to_m(22.4),
384             amplifier=triton_transmit_amp,
385             frequency=29,
386             modulation=triton_transmit_mod,
387             loss=decibel_to_watt(-6),
388         )
389         triton_receive_mod = MPhaseShiftKeying(
390             levels=4,
391             bandwidth=MHz_to_GHz(100),
392             bit_rate=mbit_to_bit(200),
393             spectral_efficiency=4, # bit/s/Hz
394             bit_error_rate=1e-5,
395             code=code,
396             rolloff_rate=0.4,
397         )
398         triton_receive_amp = Amplifier(power=35, loss=decibel_to_watt(-1))

```

```

398     triton_receive = Antenna(
399         eirp=decibel_to_watt(55.5),
400         cross_sect_area=inches_to_m(30.6) * inches_to_m(32.4),
401         amplifier=triton_receive_amp,
402         frequency=19,
403         modulation=triton_receive_mod,
404         loss=decibel_to_watt(-6),
405     )
406     triton = GroundStation(
407         name="MQ-4C Triton",
408         transmit=triton_transmit,
409         receive=triton_receive,
410         ground_coordinate=uav_loc,
411     )
412
413     ss_point = GeodeticCoordinate(latitude=-3.74603, longitude=124.401,
414 ↪ altitude=16)
415     kuiper_transmit_mod = MPhaseShiftKeying(
416         levels=4,
417         bandwidth=MHz_to_GHz(100),
418         bit_error_rate=1e-5,
419         bit_rate=mbit_to_bit(200),
420         code=code,
421         rolloff_rate=0.4,
422     )
423     kuiper_transmit_amp = Amplifier(
424         power=38.7,
425         loss=decibel_to_watt(-1),
426     )
427     kuiper_transmit = Antenna(
428         cross_sect_area=pi * 1.6**2,
429         eirp=decibel_to_watt(35.8),
430         modulation=kuiper_transmit_mod,
431         gain=decibel_to_watt(37),
432         frequency=19,
433         amplifier=kuiper_transmit_amp,
434         loss=decibel_to_watt(-6),
435     )
436     kuiper_receive_mod = MPhaseShiftKeying(
437         levels=4,
438         bit_rate=mbit_to_bit(400),
439         bit_error_rate=1e-6,
440         bandwidth=MHz_to_GHz(50),
441         code=code,
442         rolloff_rate=0.4,
443     )
444     kuiper_receive_amp = Amplifier(
445         power=38.7,
446         loss=decibel_to_watt(-1),
447     )
448     kuiper_receive = Antenna(
449         cross_sect_area=pi * 1.6**2,
450         eirp=decibel_to_watt(46.0),
451         gain=decibel_to_watt(37),
452         loss=decibel_to_watt(-6),

```

```

452         frequency=29,
453         modulation=kuiper_receive_mod,
454         amplifier=kuiper_receive_amp,
455     )
456     orbit = Orbit(orbital_radius=630 + EARTH_RADIUS)
457     kuiper = Satellite(
458         name="KuiperSat-1",
459         orbit=orbit,
460         transmit=kuiper_transmit,
461         receive=kuiper_receive,
462         ground_coordinate=ss_point,
463     )
464
465     uplink = Link(
466         transmitter=triton,
467         receiver=kuiper,
468         atmospheric_loss=decibel_to_watt(-84.0),
469         slant_range=Link.distance(kuiper, triton)
470         - triton.ground_coordinate.altitude,
471         # eb_no=decibel_to_watt(10.5),
472     )
473     downlink = Link(
474         transmitter=kuiper,
475         receiver=triton,
476         atmospheric_loss=decibel_to_watt(-84.0),
477         slant_range=Link.distance(kuiper, triton)
478         - triton.ground_coordinate.altitude,
479         # eb_no=decibel_to_watt(10.5),
480     )
481     link_budget = LinkBudget(uplink=uplink, downlink=downlink)
482     link_summary = link_budget.summary()
483     link_summary.index = link_summary.index + " (" + link_summary.pop("
↪ unit") + ")"
484     link_summary.to_csv(
485         f"{ABS_PATH}/output/Q3Tritan{scenario}.csv", float_format="
↪ {:.3f}".format
486     )
487     print(link_summary)
488
489
490 q1()
491 q2()
492 q3()

```

Listing 1: main.py

```

1 import numpy as np
2 import pandas as pd
3
4 from link_calculator.components.communicators import (
5     Communicator,
6     GroundStation,
7     Satellite,
8 )
9 from link_calculator.constants import BOLTZMANN_CONSTANT

```

```

10 from link_calculator.orbits.utils import slant_range
11 from link_calculator.propagation.conversions import watt_to_decibel
12 from link_calculator.signal_processing.conversions import GHz_to_Hz
13
14
15 class Link:
16     def __init__(
17         self,
18         transmitter: Communicator,
19         receiver: Communicator,
20         slant_range: float = None,
21         atmospheric_loss: float = 1,
22         path_loss: float = None,
23         min_elevation: float = 0,
24         transmitter_eirp: float = None,
25         receiver_carrier_power: float = None,
26         noise_temperature: float = None,
27         noise_power: float = None,
28         noise_density: float = None,
29         carrier_to_noise_density: float = None,
30         carrier_to_noise: float = None,
31         bandwidth_to_bit_rate: float = None,
32         eb_no: float = None,
33     ):
34         """
35
36         Parameters
37         -----
38
39         slant_range (float, km): slant_range between the transmit and
↪ receive antennas
40         noise_temperature (float, Kelvin): the temperature of the
↪ environment
41
42         """
43         self._transmitter = transmitter
44         self._receiver = receiver
45         self._slant_range = slant_range
46         self._atmospheric_loss = atmospheric_loss
47         self._path_loss = path_loss
48         self._min_elevation = min_elevation
49         self._transmitter_eirp = transmitter_eirp
50         self._receiver_carrier_power = receiver_carrier_power
51         self._noise_temperature = noise_temperature
52         self._carrier_to_noise_density = carrier_to_noise_density
53         self._carrier_to_noise = carrier_to_noise
54         self._bandwidth_to_bit_rate = bandwidth_to_bit_rate
55         self._eb_no = eb_no
56         self._noise_power = noise_power
57         self._noise_density = noise_density
58         self.propagate_calculations()
59
60         if self._transmitter.transmit.modulation is not None:
61             self._transmitter.transmit.modulation.eb_no = self.eb_no
62         if self._receiver.receive is not None:

```



```

63         self._receiver.receive.power_density = (
64             self.receiver.receive.power_density_eirp(
65                 self._slant_range, self._atmospheric_loss
66             )
67         )
68
69         if self._receiver.receive.modulation is not None:
70             self._receiver.receive.modulation.carrier_power = (
71                 self.receiver_carrier_power
72             )
73             self._receiver.receive.modulation.eb_no = self.eb_no
74
75     @property
76     def carrier_to_noise_density(self) -> float:
77         if self._carrier_to_noise_density is None:
78             if self.receiver.equiv_noise_temp is not None:
79                 self._carrier_to_noise_density = (
80                     self.receiver.combined_gain * self.
81     ↪ receiver_carrier_power
82                     ) / (BOLTZMANN_CONSTANT * self.receiver.equiv_noise_temp)
83             elif self.receiver.gain_to_equiv_noise_temp is not None:
84                 self._carrier_to_noise_density = (
85                     self.transmitter.transmit.eirp
86                     * self.path_loss
87                     * self.atmospheric_loss
88                     * self.receiver.receive.loss
89                     * self.receiver.gain_to_equiv_noise_temp
90                     / BOLTZMANN_CONSTANT
91                 )
92             return self._carrier_to_noise_density
93
94     @property
95     def eb_no(self) -> float:
96         if self._eb_no is None:
97             if self._carrier_to_noise_density is not None:
98                 self._eb_no = (
99                     self.carrier_to_noise_density
100                     / self.transmitter.transmit.modulation.bit_rate
101                 )
102             elif self._carrier_to_noise is not None:
103                 self._eb_no = (
104                     self.carrier_to_noise
105                     * GHz_to_Hz(self.transmitter.transmit.modulation.
106     ↪ bandwidth)
107                     * self.transmitter.transmit.modulation.bit_rate
108                 )
109             if self.transmitter.transmit.modulation.eb_no is not None:
110                 if self.transmitter.transmit.modulation.code is not None:
111                     self._eb_no = self.transmitter.transmit.modulation.
112     ↪ eb_no_coded
113                 else:
114                     self._eb_no = self.transmitter.transmit.modulation.
115     ↪ eb_no
116             return self._eb_no

```

```

114     @property
115     def carrier_to_noise(self) -> float:
116         if self._carrier_to_noise is None:
117             if self._eb_no is not None and self._bandwidth_to_bit_rate is
118                 ↪ not None:
119                 self._carrier_to_noise = self.eb_no / self.
120                 ↪ bandwidth_to_bit_rate
121             return self._carrier_to_noise
122
123     @property
124     def bandwidth_to_bit_rate(self) -> float:
125         if self._bandwidth_to_bit_rate is None:
126             self._bandwidth_to_bit_rate = (
127                 GHz_to_Hz(self.transmitter.transmit.modulation.bandwidth)
128                 / self.transmitter.transmit.modulation.bit_rate
129             )
130             return self._bandwidth_to_bit_rate
131
132     @property
133     def receiver_carrier_power(self) -> float:
134         if self._receiver_carrier_power is None:
135             self._receiver_carrier_power = (
136                 self.transmitter.transmit.eirp * self.path_loss * self.
137                 ↪ atmospheric_loss
138             )
139             return self._receiver_carrier_power
140
141     @property
142     def central_angle(self) -> float:
143         if (
144             self.transmitter.ground_coordinate is not None
145             and self.receiver.ground_coordinate is not None
146         ):
147             return self.transmitter.ground_coordinate.central_angle(
148                 self.receiver.ground_coordinate
149             )
150             return None
151
152     @staticmethod
153     def distance(satellite: Satellite, ground_station: GroundStation) ->
154         ↪ float:
155         gamma = satellite.ground_coordinate.central_angle(
156             ground_station.ground_coordinate
157         )
158         return slant_range(satellite.orbit.orbital_radius, gamma)
159
160     @property
161     def slant_range(self) -> float:
162         return self._slant_range
163
164     @property
165     def path_loss(self) -> float:
166         """
167         Calculate the free space loss between two antennas

```

```

165     Parameters
166     -----
167     slant_range (float, km): slant_range between the transmit and
↪ receive antennas
168     wavelength (float, m): frequency of the transmitter
169
170     Returns
171     -----
172     path_loss (float, )
173
174     """
175     if self._path_loss is None:
176         self._path_loss = (
177             self.transmitter.transmit.wavelength
178             / (4 * np.pi * self.slant_range * 1000)
179             ) ** 2
180     return self._path_loss
181
182 @property
183 def noise_power(self) -> float:
184     """
185     Returns
186     -----
187     noise_power (float, ): sum of the input noise power and the
↪ noise power
188         added by the amplifier
189     """
190     if self._noise_power is None:
191         if self._noise_density is not None:
192             self._noise_power = self.noise_density * GHz_to_Hz(
193                 self.transmitter.transmit.modulation.bandwidth
194             )
195     return self._noise_power
196
197 @property
198 def noise_density(self) -> float:
199     """
200     Calculate the noise density of the system
201
202     Returns
203     -----
204     noise_density (float, W/Hz): the total noise power, normalised
↪ to a 1-Hz bandwidth
205     """
206     if self._noise_density is None:
207         if self._noise_temperature is not None:
208             self._noise_density = BOLTZMANN_CONSTANT * self.
↪ noise_temperature
209     return self._noise_density
210
211 @property
212 def noise_temperature(self) -> float:
213     """
214
215     Returns

```

```

216         -----
217         noise_temperature (float, K): ambient temperature of the
↪ environment
218         """
219         if self._noise_temperature is None:
220             if self._noise_power is not None:
221                 self._noise_temperature = (
222                     self.noise_power
223                     / GHz_to_Hz(self.transmitter.transmit.modulation.
↪ bandwidth)
224                     ) / BOLTZMANN_CONSTANT
225         return self._noise_temperature
226
227     @property
228     def receiver(self) -> Communicator:
229         return self._receiver
230
231     @property
232     def transmitter(self) -> Communicator:
233         return self._transmitter
234
235     @property
236     def atmospheric_loss(self) -> float:
237         return self._atmospheric_loss
238
239     def summary(self) -> pd.DataFrame:
240         summary = pd.DataFrame.from_records(
241             [
242                 {
243                     "name": "Carrier Power Density",
244                     "unit": "dBW",
245                     "value": watt_to_decibel(self.receiver_carrier_power),
246                 },
247                 {
248                     "name": "Free-Space Path Loss",
249                     "unit": "dB",
250                     "value": watt_to_decibel(self.path_loss),
251                 },
252                 {
253                     "name": "Atmospheric Loss",
254                     "unit": "dB",
255                     "value": watt_to_decibel(self.atmospheric_loss),
256                 },
257                 {
258                     "name": "C/No Ratio",
259                     "unit": "dB",
260                     "value": watt_to_decibel(self.carrier_to_noise_density)
↪ ,
261                 },
262                 {
263                     "name": "Eb/No Ratio",
264                     "unit": "dB",
265                     "value": watt_to_decibel(self.eb_no),
266                 },
267                 {

```

```

268         "name": "Bandwidth to Bit Rate Ratio",
269         "unit": "dB",
270         "value": watt_to_decibel(self.bandwidth_to_bit_rate),
271     },
272     {
273         "name": "C/N Ratio",
274         "unit": "dB",
275         "value": watt_to_decibel(self.carrier_to_noise),
276     },
277     {"name": "Central Angle", "unit": " ", "value": self.
↪ central_angle},
278     {"name": "Slant Range", "unit": "km", "value": self.
↪ slant_range},
279     ]
280 )
281 receiver = self.receiver.summary()
282 receiver.index = "Receiver " + receiver.index
283
284 transmitter = self.transmitter.summary()
285 transmitter.index = "Transmitter " + transmitter.index
286
287 summary.set_index("name", inplace=True)
288 summary = pd.concat([summary, transmitter, receiver])
289 return summary
290
291 def propagate_calculations(self) -> float:
292     for _ in range(3):
293         for var in type(self).__dict__:
294             if not callable(var):
295                 getattr(self, var)
296
297
298 class LinkBudget:
299     def __init__(
300         self,
301         uplink: Link,
302         downlink: Link,
303     ):
304         self._uplink = uplink
305         self._downlink = downlink
306
307     @property
308     def eb_no(self) -> float:
309         return (self.uplink.eb_no * self.downlink.eb_no) / (
310             self.uplink.eb_no + self._downlink.eb_no
311         )
312
313     @property
314     def uplink(self) -> Link:
315         return self._uplink
316
317     @property
318     def downlink(self) -> Link:
319         return self._downlink
320

```

```

321     def summary(self) -> pd.DataFrame:
322         summary = pd.DataFrame.from_records(
323             [
324                 {
325                     "name": "Eb/No Ratio",
326                     "unit": "dB",
327                     "value": watt_to_decibel(self.eb_no),
328                 },
329             ]
330         )
331
332         uplink = self.uplink.summary()
333         uplink.rename(columns={"value": "Uplink", "unit": "Uplink unit"},
334 ↪ inplace=True)
335         downlink = self.downlink.summary()
336         downlink.rename(
337             columns={"value": "Downlink", "unit": "Downlink unit"}, inplace
338 ↪ =True
339         )
340         summary.set_index("name", inplace=True)
341         summary.rename(columns={"value": "Overall"}, inplace=True)
342
343         # Merge uplink and downlink summaries
344         summary = pd.concat([summary, uplink, downlink], axis=1)
345
346         # Merge unit columns
347         summary["unit"] = summary["unit"].combine_first(
348             summary["Uplink unit"].combine_first(summary["Downlink unit"])
349         )
350         summary.drop(columns=["Downlink unit", "Uplink unit"], inplace=True
351 ↪ )
352
353         # Clean and reformat columns
354         summary = summary[
355             [
356                 not (
357                     idx.startswith("Transmitter Receive")
358                     or idx.startswith("Receiver Transmit")
359                 )
360                 for idx in summary.index
361             ]
362         ]
363         summary.rename(
364             index={
365                 k: k.replace("Receiver Receive", "Receiver")
366                 if "Receiver Receive" in k
367                 else k
368                 for k in summary.index
369             },
370             inplace=True,
371         )
372         summary.rename(
373             index={
374                 k: k.replace("Transmitter Transmit", "Transmitter")
375                 if "Transmitter Transmit" in k

```

```

373         else k
374         for k in summary.index
375     },
376     inplace=True,
377 )
378 return summary

```

Listing 2: link<sub>budget</sub>.py

```

1 from math import sqrt
2
3 import pandas as pd
4
5 from link_calculator.components.antennas import Amplifier, Antenna
6 from link_calculator.constants import BOLTZMANN_CONSTANT, EARTH_MU,
   ↪ EARTH_RADIUS
7 from link_calculator.orbits.utils import GeodeticCoordinate, Orbit
8 from link_calculator.propagation.conversions import watt_to_decibel
9 from link_calculator.signal_processing.conversions import GHz_to_Hz
10
11
12 class Communicator:
13     def __init__(
14         self,
15         name: str,
16         transmit: Antenna,
17         receive: Antenna,
18         ground_coordinate: GeodeticCoordinate = None,
19         combined_gain: float = None,
20         noise_figure: float = None,
21         noise_temperature: float = None,
22         noise_density: float = None,
23         gain_to_equiv_noise_temp: float = None,
24         equiv_noise_temp: float = None,
25     ):
26         self._name = name
27         self._transmit = transmit
28         self._receive = receive
29         self._ground_coordinate = ground_coordinate
30         self._noise_figure = noise_figure
31         self._noise_density = noise_density
32         self._noise_temperature = noise_temperature
33         self._combined_gain = combined_gain
34         self._gain_to_equiv_noise_temp = gain_to_equiv_noise_temp
35         self._equiv_noise_temp = equiv_noise_temp
36         self.propagate_calculations()
37
38     @property
39     def noise_figure(self) -> float:
40         """
41         Calculate the noise figure of the device
42
43         Returns
44         -----
45         noise_figure (float, ): the ratio of the S/N ratio at the

```

```

46     ↪ input to the
        S/N ratio at the output. Measure of the relative increase
47     ↪ in noise
        power compared to increase in signal power
48     """
49     if self._noise_figure is None:
50         if self.receive.signal_to_noise is not None:
51             self.noise_figure = (
52                 self.receive.signal_to_noise / self.transmit.
53                 ↪ signal_to_noise
                    )
54         elif self._noise_temperature is not None:
55             self._noise_figure = 1 + (
56                 self.equiv_noise_temp / self.noise_temperature
57             )
58         return self._noise_figure
59
60     @property
61     def output_noise_power(self) -> float:
62         return self.receive.amplifier.gain * (
63             self.receive.noise_power + self.receive.amplifier.noise_power
64         )
65
66     @property
67     def receive_noise_power(self) -> float:
68         return (
69             BOLTZMANN_CONSTANT
70             * self.noise_temperature
71             * GHz_to_Hz(self.receive.modulation.bandwidth)
72         )
73
74     @property
75     def equiv_noise_temp(self):
76         """
77         TODO
78         """
79         if self._equiv_noise_temp is None:
80             if self._noise_temperature is not None:
81                 self._equiv_noise_temp = self.noise_temperature * (
82                     self.noise_figure - 1
83                 )
84             return self._equiv_noise_temp
85
86     @equiv_noise_temp.setter
87     def equiv_noise_temp(self, value):
88         """
89         TODO
90         """
91         self._equiv_noise_temp = value
92
93     @property
94     def combined_gain(self) -> float:
95         if self._combined_gain is None:
96             if self.receive.amplifier is not None:
97                 self._combined_gain = self.receive.amplifier.gain * self.

```



```

108         ↪ receive.gain
109         else:
110             self._combined_gain = self.receive.gain
111         return self._combined_gain
112
113     @property
114     def gain_to_equiv_noise_temp(self) -> float:
115         if self._gain_to_equiv_noise_temp is None:
116             if self._equiv_noise_temp is not None:
117                 self._gain_to_equiv_noise_temp = (
118                     self.combined_gain / self.equiv_noise_temp
119                 )
120             elif self._noise_density is not None:
121                 self._gain_to_equiv_noise_temp = (
122                     self.carrier_power * BOLTZMANN_CONSTANT
123                 ) / (self.noise_density * self.receive_carrier_power)
124         return self._gain_to_equiv_noise_temp
125
126     @property
127     def ground_coordinate(self) -> GeodeticCoordinate:
128         return self._ground_coordinate
129
130     @property
131     def receive(self) -> Antenna:
132         return self._receive
133
134     @property
135     def transmit(self) -> Antenna:
136         return self._transmit
137
138     @property
139     def noise_temperature(self) -> float:
140         return self._noise_temperature
141
142     @property
143     def noise_density(self) -> float:
144         return self._noise_density
145
146     def summary(self) -> pd.DataFrame:
147         summary = pd.DataFrame.from_records(
148             [
149                 {"name": "Noise Figure", "unit": "", "value": self.
150                 ↪ noise_figure},
151                 {
152                     "name": "Equivalent Noise Temperature",
153                     "unit": "K",
154                     "value": self.equiv_noise_temp,
155                 },
156                 {
157                     "name": "Noise Temperature",
158                     "unit": "K",
159                     "value": self.noise_temperature,
160                 },
161                 {
162                     "name": "Combined Gain",

```

```

151         "unit": "dB",
152         "value": watt_to_decibel(self.combined_gain),
153     },
154     {
155         "name": "G/Te Ratio",
156         "unit": "dBK-1",
157         "value": watt_to_decibel(self.gain_to_equiv_noise_temp)
158     },
159 ]
160 )
161 transmitter = self.transmit.summary()
162 transmitter.index = "Transmit " + transmitter.index
163
164 receiver = self.receive.summary()
165 receiver.index = "Receive " + receiver.index
166
167 summary.set_index("name", inplace=True)
168 summary = pd.concat([summary, transmitter, receiver])
169 return summary
170
171 def propagate_calculations(self) -> float:
172     for _ in range(3):
173         for var in type(self).__dict__:
174             getattr(self, var)
175
176
177 class GroundStation(Communicator):
178     def __init__(
179         self,
180         name: str,
181         transmit: Antenna,
182         receive: Antenna,
183         ground_coordinate: GeodeticCoordinate = None,
184         gain_to_equiv_noise_temp: float = None,
185         combined_gain: float = None,
186         noise_figure: float = None,
187         noise_temperature: float = None,
188         equiv_noise_temp: float = None,
189     ):
190         """
191
192         Parameters
193         -----
194             name (str,): name of the ground station
195             latitude (str, deg): the latitude of the groundstation
196             longitude (str, deg): the longitude of the groundstation
197             altitude (str, km): the altitude of the groundstation above sea
198         ↪ level
199         """
200         super().__init__(
201             name=name,
202             transmit=transmit,
203             receive=receive,
204             ground_coordinate=ground_coordinate,

```

```

204         gain_to_equiv_noise_temp=gain_to_equiv_noise_temp,
205         noise_figure=noise_figure,
206         noise_temperature=noise_temperature,
207         equiv_noise_temp=equiv_noise_temp,
208     )
209
210     def summary(self) -> pd.DataFrame:
211         summary = super().summary()
212         if self.ground_coordinate is not None:
213             coordinate = self.ground_coordinate.summary()
214             coordinate.index = "Earth Station " + coordinate.index
215             summary = pd.concat([summary, coordinate])
216         return summary
217
218
219 class Satellite(Communicator):
220     def __init__(
221         self,
222         name: str,
223         transmit: Antenna,
224         receive: Antenna,
225         orbit: Orbit = None,
226         ground_coordinate: GeodeticCoordinate = None,
227         gain_to_equiv_noise_temp: float = None,
228         combined_gain: float = None,
229         noise_figure: float = None,
230         noise_temperature: float = None,
231         equiv_noise_temp: float = None,
232     ):
233         self._orbit = orbit
234         super().__init__(
235             name=name,
236             transmit=transmit,
237             receive=receive,
238             ground_coordinate=ground_coordinate,
239             combined_gain=combined_gain,
240             gain_to_equiv_noise_temp=gain_to_equiv_noise_temp,
241             noise_figure=noise_figure,
242             noise_temperature=noise_temperature,
243             equiv_noise_temp=equiv_noise_temp,
244         )
245
246     def velocity(self, orbital_radius: float, mu: float = EARTH_MU) ->
247     ↪ float:
248         """
249         Calculate the velocity of a satellite in orbit according to Kepler'
250         ↪ s second law
251
252         Parameters
253         -----
254             semi_major_axis (float, km): The semi-major axis of the orbit
255             orbital_radius (float, km): distance from the centre of
256             ↪ mass to the satellite
257             mu (float, optional): Kepler's gravitational constant

```

```

256     Returns
257     -----
258     velocity (float, km/s): the orbit speed of the satellite
259     """
260     return sqrt(mu * (2 / orbital_radius - 1 / self.semi_major_axis))
261
262     @property
263     def semi_major_axis(self) -> float:
264         if self.orbit is not None:
265             return self.orbit.semi_major_axis
266         return None
267
268     @property
269     def orbit(self) -> float:
270         return self._orbit
271
272     def summary(self) -> pd.DataFrame:
273         summary = super().summary()
274         if self.ground_coordinate is not None:
275             coordinate = self.ground_coordinate.summary()
276             coordinate.index = "Sub-Satellite " + coordinate.index
277             summary = pd.concat([summary, coordinate])
278         return summary

```

Listing 3: components/communicators.py

```

1 from math import cos, degrees, exp, pi, radians, sin
2
3 import numpy as np
4 import pandas as pd
5
6 from link_calculator.constants import BOLTZMANN_CONSTANT
7 from link_calculator.propagation.conversions import (
8     frequency_to_wavelength,
9     watt_to_decibel,
10    wavelength_to_frequency,
11 )
12 from link_calculator.signal_processing.modulation import Modulation
13
14
15 class Amplifier:
16     def __init__(
17         self, power: float, gain: float = 1, loss: float = 1, noise_power:
18         ↪ float = None
19     ):
20         """
21         Parameters
22         -----
23         loss (float, ): losses in the amplifier (e.g. back-off loss)
24         power (float, W): the total output amplifier power
25         """
26         self._power = power
27         self._gain = gain
28         self._loss = loss
29         self._noise_power = noise_power

```

```

29
30 @property
31 def power(self) -> float:
32     """
33     TODO
34     Returns
35     -----
36     power (float, W): the total output amplifier power
37     """
38     return self._power
39
40 @property
41 def gain(self) -> float:
42     """
43     TODO
44     Returns
45     -----
46     """
47     return self._gain
48
49 @property
50 def loss(self) -> float:
51     """
52     TODO
53     Returns
54     -----
55     """
56     return self._loss
57
58 @property
59 def noise_power(self) -> float:
60     """
61     TODO
62     Returns
63     -----
64     """
65     return self._noise_power
66
67 def summary(self) -> pd.DataFrame:
68     summary = pd.DataFrame.from_records(
69         [
70             {
71                 "name": "Power",
72                 "unit": "dBW",
73                 "value": watt_to_decibel(self.power),
74             },
75             {
76                 "name": "Gain",
77                 "unit": "dB",
78                 "value": watt_to_decibel(self.gain),
79             },
80             {
81                 "name": "Back-Off Loss",
82                 "unit": "dB",
83                 "value": watt_to_decibel(self.loss),

```

```

84         },
85         {
86             "name": "Noise Power",
87             "unit": "dB",
88             "value": watt_to_decibel(self.noise_power),
89         },
90     ]
91 )
92 summary.set_index("name", inplace=True)
93 return summary
94
95
96 class Antenna:
97     def __init__(
98         self,
99         gain: float = None,
100         loss: float = 1,
101         eirp: float = None,
102         frequency: float = None,
103         wavelength: float = None,
104         effective_aperture: float = None,
105         cross_sect_area: float = None,
106         cross_sect_diameter: float = None,
107         half_beamwidth: float = None,
108         efficiency: float = None,
109         roughness_factor: float = None,
110         carrier_to_noise: float = None,
111         signal_to_noise: float = None,
112         carrier_power: float = None,
113         modulation: Modulation = None,
114         combined_loss: float = None,
115         amplifier: Amplifier = None,
116         gain_to_noise_temperature=None,
117         power_density: float = None,
118     ):
119         """
120         Instantiate an Antenna object
121
122         Parameters
123         -----
124             name (str): Name of antenna
125             gain (float, ): ratio of maximum power densiry to that of an
126 ↪ isotropic radiator
127                 at the same distance in the direction of the receiving
128 ↪ antenna
129             loss (float, ): coupling loss between transmitter and antenna
130                 in the range [0, 1]
131             frequency (float, GHz): the transmit frequency of the antenna
132             wavelength (float, m): the radiation wavelength
133             cross_sect_area (float, m^2): cross sectional area of the
134 ↪ antenna aperture
135             cross_sect_diameter (float, m): cross sectional diameter of the
136 ↪ antenna aperture
137             effective_aperture (float, m^2): The effective collecting area
138 ↪ of a receiving antenna

```

```

134         half_beamwidth (float, deg): The angle between the directions
    ↪ providing half maximum power
135         on either side of the maximum power direction.
136         efficiency (float, ): the efficiency with which the antenna
    ↪ radiates all
137         energy fed into it
138         roughness_factor (float, m): rms roughness of the antenna dish
    ↪ surface
139         """
140         self._amplifier = amplifier
141         self._gain = gain
142         self._loss = loss
143         self._eirp = eirp
144         self._efficiency = efficiency
145         self._half_beamwidth = half_beamwidth
146         self._cross_sect_area = cross_sect_area
147         self._cross_sect_diameter = cross_sect_diameter
148         self._frequency = frequency
149         self._wavelength = wavelength
150         self._modulation = modulation
151         self._effective_aperture = effective_aperture
152         self._roughness_factor = roughness_factor
153         self._carrier_to_noise = carrier_to_noise
154         self._signal_to_noise = signal_to_noise
155         self._carrier_power = carrier_power
156         self._gain_to_noise_temperature = gain_to_noise_temperature
157         self._combined_loss = combined_loss
158         self._power_density = power_density
159
160     def power_density_eirp(self, distance: float, atmospheric_loss: float =
    ↪ 1) -> float:
161         """
162         Calculate the power density of the wavefront using EIRP
163
164         Parameters
165         -----
166         eirp (float, dB)
167         distance (float, km): the distance between the transmit and
    ↪ receive antennas
168         atmospheric_loss (float, ): the total losses due to the
    ↪ atmosphere
169
170         Returns
171         -----
172         power_density (float, W/m^2): the power density at distance d
173         """
174         distance = distance * 1000 # convert to m
175         return self.eirp / (4 * np.pi * distance**2) * atmospheric_loss
176
177     def power_density_distance(self, distance: float) -> float:
178         """
179         Calculate the power density of the wavefront
180
181         Parameters
182         -----

```

```

183         power (float, W): the transmitted power
184         distance (float, km): the distance between the transmit and
↪ receive antennas
185
186     Returns
187     -----
188         power_density (float, W/m^2): the power density at distance d
189         """
190         distance = distance * 1000 # convert to m
191         return (self.amplifier.power * self.gain) / (4 * np.pi * distance
↪ **2)
192
193     @property
194     def combined_loss(self) -> float:
195         if self._combined_loss is None:
196             if self.amplifier is not None:
197                 self._combined_loss = self.loss * self.amplifier.loss
198             elif self._eirp is not None:
199                 self._combined_loss = self.eirp / (self.amplifier.power *
↪ self.gain)
200         return self._combined_loss
201
202     @property
203     def eirp(self) -> float:
204         """
205         Calculate the Effetive Isotropic Radiated Power
206
207
208     Returns
209     -----
210         eirp (float, dB): power incident at the receiver that would
↪ have had to be radiated
211         from an isotropic antenna to achieve the same power
↪ incident at the
212         receiver as that of a transmitter with a specific antenna
↪ gain
213         """
214         if self._eirp is None:
215             self._eirp = self.amplifier.power * self.combined_loss * self.
↪ gain
216         return self._eirp
217
218     @property
219     def half_beamwidth(self) -> float:
220         if self._half_beamwidth is None:
221             self._half_beamwidth = self.wavelength / (
222                 self.cross_sect_diameter * np.sqrt(self.ency)
223             )
224         return self._half_beamwidth
225
226     @property
227     def carrier_to_noise(self) -> float:
228         if self._carrier_to_noise is None:
229             self._carrier_to_noise = self.carrier_power / self.
↪ noise_density

```



```

230     return self._carrier_to_noise
231
232     @property
233     def effective_aperture(self) -> float:
234         """
235         Calculate the effective area of the receiving antenna
236
237         Parameters
238         -----
239
240         Returns
241         -----
242         effective_aperture (float, m^2): the effective aperture of the
↪ receive antenna
243         """
244         if self._effective_aperture is None:
245             self._effective_aperture = self.gain * self.wavelength**2 / (4
↪ * np.pi)
246         return self._effective_aperture
247
248     @property
249     def directive_gain(self) -> float:
250         return self.gain / self.efficiency
251
252     @property
253     def directivity(self) -> float:
254         """
255         TODO
256         """
257         return 4 * pi * self.cross_sect_area / self.wavelength**2
258
259     def pointing_loss(self, pointing_error: float) -> float:
260         """
261         TODO
262         Parameters
263         -----
264         pointing_error (float, deg): angle off nominal pointing direction
265
266         Returns
267         -----
268         pointing_loss (float, ??):
269         """
270         return exp(
271             -2.76 * (radians(pointing_error) / radians(self.half_beamwidth)
↪ ) ** 2
272         )
273
274     def surface_roughness_loss(self) -> float:
275         """
276         TODO
277         Parameters
278         -----
279
280         Returns
281         -----

```

```

282     pointing_loss (float, ??):
283     """
284     if self._surface_roughness_loss is None:
285         self._surface_roughness_loss = exp(
286             -(4 * pi * self.roughness_factor / self.wavelength)
287         )
288     return self._surface_roughness_loss
289
290 @property
291 def gain(self):
292     """
293     TODO
294     Returns
295     -----
296     gain (float, ): ratio of maximum power densiry to that of an
↪ isotropic radiation
297     at the same distance in the direction of the receiving
↪ antenna
298     """
299     if self._gain is None:
300         if self._eirp is not None:
301             self._gain = self.eirp / (self.amplifier.power * self.
↪ combined_loss)
302         elif self._efficiency is not None:
303             self._gain = (
304                 self.efficiency
305                 * 4
306                 * np.pi
307                 * self.cross_sect_area
308                 / self.wavelength**2
309             )
310     return self._gain
311
312 @property
313 def carrier_power(self) -> float:
314     return self._carrier_power
315
316 @property
317 def power_density(self) -> float:
318     return self._power_density
319
320 @power_density.setter
321 def power_density(self, value):
322     self._power_density = value
323
324 @property
325 def frequency(self):
326     """
327     TODO
328     Returns
329     -----
330     frequency (float, GHz): the transmit frequency of the antenna
331     """
332     if self._frequency is None:
333         self._frequency = wavelength_to_frequency(self._wavelength)

```

```

334         return self._frequency
335
336     @property
337     def wavelength(self):
338         """
339         TODO
340         """
341         if self._wavelength is None:
342             self._wavelength = frequency_to_wavelength(self._frequency)
343         return self._wavelength
344
345     @property
346     def efficiency(self):
347         """
348         TODO
349         """
350         if self._efficiency is None:
351             if self._gain is not None and self._cross_sect_area is not None
↪ :
352                 self._efficiency = self.gain / (
353                     4 * np.pi * self.cross_sect_area / self.wavelength**2
354                 )
355         return self._efficiency
356
357     @property
358     def cross_sect_diameter(self):
359         """
360         TODO
361         """
362         return self._cross_sect_diameter
363
364     @property
365     def cross_sect_area(self):
366         """
367         TODO
368         """
369         return self._cross_sect_area
370
371     @property
372     def loss(self):
373         """
374         TODO
375         Returns
376         -----
377         loss (float, ): coupling loss between transmitter and antenna
378                         in the range [0, 1]
379         """
380         if self._loss is None:
381             if self._combined_loss is not None:
382                 if self.amplifier is not None:
383                     self._loss = self.combined_loss / self.amplifier.loss
384                 else:
385                     self._loss = self.combined_loss
386         return self._loss
387

```

```

388     @property
389     def amplifier(self):
390         """
391         TODO
392         """
393         return self._amplifier
394
395     @property
396     def modulation(self):
397         """
398         TODO
399         """
400         return self._modulation
401
402     @property
403     def transmit_loss(self):
404         """
405         TODO
406         Returns
407         -----
408         transmit_loss (float, ): the feeder and branching losses from
409         ↪ the amplifier
410             to the transmit antenna
411         """
412         return self._transmit_loss
413
414     def receive_power(
415         self,
416         transmit_antenna: "Antenna",
417         distance: float,
418         atmospheric_loss: float = 1,
419     ) -> float:
420         """
421         Calculate the power collected by the receive antenna
422
423         Parameters
424         -----
425         distance (float, km): the distance between the transmit and
426         ↪ receive antennas
427         atmospheric_loss (float, ): The loss due to the atmosphere
428
429         Returns
430         -----
431         receive_power (float, W): the total collected power at the
432         ↪ receiver's terminals
433         """
434         pow_density = transmit_antenna.power_density_eirp(distance,
435         ↪ atmospheric_loss)
436         return pow_density * self.effective_aperture * self.loss
437
438     @property
439     def roughness_factor(self):
440         """
441         TODO
442         """

```

```

439         return self._roughness_factor
440
441     @property
442     def signal_to_noise(self):
443         """
444         calculate S/N knowing G/T, wavelength, bandwidth and the field
445         strength of the signal (Duffy 2007).
446
447         Signal/Noise=S(  $\lambda^2/4$  ) (G/T) (1/kbB) where:
448
449         S is power flux density;
450          $\lambda$  is wavelength;
451         kb is Boltzmanns constant; and
452         B is receiver equivalent noise bandwidth
453         """
454         if self._signal_to_noise is None:
455             if (
456                 self._gain_to_noise_temperature is not None
457                 and self._power_density is not None
458             ):
459                 self._signal_to_noise = (
460                     self.power_density
461                     * (self.wavelength**2 / (4 * pi))
462                     * self.gain_to_noise_temperature
463                     * (1 / (BOLTZMANN_CONSTANT * self.modulation.bandwidth))
464                 )
465             return self._signal_to_noise
466
467     @property
468     def gain_to_noise_temperature(self):
469         """
470         TODO
471         """
472         return self._gain_to_noise_temperature
473
474     def summary(self) -> pd.DataFrame:
475         summary = pd.DataFrame.from_records(
476             [
477                 {
478                     "name": "Frequency",
479                     "unit": "GHz",
480                     "value": self.frequency,
481                 },
482                 {
483                     "name": "Wavelength",
484                     "unit": "m",
485                     "value": self.wavelength,
486                 },
487                 {
488                     "name": "Efficiency",
489                     "unit": "%",
490                     "value": self.efficiency,
491                 },
492                 {

```

```

493         "name": "Feeder Loss",
494         "unit": "dB",
495         "value": watt_to_decibel(self.loss),
496     },
497     {
498         "name": "Gain",
499         "unit": "dB",
500         "value": watt_to_decibel(self.gain),
501     },
502     {
503         "name": "EIRP",
504         "unit": "dBW",
505         "value": watt_to_decibel(self.eirp),
506     },
507     {
508         "name": "S/N",
509         "unit": "dBW",
510         "value": watt_to_decibel(self.signal_to_noise),
511     },
512 ]
513 )
514 summary.set_index("name", inplace=True)
515
516 if self.amplifier is not None:
517     amplifier = self.amplifier.summary()
518     amplifier.index = "Amplifier " + amplifier.index
519     summary = pd.concat([summary, amplifier])
520
521 if self.modulation is not None:
522     modulation = self.modulation.summary()
523     modulation.index = "Modulation " + modulation.index
524     summary = pd.concat([summary, modulation])
525 return summary
526
527 def propagate_calculations(self) -> float:
528     for _ in range(3):
529         for var in type(self).__dict__:
530             getattr(self, var)
531
532
533 class HalfWaveDipole(Antenna):
534     """
535     Class for omnidirectuinal radiation pattern
536     """
537
538     def __init__(
539         self,
540         amplifier: Amplifier = None,
541         gain: float = None,
542         loss: float = None,
543         frequency: float = None,
544         effective_aperture: float = None,
545         half_beamwidth: float = None, # deg
546     ):
547         super().__init__(

```

```

548         amplifier=amplifier,
549         gain=gain,
550         loss=loss,
551         frequency=frequency,
552         effective_aperture=effective_aperture,
553         half_beamwidth=half_beamwidth,
554     )
555
556     @property
557     def effective_aperture(self) -> float:
558         if self._effective_aperture is None:
559             self._effective_aperture = 0.13 * self.wavelength
560         return self._effective_aperture
561
562     def off_sight_gain(self, theta: float) -> float:
563         """
564         theta (float, deg): ??
565         """
566         return cos(np.pi / 2 * cos(radians(theta))) ** 2 / sin(radians(
567             ↪ theta)) ** 2
568
569 class ConicalHornAntenna(Antenna):
570     def __init__(
571         self,
572         amplifier: Amplifier = None,
573         gain: float = None,
574         loss: float = None,
575         frequency: float = None,
576         effective_aperture: float = None,
577         half_beamwidth: float = 20, # deg
578     ):
579         super().__init__(
580             amplifier=amplifier,
581             gain=gain,
582             loss=loss,
583             frequency=frequency,
584             effective_aperture=effective_aperture,
585             half_beamwidth=half_beamwidth,
586         )
587
588 class SquareHornAntenna(Antenna):
589     def __init__(
590         self,
591         cross_sect_diameter: float,
592         amplifier: Amplifier = None,
593         efficiency: float = 1,
594         half_beamwidth: float = None, # deg
595         gain: float = None,
596         loss: float = 1,
597         frequency: float = None,
598         effective_aperture: float = None,
599     ):
600         effective_aperture = efficiency * cross_sect_diameter**2

```

```

602         super().__init__(
603             amplifier=amplifier,
604             gain=gain,
605             loss=loss,
606             frequency=frequency,
607             efficiency=efficiency,
608             effective_aperture=effective_aperture,
609             half_beamwidth=half_beamwidth,
610             cross_sect_diameter=cross_sect_diameter,
611         )
612
613     @property
614     def gain(self):
615         """
616         TODO
617         """
618         if self._gain is None:
619             self._gain = (
620                 self.efficiency
621                 * 4
622                 * pi
623                 * self.cross_sect_diameter**2
624                 / self.wavelength**2
625             )
626         return self._gain
627
628     @property
629     def half_beamwidth(self) -> float:
630         """
631         TODO
632         """
633         if self._half_beamwidth is None:
634             self._half_beamwidth = degrees(
635                 0.88 * self.wavelength / self.cross_sect_diameter
636             )
637         return self._half_beamwidth
638
639
640 class ParabolicAntenna(Antenna):
641     def __init__(
642         self,
643         circular_diameter: float,
644         amplifier: Amplifier = None,
645         gain: float = None,
646         loss: float = None,
647         frequency: float = None,
648         wavelength: float = None,
649         effective_aperture: float = None,
650         efficiency: float = None,
651         beamwidth_scale_factor: float = None,
652         half_beamwidth: float = None, # deg
653         roughness_factor: float = None,
654         carrier_to_noise: float = None,
655         signal_to_noise: float = None,
656         modulation: Modulation = None,

```



```

657         combined_loss: float = None,
658     ):
659         self._beamwidth_scale_factor = beamwidth_scale_factor
660         super().__init__(
661             amplifier=amplifier,
662             gain=gain,
663             loss=loss,
664             frequency=frequency,
665             wavelength=wavelength,
666             efficiency=efficiency,
667             effective_aperture=effective_aperture,
668             half_beamwidth=half_beamwidth,
669             cross_sect_diameter=circular_diameter,
670             modulation=modulation,
671             carrier_to_noise=carrier_to_noise,
672             signal_to_noise=signal_to_noise,
673             combined_loss=combined_loss,
674         )
675
676     @property
677     def gain(self) -> float:
678         """
679         TODO
680         """
681         if self._gain is None:
682             self._gain = (
683                 self.efficiency * (pi * self.cross_sect_diameter / self.
↵ wavelength) ** 2
684             )
685         return self._gain
686
687     @property
688     def cross_sect_area(self):
689         """
690         TODO
691         """
692         if self._cross_sect_area is None:
693             self._cross_sect_area = pi / 4 * self.circular_diameter**2
694         return self._cross_sect_area
695
696     @property
697     def half_beamwidth(self) -> float:
698         """
699         TODO
700         """
701         if self._half_beamwidth is None:
702             self._half_beamwidth = self._beamwidth_scale_factor * (
703                 self.wavelength / self.cross_sect_diameter
704             )
705         return self._half_beamwidth
706
707     def off_sight_gain(self, k: float, theta: float):
708         """
709         TODO
710         """

```

```

711         return self.gain * self.pointing_loss(theta)
712
713     def summary(self) -> pd.DataFrame:
714         summary = pd.DataFrame.from_records(
715             [
716                 {
717                     "name": "Diameter",
718                     "unit": "m",
719                     "value": self.cross_sect_diameter,
720                 },
721                 {
722                     "name": "Gain",
723                     "unit": "dB",
724                     "value": watt_to_decibel(self.gain),
725                 },
726                 {
727                     "name": "Half Beamwidth",
728                     "unit": "dBW",
729                     "value": watt_to_decibel(self.eirp),
730                 },
731             ]
732         )
733         summary.set_index("name", inplace=True)
734
735         antenna = super().summary()
736         antenna.drop("Gain", inplace=True)
737         summary = pd.concat([summary, antenna])
738         return summary
739
740
741 class HelicalAntenna(Antenna):
742     def __init__(
743         self,
744         circular_diameter: float,
745         n_helix_turns: float,
746         turn_spacing: float,
747         amplifier: Amplifier = None,
748         gain: float = None,
749         loss: float = None,
750         frequency: float = None,
751         effective_aperture: float = None,
752         efficiency: float = None,
753         half_beamwidth: float = 20, # deg
754     ):
755         self.n_helix_turns = n_helix_turns
756         self.turn_spacing = n_helix_turns
757         super().__init__(
758             amplifier=amplifier,
759             gain=gain,
760             loss=loss,
761             frequency=frequency,
762             effective_aperture=effective_aperture,
763             half_beamwidth=half_beamwidth,
764             cross_sect_diameter=circular_diameter,
765         )

```

```

766
767     @property
768     def gain(self) -> float:
769         """
770         TODO
771         """
772         if self._gain is None:
773             self._gain = (
774                 15
775                 * self.n_helix_turns
776                 * self.turn_spacing
777                 * (pi**2)
778                 * (self.cross_sect_diameter**2)
779                 / self.wavelength**3
780             )
781         return self._gain

```

Listing 4: components/antennas.py

```

1 from math import erfc, log2, log10, pi, sin, sqrt
2
3 import pandas as pd
4 from scipy.special import erfcinv
5
6 from link_calculator.propagation.conversions import watt_to_decibel
7 from link_calculator.signal_processing.conversions import (
8     GHz_to_Hz,
9     Hz_to_GHz,
10    Hz_to_MHz,
11    bit_to_mbit,
12    mbit_to_bit,
13 )
14
15
16 class Modulation:
17     def __init__(
18         self,
19         bandwidth: float = None,
20         carrier_to_noise: float = None,
21     ):
22         self._bandwidth = bandwidth
23         self._carrier_to_noise = carrier_to_noise
24
25     @property
26     def channel_capacity(self) -> float:
27         return self.bandwidth * log2(1 + self.carrier_to_noise)
28
29     @property
30     def channel_capacity_to_bandwidth_ideal(self) -> float:
31         return log2(1 + self.eb_no * self.channel_capacity / self.bandwidth
32         ↪ )
33
34     @property
35     def channel_capacity_to_bandwidth(self) -> float:
36         return log2(1 + self.eb_no * self.bit_rate / self.bandwidth)

```

```

36
37     @property
38     def eb_no(self) -> float:
39         return (2 ** (self.channel_capacity / self.bandwidth) - 1) / (
40             self.channel_capacity / self.bandwidth
41         )
42
43
44 class FrequencyModulation(Modulation):
45     def __init__(
46         self,
47         bandwidth: float = None,
48         baseband_bandwidth: float = None,
49         carrier_to_noise: float = None,
50         signal_to_noise: float = None,
51         deviation_ratio: float = None,
52         frequency_deviation: float = None,
53         threshold: float = None,
54         link_margin: float = None,
55         deemphasis_improvement: float = 1,
56         preemphasis_improvement: float = 1,
57     ):
58         """
59         bandwidth (float, GHz): the range of frequencies required to
↪ transmit the signal
60         baseband_bandwidth (float, GHz): the range of frequencies required
↪ to
61             transmit the baseband signal
62         carrier_to_noise (float, W):
63         deemphasis_improvement (float, W):
64         preemphasis_improvement (float, W):
65         """
66         self._bandwidth = bandwidth
67         self._baseband_bandwidth = baseband_bandwidth
68         self._carrier_to_noise = carrier_to_noise
69         self._signal_to_noise = signal_to_noise
70         self._deviation_ratio = deviation_ratio
71         self._frequency_deviation = frequency_deviation
72         self._deemphasis_improvement = deemphasis_improvement
73         self._preemphasis_improvement = preemphasis_improvement
74         self._threshold = threshold
75         self._link_margin = link_margin
76
77     @property
78     def bandwidth(self) -> float:
79         if self._bandwidth is None:
80             self._bandwidth = 2 * (self.frequency_deviation + self.
↪ baseband_bandwidth)
81         return self._bandwidth
82
83     @property
84     def frequency_deviation(self) -> float:
85         if self._frequency_deviation is None:
86             self._frequency_deviation = self.bandwidth / 2 - self.
↪ baseband_bandwidth

```

```

87         return self._frequency_deviation
88
89     @property
90     def deviation_ratio(self) -> float:
91         if self._deviation_ratio is None:
92             self._deviation_ratio = self.frequency_deviation / self.
↪ baseband_bandwidth
93         return self._deviation_ratio
94
95     @property
96     def signal_to_noise(self) -> float:
97         if self._signal_to_noise is None:
98             self._signal_to_noise = (
99                 (3 / 2)
100                 * self.carrier_to_noise
101                 * (self.bandwidth / self.baseband_bandwidth)
102                 * self.deviation_ratio**2
103                 * self.preemphasis_improvement
104                 * self.deemphasis_improvement
105             )
106         return self._signal_to_noise
107
108     @property
109     def link_margin(self) -> float:
110         if self._link_margin is None:
111             self._link_margin = self.carrier_to_noise / self.threshold
112         return self._link_margin
113
114     @property
115     def threshold(self) -> float:
116         return self._threshold
117
118     @property
119     def baseband_bandwidth(self) -> float:
120         return self._baseband_bandwidth
121
122     @property
123     def carrier_to_noise(self) -> float:
124         return self._carrier_to_noise
125
126     @property
127     def deemphasis_improvement(self) -> float:
128         return self._deemphasis_improvement
129
130     @property
131     def preemphasis_improvement(self) -> float:
132         return self._preemphasis_improvement
133
134
135 class Waveform:
136     def __init__(
137         self, frequency: float = None, amplitude: float = None, phase:
↪ float = None
138     ):
139         self.frequency = frequency

```

```

140         self.amplitude = amplitude
141         self.phase = phase
142
143
144 class ConvolutionalCode:
145     def __init__(
146         self,
147         coding_rate: float = None,
148         coding_gain: float = None,
149         min_distance: float = None,
150     ):
151         """
152         coding_rate (float, bps):
153         coding_gain (float, W):
154         """
155         self._coding_rate = coding_rate
156         self._coding_gain = coding_gain
157         self._min_distance = min_distance
158
159     @property
160     def coding_rate(self) -> float:
161         return self._coding_rate
162
163     @property
164     def coding_gain(self) -> float:
165         if self._coding_gain is None:
166             if self._min_distance is not None:
167                 self._coding_gain = self.code_rate * self.min_distance
168         return self._coding_gain
169
170     @staticmethod
171     def coding_gain_eb_no(eb_no_coded: float, eb_no_uncoded: float) ->
172     ↪ float:
173         """
174         calculate the difference in Eb/No required to produce the same
175         ↪ error rate for coded and
176         uncoded signals
177
178         Parameters
179         -----
180             eb_no_coded (float, W); the Eb/No of the coded signal
181             eb_no_uncoded (float, W): the Eb/No of the uncoded signal
182
183         Returns
184         -----
185             coding_gain (float, ):
186             """
187             return eb_no_uncoded / eb_no_coded
188
189     def summary(self) -> pd.DataFrame:
190         summary = pd.DataFrame.from_records(
191             [
192                 {
193                     "name": "Coding Rate",
194                     "unit": "mbps",

```

```

193         "value": self.coding_rate,
194     },
195     {
196         "name": "Coding Gain",
197         "unit": "dB",
198         "value": watt_to_decibel(self.coding_gain),
199     },
200 ]
201 )
202 summary.set_index("name", inplace=True)
203 return summary
204
205
206 class MPhaseShiftKeying(Modulation):
207     def __init__(
208         self,
209         levels: int,
210         bandwidth: float = None,
211         carrier_signal: Waveform = None,
212         modulating_signal: Waveform = None,
213         energy_per_symbol: float = None,
214         energy_per_bit: float = None,
215         symbol_rate: float = None,
216         symbol_period: float = None,
217         bit_rate: float = None,
218         bit_error_rate: float = None,
219         bit_error_rate_coded: float = None,
220         bit_period: float = None,
221         bits_per_symbol: int = None,
222         carrier_power: float = None,
223         coding_rate: float = None,
224         carrier_to_noise: float = None,
225         carrier_to_noise_coded: float = None,
226         spectral_efficiency: float = None,
227         eb_no: float = None,
228         eb_no_coded: float = None,
229         es_no: float = None,
230         es_no_coded: float = None,
231         rolloff_rate: float = None,
232         frequency_range: list = None,
233         noise_probability: float = None,
234         noise_probability_coded: float = None,
235         noise_power_density: float = None,
236         noise_power_density_coded: float = None,
237         code: ConvolutionalCode = None,
238         data_rate: float = None,
239     ):
240         """
241         Parameters
242         -----
243             levels (int,): number of levels in the waveform (equivalent to
244             ↪ number of symbols)
245             bandwidth (float, GHz): the range of frequencies required to
246             ↪ transmit the signal
247             energy_per_symbol (float, J): energy required to transmit one

```

```

↪ symbol
246         energy_per_bit (float, J): energy required to transmit one bit
247         symbol_rate (float, symbols/s): number of symbols transmitted
↪ per second
248         symbol_period (float, s/symbol): seconds to transmit a symbol
249         bit_rate (float, bits/s): number of bits transmitted per second
250         bit_period (float, s/bit): number of seconds to transmit a bit
251         carrier_power (float, W): total transmit power of signal
252         spectral_efficiency (float bit/s/Hz):
253         """
254         self._levels = levels
255         self._bandwidth = bandwidth
256         self._energy_per_symbol = energy_per_symbol
257         self._energy_per_bit = energy_per_bit
258         self._symbol_rate = symbol_rate
259         self._symbol_period = symbol_period
260         self._bits_per_symbol = bits_per_symbol
261         self._bit_rate = bit_rate
262         self._bit_error_rate = bit_error_rate
263         self._bit_period = bit_period
264         self._carrier_power = carrier_power
265         self._carrier_to_noise = carrier_to_noise
266         self._eb_no = eb_no
267         self._es_no = es_no
268         self._rolloff_rate = rolloff_rate
269         self._carrier_signal = carrier_signal
270         self._modulating_signal = modulating_signal
271         self._frequency_range = frequency_range
272         self._spectral_efficiency = spectral_efficiency
273         self._noise_probability = noise_probability
274         self._code = code
275         self._data_rate = data_rate
276         self._bit_error_rate_coded = bit_error_rate_coded
277         self._carrier_to_noise_coded = carrier_to_noise_coded
278         self._eb_no_coded = eb_no_coded
279         self._noise_power_density_coded = noise_power_density_coded
280         self._noise_power_density = noise_power_density
281         self._noise_probability_coded = noise_probability_coded
282         self._es_no_coded = es_no_coded
283         self.propagate_calculations()
284
285     @property
286     def es_no(self) -> float:
287         if self._es_no is None:
288             if self._carrier_to_noise is not None:
289                 self._es_no = (
290                     self.carrier_to_noise * GHz_to_Hz(self.bandwidth) /
↪ self.symbol_rate
291                 )
292             if self._eb_no is not None:
293                 return self.eb_no * self.bits_per_symbol
294             return self._es_no
295
296     @property
297     def es_no_coded(self) -> float:

```



```

298         if self._es_no_coded is None:
299             if self._carrier_to_noise_coded is not None:
300                 self._es_no = (
301                     self.carrier_to_noise_coded
302                     * GHz_to_Hz(self.bandwidth)
303                     / self.symbol_rate
304                 )
305             elif self._eb_no_coded is not None:
306                 return self.eb_no_coded * self.bits_per_symbol
307         return self._es_no_coded
308
309     @property
310     def eb_no(self) -> float:
311         if self._eb_no is None:
312             if self._bit_error_rate is not None:
313                 self._eb_no = (
314                     erfcinv(self.bit_error_rate * self.bits_per_symbol)
315                     / sin(pi / self.levels)
316                 ) ** 2 / self.bits_per_symbol
317             elif self._es_no is not None:
318                 self._eb_no = self.es_no / self.bits_per_symbol
319             elif (
320                 self._energy_per_bit is not None
321                 and self._noise_power_density is not None
322             ):
323                 self._eb_no = self.energy_per_bit / self.
↪ noise_power_density
324         return self._eb_no
325
326     @property
327     def eb_no_coded(self) -> float:
328         if self._eb_no_coded is None:
329             if self._code is not None:
330                 self._eb_no_coded = self.eb_no * self.code.coding_gain
331             else:
332                 self._eb_no_coded = self.eb_no
333         return self._eb_no_coded
334
335     @eb_no.setter
336     def eb_no(self, value) -> None:
337         self._eb_no = value
338
339     @property
340     def carrier_power(self) -> float:
341         return self._carrier_power
342
343     @carrier_power.setter
344     def carrier_power(self, value) -> None:
345         self._carrier_power = value
346
347     @property
348     def bit_rate(self) -> float:
349         if self._bit_rate is None:
350             if self._bit_period is not None:
351                 self._bit_rate = 1.0 / self.bit_period

```

```

352         elif self._rolloff_rate is not None:
353             self._bit_rate = (
354                 self.bits_per_symbol
355                 * GHz_to_Hz(self.bandwidth)
356                 / (1 + self.rolloff_rate)
357             )
358         return self._bit_rate
359
360     @property
361     def data_rate(self) -> float:
362         if self._data_rate is None:
363             if self._code is not None:
364                 self._data_rate = self.bit_rate * self.code.coding_rate
365             else:
366                 self._data_rate = self.bit_rate
367         return self._data_rate
368
369     @property
370     def bit_period(self) -> float:
371         if self._bit_period is None:
372             self._bit_period = self.symbol_period / self.bits_per_symbol
373         return self._bit_period
374
375     @property
376     def symbol_rate(self) -> float:
377         if self._symbol_rate is None:
378             if self._bandwidth is not None and self._rolloff_rate is not
↪ None:
379                 self._symbol_rate = (GHz_to_Hz(self._bandwidth)) / (
380                     1 + self.rolloff_rate
381                 )
382             else:
383                 self._symbol_rate = self.bit_rate / self.bits_per_symbol
384         return self._symbol_rate
385
386     @property
387     def symbol_period(self) -> float:
388         if self._symbol_period is None:
389             self._symbol_period = 1.0 / self.symbol_rate
390         return self._symbol_period
391
392     @property
393     def spectral_efficiency(self) -> float:
394         if self._spectral_efficiency is None:
395             if self._bit_rate is not None and self._bandwidth is not None:
396                 self._spectral_efficiency = self.bit_rate / GHz_to_Hz(self.
↪ bandwidth)
397         return self._spectral_efficiency
398
399     @property
400     def energy_per_symbol(self) -> float:
401         if self._energy_per_symbol is None:
402             if self._carrier_power is not None and self._symbol_period is
↪ not None:
403                 self._energy_per_symbol = self.carrier_power * self.

```

```

404     ↪ symbol_period
         return self._energy_per_symbol
405
406     @property
407     def bits_per_symbol(self):
408         if self._bits_per_symbol is None:
409             self._bits_per_symbol = log2(self.levels)
410         return self._bits_per_symbol
411
412     @property
413     def energy_per_bit(self) -> float:
414         """
415         Returns
416             energy_per_bit (float, J/s)
417         """
418         if self._energy_per_bit is None:
419             if self._carrier_power is not None:
420                 self._energy_per_bit = self.carrier_power * self.bit_period
421             return self._energy_per_bit
422
423     @property
424     def noise_power_density(self) -> float:
425         if self._noise_power_density is None:
426             if (
427                 self._symbol_rate is not None
428                 and self._carrier_to_noise is not None
429                 and self._energy_per_symbol is not None
430             ):
431                 self._noise_power_density = self.symbol_rate / (
432                     self.bandwidth * self.carrier_to_noise * self.
433     ↪ energy_per_symbol
         )
434         return self._noise_power_density
435
436     @property
437     def noise_power_density_coded(self) -> float:
438         if self._noise_power_density_coded is None:
439             if (
440                 self._symbol_rate is not None
441                 and self._carrier_to_noise_coded is not None
442                 and self._energy_per_symbol is not None
443             ):
444                 self._noise_power_density_coded = self.symbol_rate / (
445                     self.bandwidth
446                     * self.carrier_to_noise_coded
447                     * self.energy_per_symbol
448                 )
449             return self._noise_power_density_coded
450
451     @property
452     def noise_probability(self) -> float:
453         if self._noise_probability is None:
454             if self._es_no is not None:
455                 self._noise_probability = erfc(sqrt(self.es_no) * sin(pi /
456     ↪ self.levels))

```

```

456         return self._noise_probability
457
458     @property
459     def noise_probability_coded(self) -> float:
460         if self._noise_probability_coded is None:
461             if self._eb_no_coded is not None:
462                 self._noise_probability_coded = erfc(
463                     sqrt(self.bits_per_symbol * self.eb_no_coded)
464                     * sin(pi / self.levels)
465                 )
466             return self._noise_probability_coded
467
468     @property
469     def bit_error_rate(self) -> float:
470         if self._bit_error_rate is None:
471             if self._noise_probability is not None:
472                 self._bit_error_rate = self.noise_probability / self.
↪ bits_per_symbol
473             return self._bit_error_rate
474
475     @property
476     def bit_error_rate_coded(self) -> float:
477         if self._bit_error_rate_coded is None:
478             if self._noise_probability_coded is not None:
479                 self._bit_error_rate_coded = (
480                     self.noise_probability_coded / self.bits_per_symbol
481                 )
482             return self._bit_error_rate_coded
483
484     @property
485     def levels(self) -> float:
486         return self._levels
487
488     @property
489     def rolloff_rate(self) -> float:
490         return self._rolloff_rate
491
492     @property
493     def carrier_to_noise(self) -> float:
494         if self._carrier_to_noise is None:
495             if self._eb_no is not None:
496                 self._carrier_to_noise = (
497                     self.eb_no * self.bit_rate / GHz_to_Hz(self._bandwidth)
498                 )
499             return self._carrier_to_noise
500
501     @property
502     def carrier_to_noise_coded(self) -> float:
503         if self._carrier_to_noise_coded is None:
504             if self._eb_no_coded is not None:
505                 self._carrier_to_noise_coded = (
506                     self.eb_no_coded * self.bit_rate / GHz_to_Hz(self.
↪ _bandwidth)
507                 )
508             return self._carrier_to_noise_coded

```

```

509
510     @property
511     def carrier_signal(self) -> Waveform:
512         return self._carrier_signal
513
514     @property
515     def code(self) -> ConvolutionalCode:
516         return self._code
517
518     @property
519     def bandwidth(self) -> float:
520         if self._bandwidth is None:
521             if self._symbol_rate is not None and self._rolloff_rate is not
↪ None:
522                 self._bandwidth = self.symbol_rate * (1 + self.rolloff_rate
↪ )
523         return self._bandwidth
524
525     @property
526     def frequency_range(self) -> float:
527         if self._frequency_range is None:
528             self._frequency_range = [
529                 self.carrier_signal.frequency - self.bandwidth / 2,
530                 self.carrier_signal.frequency + self.bandwidth / 2,
531             ]
532         return self._frequency_range
533
534     def summary(self) -> pd.DataFrame:
535         summary = pd.DataFrame.from_records(
536             [
537                 {
538                     "name": "Maximum Bit Rate",
539                     "unit": "mbps",
540                     "value": bit_to_mbit(self.bit_rate),
541                 },
542                 {
543                     "name": "Data Rate",
544                     "unit": "mbps",
545                     "value": bit_to_mbit(self.data_rate),
546                 },
547                 {"name": "Bandwidth", "unit": "GHz", "value": self.
↪ bandwidth},
548                 {
549                     "name": "Spectral Efficiency",
550                     "unit": "bits/s/Hz",
551                     "value": self.spectral_efficiency,
552                 },
553                 {
554                     "name": "C/N Ratio",
555                     "unit": "bits/s/GHz",
556                     "value": watt_to_decibel(self.carrier_to_noise),
557                 },
558                 {
559                     "name": "Coded C/N Ratio",
560                     "unit": "bits/s/GHz",

```

```

561         "value": watt_to_decibel(self.carrier_to_noise_coded),
562     },
563     {
564         "name": "Eb/No Ratio",
565         "unit": "dB",
566         "value": watt_to_decibel(self.eb_no),
567     },
568     {
569         "name": "Coded Eb/No Ratio",
570         "unit": "dB",
571         "value": watt_to_decibel(self.eb_no_coded),
572     },
573     {
574         "name": "Bit Error Rate",
575         "unit": "",
576         "value": self.bit_error_rate,
577     },
578     {
579         "name": "Coded Bit Error Rate",
580         "unit": "",
581         "value": self.bit_error_rate_coded,
582     },
583     {"name": "Roll-Off Factor", "unit": "", "value": self.
↪ rolloff_rate},
584     ]
585     )
586     summary.set_index("name", inplace=True)
587
588     if self.code is not None:
589         code = self.code.summary()
590         summary = pd.concat([summary, code])
591     else:
592         summary = summary[[not ("Coded" in idx) for idx in summary.
↪ index]]
593
594     return summary
595
596     def propagate_calculations(self) -> float:
597         for _ in range(4):
598             for var in type(self).__dict__:
599                 getattr(self, var, None)
600
601
602 class BinaryPhaseShiftKeying(MPhaseShiftKeying):
603     def __init__(
604         self,
605         bandwidth: float = None,
606         carrier_signal: Waveform = None,
607         modulating_signal: Waveform = None,
608         energy_per_symbol: float = None,
609         energy_per_bit: float = None,
610         symbol_rate: float = None,
611         symbol_period: float = None,
612         bit_rate: float = None,
613         bit_period: float = None,

```

```

614         bits_per_symbol: float = None,
615         carrier_power: float = None,
616         rolloff_rate: float = None,
617         frequency_range: list = None,
618     ):
619         super().__init__(
620             levels=2,
621             bandwidth=bandwidth,
622             carrier_signal=carrier_signal,
623             modulating_signal=modulating_signal,
624             energy_per_bit=energy_per_bit,
625             rolloff_rate=rolloff_rate,
626             frequency_range=frequency_range,
627             carrier_power=carrier_power,
628             energy_per_symbol=energy_per_symbol,
629             symbol_rate=symbol_rate,
630             bit_rate=bit_rate,
631             bit_period=bit_period,
632             bits_per_symbol=bits_per_symbol,
633         )
634
635     @property
636     def noise_probability(self) -> float:
637         if self._noise_probability is None:
638             if self._carrier_to_noise is not None:
639                 self._noise_probability = 0.5 * erfc(sqrt(self.
↪ carrier_to_noise))
640         return self._noise_probability
641
642
643 class QuadraturePhaseShiftKeying(MPhaseShiftKeying):
644     def __init__(
645         self,
646         bandwidth: float = None,
647         carrier_signal: Waveform = None,
648         modulating_signal: Waveform = None,
649         energy_per_symbol: float = None,
650         symbol_rate: float = None,
651         symbol_period: float = None,
652         carrier_power: float = None,
653         bit_rate: float = None,
654         bit_period: float = None,
655         bits_per_symbol: float = None,
656         rolloff_rate: float = None,
657     ):
658         super().__init__(
659             levels=4,
660             bandwidth=bandwidth,
661             carrier_signal=carrier_signal,
662             modulating_signal=modulating_signal,
663             energy_per_symbol=energy_per_symbol,
664             symbol_rate=symbol_rate,
665             carrier_power=carrier_power,
666             bit_rate=bit_rate,
667             bit_period=bit_period,

```

```

668         bits_per_symbol=bits_per_symbol,
669         rolloff_rate=rolloff_rate,
670     )
671
672     @property
673     def noise_probability(self) -> float:
674         if self._noise_probability is None:
675             if self._carrier_to_noise is not None:
676                 self._noise_probability = 0.5 * erfc(sqrt(self.
↪ carrier_to_noise * 0.5))
677         return self._noise_probability

```

Listing 5: *signal\_processing/modulation.py*

```

1 from math import atan2, cos, degrees, radians
2
3 import numpy as np
4
5 from link_calculator.components.antennas import Antenna
6 from link_calculator.constants import EARTH_RADIUS
7
8
9 def slant_path(
10     elevation_angle: float,
11     rain_altitude: float,
12     station_altitude: float,
13 ) -> float:
14     """
15     Calculate the slant path
16
17     Parameters
18     -----
19     angle_of_elevation (float, deg): the angle between the Earth
↪ station and the satellite
20     rain_height (float, km): the rain height
21     station_altitude (float, km): the rain height of the Earth station
↪ above sea level
22     refraction_radius (float, km): The modified radius of the Earth to
↪ account for the
23         refraction of the wave by thr troposphere
24
25     Returns
26     -----
27     d_s (float, km): The slant height
28     """
29     refraction_radius = 8500 if station_altitude < 1.0 else EARTH_RADIUS
30     elevation_angle_rad = radians(elevation_angle)
31     if elevation_angle < 5:
32         return (
33             2
34             * (rain_altitude - station_altitude)
35             / np.sqrt(
36                 np.sin(elevation_angle_rad) ** 2
37                 + 2 * (rain_altitude - station_altitude) /
↪ refraction_radius

```



```

38         )
39     )
40     else:
41         return (rain_altitude - station_altitude) / np.sin(
42             ↪ elevation_angle_rad)
43
44 def rain_specific_attenuation(frequency: float, rain_rate: float,
45     ↪ polarization: str):
46     """
47     TODO()
48
49     Parameters
50     -----
51
52     Returns
53     -----
54
55     """
56     _f = [
57         1,
58         2,
59         4,
60         6,
61         7,
62         8,
63         10,
64         12,
65         15,
66         20,
67         25,
68         30,
69         35,
70         40,
71         45,
72         50,
73         60,
74         70,
75         80,
76         90,
77         100,
78         120,
79         150,
80         200,
81         300,
82         400,
83     ]
84
85     _kH = [
86         0.0000387,
87         0.000154,
88         0.00065,
89         0.00175,
90         0.00301,
91         0.00454,

```

```
91      0.0101,
92      0.0188,
93      0.0367,
94      0.0751,
95      0.124,
96      0.187,
97      0.263,
98      0.35,
99      0.442,
100     0.536,
101     0.707,
102     0.851,
103     0.975,
104     1.06,
105     1.12,
106     1.18,
107     1.31,
108     1.45,
109     1.36,
110     1.32,
111 ]
112
113 _kV = [
114     0.0000352,
115     0.000138,
116     0.000591,
117     0.00155,
118     0.00265,
119     0.00395,
120     0.00887,
121     0.0168,
122     0.0335,
123     0.0691,
124     0.113,
125     0.167,
126     0.233,
127     0.31,
128     0.393,
129     0.479,
130     0.642,
131     0.784,
132     0.906,
133     0.999,
134     1.06,
135     1.13,
136     1.27,
137     1.42,
138     1.35,
139     1.31,
140 ]
141
142 _alphaH = [
143     0.912,
144     0.963,
145     1.121,
```

```
146         1.308,
147         1.332,
148         1.327,
149         1.276,
150         1.217,
151         1.154,
152         1.099,
153         1.061,
154         1.021,
155         0.979,
156         0.939,
157         0.903,
158         0.873,
159         0.826,
160         0.793,
161         0.769,
162         0.753,
163         0.743,
164         0.731,
165         0.71,
166         0.689,
167         0.688,
168         0.683,
169     ]
170
171     _alphaV = [
172         0.88,
173         0.923,
174         1.075,
175         1.265,
176         1.312,
177         1.31,
178         1.264,
179         1.2,
180         1.128,
181         1.065,
182         1.03,
183         1,
184         0.963,
185         0.929,
186         0.897,
187         0.868,
188         0.824,
189         0.793,
190         0.769,
191         0.754,
192         0.744,
193         0.732,
194         0.711,
195         0.69,
196         0.689,
197         0.684,
198     ]
199
200     KH = np.exp(np.interp(np.log(frequency), np.log(_f), np.log(_kH)))
```

```

201     KV = np.exp(np.interp(np.log(frequency), np.log(_f), np.log(_kV)))
202
203     alphaH = np.interp(np.log(frequency), np.log(_f), _alphaH)
204     alphaV = np.interp(np.log(frequency), np.log(_f), _alphaV)
205
206     if polarization == "circular":
207         k = (KH + KV) / 2
208         alpha = (KH * alphaH + KV * alphaV) / (2 * k)
209     elif polarization == "vertical":
210         k = KV
211         alpha = alphaV
212     elif polarization == "horizontal":
213         k = KH
214         alpha = alphaH
215     else:
216         raise Exception("Invalid Polarization")
217
218     return k, alpha, k * rain_rate**alpha
219
220
221 def horizontal_reduction(
222     horizontal_projection: float, specific_attenuation: float, frequency:
223     ↪ float
224 ) -> float:
225     """
226     TODO()
227
228     Parameters
229     -----
230
231     Returns
232     -----
233     """
234     return 1 / (
235         1
236         + 0.78 * np.sqrt(horizontal_projection * specific_attenuation /
237         ↪ frequency)
238         - 0.38 * (1 - np.exp(-2 * horizontal_projection))
239     )
240
241 def vertical_adjustment(
242     elevation_angle: float,
243     specific_attenuation: float,
244     d_r: float,
245     frequency: float,
246     chi: float,
247 ) -> float:
248     """
249     Calculate the vertical adjustment factor
250
251     Parameters
252     -----
253     elevation_angle (float, deg):
254     specific_attenuation (float, dBKm-1)

```

```

254     d_r (float, km):
255     frequency (float, GHz):
256     chi (float, deg):
257
258     Returns
259     -----
260     vert_adj (float, )
261     """
262     return 1 / (
263         1
264         + np.sqrt(np.sin(radians(elevation_angle)))
265         * (
266             31
267             * (1 - np.exp(-elevation_angle / (1 + chi)))
268             * (np.sqrt(d_r * specific_attenuation) / (frequency**2))
269             - 0.45
270         )
271     )
272
273
274 def zeta(
275     rain_altitude: float,
276     station_altitude: float,
277     horizontal_projection: float,
278     horizontal_reduction: float,
279 ) -> float:
280     """
281     Calculate interim vertical adjustment value
282
283     Parameters
284     -----
285
286     Returns
287     -----
288     """
289     return degrees(
290         atan2(
291             rain_altitude - station_altitude,
292             horizontal_projection * horizontal_reduction,
293         )
294     )
295
296
297 def rain_attenuation(
298     elevation_angle: float,
299     slant_path: float,
300     frequency: float,
301     rain_altitude: float,
302     station_altitude: float,
303     station_latitude: float,
304     rain_rate: float = 0.01,
305     polarization: str = "vertical",
306 ) -> float:
307     """
308

```

```

309     Parameters
310     -----
311
312     Returns
313     -----
314
315     """
316     elevation_angle_rad = radians(elevation_angle)
317     horiz_proj = slant_path * np.cos(elevation_angle_rad)
318
319     _, _, specific_att = rain_specific_attenuation(frequency, rain_rate,
320     ↪ polarization)
321
322     horiz_reduction = horizontal_reduction(horiz_proj, specific_att,
323     ↪ frequency)
324
325     zeta_ = zeta(rain_altitude, station_altitude, horiz_proj,
326     ↪ horiz_reduction)
327
328     if zeta_ > elevation_angle:
329         d_r = horiz_proj * horiz_reduction / np.cos(elevation_angle_rad)
330     else:
331         d_r = slant_path
332
333     if abs(station_latitude) < 36:
334         chi = 36 - abs(station_latitude)
335     else:
336         chi = 0
337
338     vert_adj = vertical_adjustment(elevation_angle, specific_att, d_r,
339     ↪ frequency, chi)
340     effective_path = slant_path * vert_adj
341
342     return specific_att * effective_path
343
344
345 def worst_rain_rate(rain_rate: float) -> float:
346     return (rain_rate / 0.3) ** 0.87
347
348
349 def polarization_loss(faraday_rotation: float) -> float:
350     rotation_rad = radians(faraday_rotation)
351     return 20 * np.log(cos(rotation_rad))

```

Listing 6: propagation/utils.py

```

1 from math import acos, atan, cos, degrees, pi, radians, sin, sqrt, tan
2
3 import pandas as pd
4
5 from link_calculator.constants import EARTH_MU, EARTH_RADIUS
6
7
8 class Orbit:
9     def __init__(

```

```

10         self,
11         semi_major_axis: float = None,
12         eccentricity: float = None,
13         inclination: float = float,
14         raan: float = None,
15         arg_of_perigee: float = None,
16         true_anomaly: float = None,
17         period: float = None,
18         orbital_radius: float = None,
19     ):
20         """
21
22         Parameters
23         -----
24
25         semi_major_axis (float, km): The semi-major axis of the orbit
26         """
27         self._semi_major_axis = semi_major_axis
28         self._eccentricity = eccentricity
29         self._inclination = inclination
30         self._raan = raan
31         self._arg_of_perigee = arg_of_perigee
32         self._true_anomaly = true_anomaly
33         self._period = period
34         self._orbital_radius = orbital_radius
35
36     def period(self, mu: float = EARTH_MU) -> float:
37         """
38         Calculate the period of the satellite's orbit according to Kepler's
39         ↪ third law
40
41         mu (float, km^3/s^-2, optional): Kepler's gravitational
42         ↪ constant
43
44         Returns
45         -----
46
47         period (float, s): the time taken for the satellite to complete
48         ↪ a revolution
49         """
50         if self._period is None:
51             self._period = 2 * pi * sqrt(self.semi_major_axis**3 / mu)
52         return self._period
53
54     @property
55     def semi_major_axis(self) -> float:
56         return self._semi_major_axis
57
58     @property
59     def orbital_radius(self) -> float:
60         if self._orbital_radius is None:
61             self._orbital_radius = (
62                 self.semi_major_axis
63                 * (1 - self.eccentricity**2)
64                 / (1 + self.eccentricity * cos(radians(self.true_anomaly)))
65             )
66         return self._orbital_radius

```

```

62
63
64 class GeodeticCoordinate:
65     def __init__(self, latitude: float, longitude: float, altitude: float =
        ↪ 0):
66         self._latitude = latitude
67         self._longitude = longitude
68         self._altitude = altitude
69
70     @property
71     def latitude(self) -> float:
72         return self._latitude
73
74     @property
75     def longitude(self) -> float:
76         return self._longitude
77
78     @property
79     def altitude(self) -> float:
80         return self._altitude
81
82     def central_angle(
83         self,
84         point: "GeodeticCoordinate",
85     ) -> float:
86         """
87         ↪ Calculate angle gamma at the centre of the ground, between the
            ↪ Earth station and the satellite
88
89         Parameters
90         -----
91         ↪ ground_station_lat (float, deg): the latitude of the ground
            ↪ station
92         ↪ ground_station_long (float, deg): the longitude of the ground
            ↪ station
93         ↪ sat_lat (float, deg): the latitude of the satellite
94         ↪ sat_long (float, deg): the longitude of the satellite
95
96         Returns
97         -----
98         ↪ gamma (float, rad): angle between satellite and ground station
99         """
100         gamma = acos(
101             cos(radians(self.latitude))
102             * cos(radians(point.latitude))
103             * cos(radians(point.longitude) - radians(self.longitude))
104             + sin(radians(self.latitude)) * sin(radians(point.latitude))
105         )
106         return degrees(gamma)
107
108     def summary(self) -> pd.DataFrame:
109         summary = pd.DataFrame.from_records(
110             [
111                 {"name": "Latitude", "unit": " ", "value": self.latitude},
112                 {"name": "Longitude", "unit": " ", "value": self.longitude}

```



```

113         ↪ },
114         {"name": "Altitude", "unit": " ", "value": self.altitude},
115     ]
116     summary.set_index("name", inplace=True)
117     return summary
118
119
120 def central_angle_orbital_radius(
121     orbital_radius: float, planet_radius: float = EARTH_RADIUS, elevation:
122     ↪ float = 0
123 ):
124     """
125     Calculate angle gamma at the centre of the ground, between the Earth
126     station and the satellite, given the orbital radius of the satellite
127
128     Parameters
129     -----
130     orbital_radius (float, km): distance from the centre of mass to the
131     ↪ satellite
132     planet_radius (float, km, optional): radius of the planet
133     elevation (float, deg): the angle of elevation over the horizon
134
135     Returns
136     -----
137     gamma (float, deg): angle between satellite and ground station
138     """
139     elevation_rad = radians(elevation)
140     gamma = acos((planet_radius * cos(elevation_rad)) / orbital_radius) -
141     ↪ elevation_rad
142     return degrees(gamma)
143
144 def slant_range(
145     orbital_radius: float, central_angle: float, planet_radius: float =
146     ↪ EARTH_RADIUS
147 ) -> float:
148     """
149     Calculate the slant range from the ground station to the satellite
150
151     Parameters
152     -----
153     orbital_radius (float, km): distance from the centre of mass to the
154     ↪ satellite
155     gamma (float, deg): central_angle; angle from the satellite to the
156     ↪ ground station, centred at the centre of mass
157     planet_radius (float, km, optional): radius of the planet
158
159     Return
160     -----
161     slant_range (float, km): the distance from the ground station to
162     ↪ the satellite
163
164     """
165     return sqrt(

```

```

160     planet_radius**2
161     + orbital_radius**2
162     - 2 * planet_radius * orbital_radius * cos(radians(central_angle))
163 )
164
165
166 def elevation_angle(
167     orbital_radius: float, central_angle: float, planet_radius: float =
168     ↪ EARTH_RADIUS
169 ) -> float:
170     """
171     Calculate the elevation angle from the ground station to the satellite
172
173     Parameters
174     -----
175     orbital_radius (float, km): distance from the centre of mass to the
176     ↪ satellite
177     central_angle (float, rad): angle from the satellite to the ground
178     ↪ station, centred at the centre of mass
179     planet_radius (float, km, optional): radius of the planet
180
181     Return
182     -----
183     elevation_angle (float, rad): the distance from the ground station
184     ↪ to the satellite
185
186     """
187     gamma_rad = radians(central_angle)
188     elev = acos(
189         sin(gamma_rad)
190         / sqrt(
191             1
192             + (planet_radius / orbital_radius) ** 2
193             - 2 * (planet_radius / orbital_radius) * cos(gamma_rad)
194         )
195     )
196     return degrees(elev)
197
198
199 def area_of_coverage(central_angle: float, planet_radius: float =
200     ↪ EARTH_RADIUS):
201     """
202     Calculate the surface area coverage of the Earth from a satellite
203
204     Parameters
205     -----
206     central_angle (float, deg): angle from the satellite to the ground
207     ↪ station, centred at the centre of mass
208     planet_radius (float, km, optional): radius of the planet
209
210     Return
211     -----
212     area_coverage (float, km): the area of the Earth's surface visible
213     ↪ from a satellite

```

```

208     """
209     return 2 * pi * (planet_radius**2) * (1 - cos(radians(central_angle)))
210
211
212 def percentage_of_coverage(
213     central_angle: float, planet_radius: float = EARTH_RADIUS
214 ) -> float:
215     """
216     Calculate the surface area coverage of the Earth from a satellite as a
217     ↪ percentage of the total
218         surface area
219
220     Parameters
221     -----
222         central_angle (float, rad): angle from the satellite to the ground
223     ↪ station, centred at the centre of mass
224         planet_radius (float, km, optional): radius of the planet
225
226     Return
227     -----
228         area_coverage (float, %): the percentage of the Earth's surface
229     ↪ visible from a satellite
230
231     """
232     return (
233         area_of_coverage(central_angle, planet_radius)
234         / (4 * pi * planet_radius**2)
235         * 100
236     )
237
238 def percentage_of_coverage_gamma(central_angle: float) -> float:
239     """
240     Calculate the surface area coverage of the Earth from a satellite as a
241     ↪ percentage of the total
242         surface area
243
244     Parameters
245     -----
246         central_angle (float, rad): angle from the satellite to the ground
247     ↪ station, centred at the centre of mass
248
249     Return
250     -----
251         area_coverage (float, %): the percentage of the Earth's surface
252     ↪ visible from a satellite
253
254     """
255     gamma_rad = radians(central_angle)
256     return 50 * (1 - cos(gamma_rad))
257
258 def azimuth_intermediate(
259     ground_station_lat: float, ground_station_long: float, sat_long: float
260 ) -> float:

```

```

257     """
258     Calculate the azimuth of a geostationary satellite
259
260     Parameters
261     -----
262         ground_station_lat (float, deg): the latitude of the ground station
263         ground_station_long (float, deg): the longitude of the ground
↪ station
264         sat_long (float, deg): the longitude of the satellite
265
266     Returns
267     -----
268         azimuth (float, rad): horizontal pointing angle of the ground
↪ station antenna to
269         the satellite. The azimuth angle is usually measured in
↪ clockwise direction
270         in degrees from true north.
271     """
272     gs_lat_rad = radians(ground_station_lat)
273     gs_long_rad = radians(ground_station_long)
274     sat_long_rad = radians(sat_long)
275     az = atan(tan(abs(sat_long_rad - gs_long_rad)) / sin(gs_lat_rad))
276     return degrees(az)
277
278
279 def max_visible_distance(
280     orbital_radius: float,
281     ground_station_lat: float,
282     min_angle: float = 0.0,
283     planet_radius: float = EARTH_RADIUS,
284 ) -> float:
285     """
286     Calculate the radius of visibility for a satellite and a ground station
287
288     Parameters
289     -----
290         orbital_radius (float, km): the radius of the satellite from the
↪ centre of the Earth
291         ground_station_lat (float, deg): the latitude of the ground station
292         min_angle (float, deg): the minimum angle of visibility over the
↪ horizon
293         planet_radius (float, km, optional): radius of the planet
294
295     Returns
296     -----
297
298     """
299     gs_lat_rad = radians(ground_station_lat)
300     min_angle_rad = radians(min_angle)
301     return acos(
302         cos(
303             (
304                 acos((planet_radius / orbital_radius) * cos(min_angle_rad))
305                 - min_angle_rad
306             )

```

```

307         / cos(gs_lat_rad)
308     )
309 )
310
311
312 def azimuth(
313     ground_station_lat: float,
314     ground_station_long: float,
315     sat_lat: float,
316     sat_long: float,
317 ) -> float:
318     """
319     Calculate the azimuth of a satellite
320
321     Parameters
322     -----
323         ground_station_lat (float, deg): the latitude of the ground station
324         ground_station_long (float, deg): the longitude of the ground
325         ↪ station
326         sat_lat (float, deg): the latitude of the satellite
327         sat_long (float, deg): the longitude of the satellite
328
329     Returns
330     -----
331         azimuth (float, rad): horizontal pointing angle of the ground
332         ↪ station antenna to
333         the satellite. The azimuth angle is usually measured in
334         ↪ clockwise direction
335         in degrees from true north.
336     """
337     # Ground station is in northern hemisphere
338     # rel_pos = ground_station_long - sat_long
339     #
340     # if 90 > gs_long_rad > 0:
341     #
342     # At the equator
343     # In the southern hemisphere
344     # elif 0 > gs_long_rad >= -90:
345     #     if (sat_lat - ground_station_lat)
346     #         return pi / 2
347     #     else:
348     #         raise ValueError("Invalid value for the ground station's longitude:
349     ↪ must be in range -90 < L < 90")

```

Listing 7: orbits/utils.py

```

1 # Orbits
2 EARTH_GRAVITATIONAL_CONSTANT = 6.6743e-11 # Nm^2 / kg^2
3 EARTH_MASS = 5.98e24 # km^3/s^2
4 EARTH_MU = 3.986004418e5 # km^3/s^-2
5 EARTH_RADIUS = 6378.14 # km
6 EARTH_POLAR_RADIUS = 6357 # km
7 EARTH_SOLAR_YEAR = 365.25 # days
8 SIDEREAL_DAY = 23.935 # hours
9 SIDEREAL_DAY_S = SIDEREAL_DAY * 60 * 60

```

```

10
11 # Propagation
12 SPEED_OF_LIGHT = 299792458 # m/s
13 BOLTZMANN_CONSTANT = 1.38e-23 # JK-1

```

Listing 8: constants.py

```

1 import numpy as np
2
3 from link_calculator.constants import SPEED_OF_LIGHT
4
5
6 def decibel_to_watt(value: float):
7     """
8     Convert decibel watt units to watts
9
10    Parameters
11    -----
12    decibels (float, dBW): decibel value
13
14    Returns
15    -----
16    watts (float, W): Watt value
17    """
18    if value is None:
19        return None
20    return 10 ** (value / 10)
21
22
23 def watt_to_decibel(value: float):
24     """
25     Convert watts to decibel watts
26
27    Parameters
28    -----
29    watts (float, W): Watt value
30
31    Returns
32    -----
33    decibels (float, dBW): decibel value
34    """
35    if value is None:
36        return None
37    return 10 * np.log10(value)
38
39
40 def frequency_to_wavelength(value: float) -> float:
41     """
42     Convert frequency to wavelength
43
44    Parameters
45    -----
46    frequency (float, GHz): the frequency of the wave
47
48    Returns

```

```

49     -----
50     wavelength (float, m): length of the wave between peaks
51     """
52     if value is None:
53         return None
54     return SPEED_OF_LIGHT / (value * 1e9)
55
56
57 def wavelength_to_frequency(value: float) -> float:
58     """
59     Convert frequency to wavelength
60
61     Parameters
62     -----
63     wavelength (float, m): length of the wave between peaks
64
65     Returns
66     -----
67     frequency (float, GHz): the frequency of the wave
68     """
69     if value is None:
70         return None
71     return (SPEED_OF_LIGHT / value) * 1e-9

```

Listing 9: propagation/conversions.py

```

1 def MHz_to_GHz(value: float) -> float:
2     if value is None:
3         return None
4     return value * 1e-3
5
6
7 def GHz_to_MHz(value: float) -> float:
8     if value is None:
9         return None
10    return value * 1e3
11
12
13 def GHz_to_Hz(value: float) -> float:
14     if value is None:
15         return None
16     return value * 1e9
17
18
19 def Hz_to_GHz(value: float) -> float:
20     if value is None:
21         return None
22     return value * 1e-9
23
24
25 def MHz_to_Hz(value: float) -> float:
26     if value is None:
27         return None
28     return value * 1e6
29

```

```
30
31 def Hz_to_MHz(value: float) -> float:
32     if value is None:
33         return None
34     return value * 1e-6
35
36
37 def mbit_to_bit(value: float) -> float:
38     if value is None:
39         return None
40     return value * 1e6
41
42
43 def bit_to_mbit(value: float) -> float:
44     if value is None:
45         return None
46     return value * 1e-6
```

Listing 10: *signal\_processing/conversions.py*

```
1 from math import log10
2
3
4 def joules_to_decibel_joules(value: float) -> float:
5     if value is None:
6         return None
7     return 10 * log10(value)
8
9
10 def decibel_joules_to_joules(value: float) -> float:
11     if value is None:
12         return None
13     return 10 ** (value / 10)
```

Listing 11: *components/conversions.py*

```
1 import numpy as np
2
3
4 def axes_to_eccentricity(a: float, b: float):
5     return np.sqrt(a**2, b**2) / a
```

Listing 12: *orbits/conversions.py*