ZEIT8219
Satellite Communications

Assignment 2

NINA AVERILL
z3531215

UNSW Canberra
Apr 2022

Contents

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 capabilitie, 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/Te 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/Te 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 suggest that the station is a more efficient transmitter of power and, seeing as both antennas have equal losses, would 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 7.026×10^{-10} and 8.907×10^{-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 1×10^{-9}.

However, though the convolutional code is optimal for conditions where the dominant source of errors is uniformally random errors, it may not 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 neccessary 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 downklink 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 (in^{-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 ±56°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 (°)	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 an 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 attitude 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 to 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 resiliancy, 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 in 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/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	
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 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 Kuniper 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

```python
import pathlib
from math import inf, pi

import numpy as np
import pandas as pd

from link_calculator.components.antennas import Amplifier, Antenna,
    ↪ ParabolicAntenna
from link_calculator.components.communicators import GroundStation,
    ↪ Satellite
from link_calculator.constants import BOLTZMANN_CONSTANT, EARTH_RADIUS
from link_calculator.link_budget import Link, LinkBudget
from link_calculator.orbits.utils import GeodeticCoordinate, Orbit
from link_calculator.propagation.conversions import decibel_to_watt,
    ↪ watt_to_decibel
from link_calculator.signal_processing.conversions import (
    Hz_to_GHz,
    MHz_to_GHz,
    MHz_to_Hz,
    mbit_to_bit,
)
from link_calculator.signal_processing.modulation import (
    ConvolutionalCode,
    MPhaseShiftKeying,
)

print("\n\n")
```

```python

ABS_PATH = pathlib.Path(__file__).parent.resolve()


def q1():
    sat_mod = MPhaseShiftKeying(
        levels=8,
        bandwidth=MHz_to_GHz(50),
        rolloff_rate=0.3,
    )

    # Ground Station A
    gsA_point = GeodeticCoordinate(latitude=-19.08, longitude=178.18)
    gsA_amp = Amplifier(
        power=40,
    )
    gsA_transmit = ParabolicAntenna(
        frequency=30.5,
        efficiency=0.55,
        circular_diameter=1.2,
        loss=decibel_to_watt(-1),
        amplifier=gsA_amp,
        modulation=sat_mod,
    )
    gsA_receive = ParabolicAntenna(
        frequency=20.5,
        efficiency=0.55,
        circular_diameter=1.2,
        loss=decibel_to_watt(-1),
        amplifier=gsA_amp,
        modulation=sat_mod,
    )
    gsA = GroundStation(
        name="EarthStationA",
        noise_figure=2.8,
        noise_temperature=25.0,
        ground_coordinate=gsA_point,
        transmit=gsA_transmit,
        receive=gsA_receive,
    )

    # Ground Station B
    gsB_point = GeodeticCoordinate(latitude=-19.08, longitude=178.18)
    gsB_amp = Amplifier(
        power=80,
    )
    gsB_transmit = ParabolicAntenna(
        frequency=30.5,
        circular_diameter=0.8,
        efficiency=0.6,
        loss=decibel_to_watt(-1.0),
        amplifier=gsB_amp,
        modulation=sat_mod,
    )
    gsB_receive = ParabolicAntenna(
```

```
        frequency=20.5,
        circular_diameter=0.8,
        efficiency=0.6,
        loss=decibel_to_watt(-1.0),
        amplifier=gsB_amp,
        modulation=sat_mod,
    )
    gsB = GroundStation(
        name="EarthStationB",
        noise_figure=2.5,
        noise_temperature=25.0,
        ground_coordinate=gsB_point,
        transmit=gsB_transmit,
        receive=gsB_receive,
    )

    # Ground Station F
    gsF_point = GeodeticCoordinate(latitude=-35.17, longitude=147.27)
    gsF_amp = Amplifier(
        power=500.0,
    )
    gsF_transmit = ParabolicAntenna(
        frequency=30.5,
        circular_diameter=10.5,
        efficiency=0.65,
        loss=decibel_to_watt(-2.5),
        amplifier=gsF_amp,
        modulation=sat_mod,
    )
    gsF_receive = ParabolicAntenna(
        frequency=20.5,
        circular_diameter=10.5,
        efficiency=0.65,
        loss=decibel_to_watt(-2.5),
        amplifier=gsF_amp,
        modulation=sat_mod,
    )
    gsF = GroundStation(
        name="EarthStationFixed",
        noise_figure=2.1,
        noise_temperature=25.0,
        transmit=gsF_transmit,
        receive=gsF_receive,
        ground_coordinate=gsF_point,
    )

    # Satellite
    orbit = Orbit(orbital_radius=42164)
    ss_point = GeodeticCoordinate(latitude=0, longitude=149.8)
    sat_amp = Amplifier(
        power=8.0,
    )
    sat_transmit = ParabolicAntenna(
        frequency=20.5,  # GHz
        modulation=sat_mod,
```

```
        amplifier=sat_amp,
        circular_diameter=1.0,
        efficiency=0.6,
        loss=decibel_to_watt(-0.5),
    )
    sat_receive = ParabolicAntenna(
        frequency=30.5,  # GHz
        modulation=sat_mod,
        amplifier=sat_amp,
        circular_diameter=1.0,
        efficiency=0.6,
        loss=decibel_to_watt(-0.5),
    )
    sat = Satellite(
        name="sat",
        noise_temperature=300,
        noise_figure=2.5,
        transmit=sat_transmit,
        receive=sat_receive,
        ground_coordinate=ss_point,
        orbit=orbit,
    )

    gsA_uplink_upstream = Link(
        transmitter=gsA,
        receiver=sat,
        atmospheric_loss=decibel_to_watt(-6.0),
        slant_range=Link.distance(sat, gsA),
    )
    gsA_downlink_upstream = Link(
        slant_range=Link.distance(sat, gsF),
        transmitter=sat,
        receiver=gsF,
        atmospheric_loss=decibel_to_watt(-6.0),
    )
    gsA_upstream_budget = LinkBudget(
        uplink=gsA_uplink_upstream, downlink=gsA_downlink_upstream
    )
    gsA_upstream_summary = gsA_upstream_budget.summary()
    gsA_upstream_summary.rename(
        columns={"name": "Earth Station A Upstream"}, inplace=True
    )
    gsA_upstream_summary.index = (
        gsA_upstream_summary.index + " (" + gsA_upstream_summary.pop("unit") + ")"
    )
    gsA_upstream_summary.to_csv(
        f"{ABS_PATH}/output/Q1EarthStationAUpstream.csv", float_format="{:,.3f}".format
    )
    print(gsA_upstream_summary)

    gsA_uplink_downstream = Link(
        transmitter=gsF,
        receiver=sat,
```

```
        atmospheric_loss=decibel_to_watt(-6.0),
        slant_range=Link.distance(sat, gsF),
    )
    gsA_downlink_downstream = Link(
        slant_range=Link.distance(sat, gsA),
        transmitter=sat,
        receiver=gsA,
        atmospheric_loss=decibel_to_watt(-6.0),
    )
    gsA_downstream_budget = LinkBudget(
        uplink=gsA_uplink_downstream, downlink=gsA_downlink_downstream
    )
    gsA_downstream_summary = gsA_downstream_budget.summary()
    gsA_downstream_summary.rename(
        columns={"name": "Earth Station A Downstream"}, inplace=True
    )
    gsA_downstream_summary.index = (
        gsA_downstream_summary.index + " (" + gsA_downstream_summary.pop("unit") + ")"
    )
    gsA_downstream_summary.to_csv(
        f"{ABS_PATH}/output/Q1EarthStationADownstream.csv",
        float_format="{:,.3f}".format,
    )
    print(gsA_downstream_summary)

    gsB_uplink_upstream = Link(
        transmitter=gsB,
        receiver=sat,
        atmospheric_loss=decibel_to_watt(-6.0),
        slant_range=Link.distance(sat, gsB),
    )
    gsB_downlink_upstream = Link(
        slant_range=Link.distance(sat, gsF),
        transmitter=sat,
        receiver=gsF,
        atmospheric_loss=decibel_to_watt(-6.0),
    )
    gsB_upstream_budget = LinkBudget(
        uplink=gsB_uplink_upstream, downlink=gsB_downlink_upstream
    )
    gsB_upstream_summary = gsB_upstream_budget.summary()
    gsB_upstream_summary.rename(
        columns={"name": "Earth Station B Upstream"}, inplace=True
    )
    gsB_upstream_summary.index = (
        gsB_upstream_summary.index + " (" + gsB_upstream_summary.pop("unit") + ")"
    )
    gsB_upstream_summary.to_csv(
        f"{ABS_PATH}/output/Q1EarthStationBUpstream.csv", float_format="{:,.3f}".format
    )
    print(gsB_upstream_summary)

```

```
    gsB_uplink_downstream = Link(
        transmitter=gsF,
        receiver=sat,
        atmospheric_loss=decibel_to_watt(-6.0),
        slant_range=Link.distance(sat, gsF),
    )
    gsB_downlink_downstream = Link(
        slant_range=Link.distance(sat, gsB),
        transmitter=sat,
        receiver=gsB,
        atmospheric_loss=decibel_to_watt(-6.0),
    )
    gsB_downstream_budget = LinkBudget(
        uplink=gsB_uplink_downstream, downlink=gsB_downlink_downstream
    )
    gsB_downstream_summary = gsB_downstream_budget.summary()
    gsB_downstream_summary.rename(
        columns={"name": "Earth Station B Downstream"}, inplace=True
    )
    gsB_downstream_summary.index = (
        gsB_downstream_summary.index + " (" + gsB_downstream_summary.pop("unit") + ")"
    )
    gsB_downstream_summary.to_csv(
        f"{ABS_PATH}/output/Q1EarthStationBDownstream.csv",
        float_format="{:,.3f}".format,
    )


def q2():
    def link_eb_no(overall_eb_no: float, link_eb_no) -> float:
        return (overall_eb_no * link_eb_no) / (link_eb_no - overall_eb_no)

    uplink_eb_no = decibel_to_watt(33.2)
    min_bit_error_rate = 1e-9
    min_data_rate = mbit_to_bit(60)

    codes = {
        "7-8": {"code_rate": 7 / 8, "code_gain": decibel_to_watt(2.5)},
        "3-4": {"code_rate": 3 / 4, "code_gain": decibel_to_watt(3)},
        "1-2": {"code_rate": 1 / 2, "code_gain": decibel_to_watt(3.5)},
    }
    results = {}
    for name, params in codes.items():
        print(decibel_to_watt(params["code_gain"]))
        code = ConvolutionalCode(
            coding_rate=params["code_rate"], coding_gain=params["code_gain"]
        )
        mod = MPhaseShiftKeying(
            levels=8,
            bandwidth=MHz_to_GHz(50),
            rolloff_rate=0.3,
            code=code,
            eb_no=uplink_eb_no,
```

```
        )
        results[name] = {"data_rate": mod.data_rate, "mod": mod}
        summary = mod.summary()
        summary.rename(columns={"name": f"{name} Modulation Summary"},
    ↪ inplace=True)
        print(summary)
        summary.index = summary.index + " (" + summary.pop("unit") + ")"
        summary.to_csv(
            f"{ABS_PATH}/output/Q2{name}Modulation.csv", float_format="
    ↪ {:,.3f}".format
        )

    best_code = min(
        results,
        key=lambda x: results[x]["data_rate"]
        if results[x]["data_rate"] > min_data_rate
        else inf,
    )
    best_mod = results[best_code]["mod"]

    overall_mod = MPhaseShiftKeying(
        levels=8,
        bit_error_rate=min_bit_error_rate,
        bandwidth=MHz_to_GHz(50),
        rolloff_rate=0.3,
        code=best_mod.code,
    )
    overall_summary = overall_mod.summary()
    overall_summary.rename(
        columns={"name": "Overall Link Modulation Summary"}, inplace=True
    )
    overall_summary.index = (
        overall_summary.index + " (" + overall_summary.pop("unit") + ")"
    )
    print(overall_summary)
    overall_summary.to_csv(
        f"{ABS_PATH}/output/Q2OverallModulation.csv", float_format="{:,.3f}
    ↪ ".format
    )

    downlink_eb_no = link_eb_no(overall_mod.eb_no, uplink_eb_no)
    downlink_mod = MPhaseShiftKeying(
        levels=8,
        bandwidth=MHz_to_GHz(50),
        eb_no=downlink_eb_no,
        rolloff_rate=0.3,
        code=best_mod.code,
    )
    downlink_summary = downlink_mod.summary()
    downlink_summary.index = (
        downlink_summary.index + " (" + downlink_summary.pop("unit") + ")"
    )
    print(downlink_summary)
    downlink_summary.to_csv(
        f"{ABS_PATH}/output/Q2DownlinkModulation.csv", float_format="{:,.3f
```

```
    ↪ }".format
    )


def inches_to_m(value) -> float:
    return 0.0254 * value


def q3():
    """
    TODO
        - finalise parameters
            - bit error rate
        - add signal to noise to output
    """
    locs = {
        "LowAltitudeOperations": GeodeticCoordinate(
            latitude=-3.74603, longitude=124.401, altitude=0
        ),  # 15000km outside of RAAF Base Tindal
        "HighAltitudeOperations": GeodeticCoordinate(
            latitude=-3.74603, longitude=124.401, altitude=17
        ),
    }

    for scenario, uav_loc in locs.items():
        code = ConvolutionalCode(coding_rate=3 / 4, coding_gain=
    ↪ decibel_to_watt(6.5))
        triton_transmit_mod = MPhaseShiftKeying(
            levels=4,
            bandwidth=MHz_to_GHz(50),
            bit_rate=mbit_to_bit(400),
            spectral_efficiency=4,  # bit/s/Hz
            bit_error_rate=1e-6,
            code=code,
            rolloff_rate=0.4,
        )
        triton_transmit_amp = Amplifier(power=35, loss=decibel_to_watt(-1))
        triton_transmit = Antenna(
            eirp=decibel_to_watt(55.5),
            cross_sect_area=inches_to_m(24.2) * inches_to_m(22.4),
            amplifier=triton_transmit_amp,
            frequency=29,
            modulation=triton_transmit_mod,
            loss=decibel_to_watt(-6),
        )
        triton_receive_mod = MPhaseShiftKeying(
            levels=4,
            bandwidth=MHz_to_GHz(100),
            bit_rate=mbit_to_bit(200),
            spectral_efficiency=4,  # bit/s/Hz
            bit_error_rate=1e-5,
            code=code,
            rolloff_rate=0.4,
        )
        triton_receive_amp = Amplifier(power=35, loss=decibel_to_watt(-1))
```

```
        triton_receive = Antenna(
            eirp=decibel_to_watt(55.5),
            cross_sect_area=inches_to_m(30.6) * inches_to_m(32.4),
            amplifier=triton_receive_amp,
            frequency=19,
            modulation=triton_receive_mod,
            loss=decibel_to_watt(-6),
        )
        triton = GroundStation(
            name="MQ-4C Triton",
            transmit=triton_transmit,
            receive=triton_receive,
            ground_coordinate=uav_loc,
        )

        ss_point = GeodeticCoordinate(latitude=-3.74603, longitude=124.401,
        ↪ altitude=16)
        kuiper_transmit_mod = MPhaseShiftKeying(
            levels=4,
            bandwidth=MHz_to_GHz(100),
            bit_error_rate=1e-5,
            bit_rate=mbit_to_bit(200),
            code=code,
            rolloff_rate=0.4,
        )
        kuiper_transmit_amp = Amplifier(
            power=38.7,
            loss=decibel_to_watt(-1),
        )
        kuiper_transmit = Antenna(
            cross_sect_area=pi * 1.6**2,
            eirp=decibel_to_watt(35.8),
            modulation=kuiper_transmit_mod,
            gain=decibel_to_watt(37),
            frequency=19,
            amplifier=kuiper_transmit_amp,
            loss=decibel_to_watt(-6),
        )
        kuiper_receive_mod = MPhaseShiftKeying(
            levels=4,
            bit_rate=mbit_to_bit(400),
            bit_error_rate=1e-6,
            bandwidth=MHz_to_GHz(50),
            code=code,
            rolloff_rate=0.4,
        )
        kuiper_receive_amp = Amplifier(
            power=38.7,
            loss=decibel_to_watt(-1),
        )
        kuiper_receive = Antenna(
            cross_sect_area=pi * 1.6**2,
            eirp=decibel_to_watt(46.0),
            gain=decibel_to_watt(37),
            loss=decibel_to_watt(-6),
```

```
            frequency=29,
            modulation=kuiper_receive_mod,
            amplifier=kuiper_receive_amp,
        )
        orbit = Orbit(orbital_radius=630 + EARTH_RADIUS)
        kuiper = Satellite(
            name="KuiperSat-1",
            orbit=orbit,
            transmit=kuiper_transmit,
            receive=kuiper_receive,
            ground_coordinate=ss_point,
        )

        uplink = Link(
            transmitter=triton,
            receiver=kuiper,
            atmospheric_loss=decibel_to_watt(-84.0),
            slant_range=Link.distance(kuiper, triton)
            - triton.ground_coordinate.altitude,
            #             eb_no=decibel_to_watt(10.5),
        )
        downlink = Link(
            transmitter=kuiper,
            receiver=triton,
            atmospheric_loss=decibel_to_watt(-84.0),
            slant_range=Link.distance(kuiper, triton)
            - triton.ground_coordinate.altitude,
            #            eb_no=decibel_to_watt(10.5),
        )
        link_budget = LinkBudget(uplink=uplink, downlink=downlink)
        link_summary = link_budget.summary()
        link_summary.index = link_summary.index + " (" + link_summary.pop("
    ↪ unit") + ")"
        link_summary.to_csv(
            f"{ABS_PATH}/output/Q3Tritan{scenario}.csv", float_format="
    ↪ {:,.3f}".format
        )
        print(link_summary)


q1()
q2()
q3()
```

Listing 1: main.py

```
import numpy as np
import pandas as pd

from link_calculator.components.communicators import (
    Communicator,
    GroundStation,
    Satellite,
)
from link_calculator.constants import BOLTZMANN_CONSTANT
```

```python
from link_calculator.orbits.utils import slant_range
from link_calculator.propagation.conversions import watt_to_decibel
from link_calculator.signal_processing.conversions import GHz_to_Hz


class Link:
    def __init__(
        self,
        transmitter: Communicator,
        receiver: Communicator,
        slant_range: float = None,
        atmospheric_loss: float = 1,
        path_loss: float = None,
        min_elevation: float = 0,
        transmitter_eirp: float = None,
        receiver_carrier_power: float = None,
        noise_temperature: float = None,
        noise_power: float = None,
        noise_density: float = None,
        carrier_to_noise_density: float = None,
        carrier_to_noise: float = None,
        bandwidth_to_bit_rate: float = None,
        eb_no: float = None,
    ):
        """



        Parameters
        ----------
            slant_range (float, km): slant_range between the transmit and
    ↪ receive antennas
            noise_temperature (float, Kelvin): the temperature of the
    ↪ environment

        """
        self._transmitter = transmitter
        self._receiver = receiver
        self._slant_range = slant_range
        self._atmospheric_loss = atmospheric_loss
        self._path_loss = path_loss
        self._min_elevation = min_elevation
        self._transmitter_eirp = transmitter_eirp
        self._receiver_carrier_power = receiver_carrier_power
        self._noise_temperature = noise_temperature
        self._carrier_to_noise_density = carrier_to_noise_density
        self._carrier_to_noise = carrier_to_noise
        self._bandwidth_to_bit_rate = bandwidth_to_bit_rate
        self._eb_no = eb_no
        self._noise_power = noise_power
        self._noise_density = noise_density
        self.propagate_calculations()

        if self._transmitter.transmit.modulation is not None:
            self._transmitter.transmit.modulation.eb_no = self.eb_no
        if self._receiver.receive is not None:
```

```python
            self._receiver.receive.power_density = (
                self.receiver.receive.power_density_eirp(
                    self._slant_range, self._atmospheric_loss
                )
            )

            if self._receiver.receive.modulation is not None:
                self._receiver.receive.modulation.carrier_power = (
                    self.receiver_carrier_power
                )
                self._receiver.receive.modulation.eb_no = self.eb_no

    @property
    def carrier_to_noise_density(self) -> float:
        if self._carrier_to_noise_density is None:
            if self.receiver.equiv_noise_temp is not None:
                self._carrier_to_noise_density = (
                    self.receiver.combined_gain * self.receiver_carrier_power
                ) / (BOLTZMANN_CONSTANT * self.receiver.equiv_noise_temp)
            elif self.receiver.gain_to_equiv_noise_temp is not None:
                self._carrier_to_noise_density = (
                    self.transmitter.transmit.eirp
                    * self.path_loss
                    * self.atmospheric_loss
                    * self.receiver.receive.loss
                    * self.receiver.gain_to_equiv_noise_temp
                    / BOLTZMANN_CONSTANT
                )
        return self._carrier_to_noise_density

    @property
    def eb_no(self) -> float:
        if self._eb_no is None:
            if self._carrier_to_noise_density is not None:
                self._eb_no = (
                    self.carrier_to_noise_density
                    / self.transmitter.transmit.modulation.bit_rate
                )
            elif self._carrier_to_noise is not None:
                self._eb_no = (
                    self.carrier_to_noise
                    * GHz_to_Hz(self.transmitter.transmit.modulation.bandwidth)
                    * self.transmitter.transmit.modulation.bit_rate
                )
            if self.transmitter.transmit.modulation.eb_no is not None:
                if self.transmitter.transmit.modulation.code is not None:
                    self._eb_no = self.transmitter.transmit.modulation.eb_no_coded
                else:
                    self._eb_no = self.transmitter.transmit.modulation.eb_no
        return self._eb_no

```

```python
    @property
    def carrier_to_noise(self) -> float:
        if self._carrier_to_noise is None:
            if self._eb_no is not None and self._bandwidth_to_bit_rate is not None:
                self._carrier_to_noise = self.eb_no / self.bandwidth_to_bit_rate
        return self._carrier_to_noise

    @property
    def bandwidth_to_bit_rate(self) -> float:
        if self._bandwidth_to_bit_rate is None:
            self._bandwidth_to_bit_rate = (
                GHz_to_Hz(self.transmitter.transmit.modulation.bandwidth)
                / self.transmitter.transmit.modulation.bit_rate
            )
        return self._bandwidth_to_bit_rate

    @property
    def receiver_carrier_power(self) -> float:
        if self._receiver_carrier_power is None:
            self._receiver_carrier_power = (
                self.transmitter.transmit.eirp * self.path_loss * self.atmospheric_loss
            )
        return self._receiver_carrier_power

    @property
    def central_angle(self) -> float:
        if (
            self.transmitter.ground_coordinate is not None
            and self.receiver.ground_coordinate is not None
        ):
            return self.transmitter.ground_coordinate.central_angle(
                self.receiver.ground_coordinate
            )
        return None

    @staticmethod
    def distance(satellite: Satellite, ground_station: GroundStation) -> float:
        gamma = satellite.ground_coordinate.central_angle(
            ground_station.ground_coordinate
        )
        return slant_range(satellite.orbit.orbital_radius, gamma)

    @property
    def slant_range(self) -> float:
        return self._slant_range

    @property
    def path_loss(self) -> float:
        """
        Calculate the free space loss between two antennas

```

```
        Parameters
        ----------
            slant_range (float, km): slant_range between the transmit and receive antennas
            wavelength (float, m): frequency of the transmitter

        Returns
        -------
            path_loss (float, )

        """
        if self._path_loss is None:
            self._path_loss = (
                self.transmitter.transmit.wavelength
                / (4 * np.pi * self.slant_range * 1000)
            ) ** 2
        return self._path_loss

    @property
    def noise_power(self) -> float:
        """
        Returns
        ----------
            noise_power (float, ): sum of the input noise power and the noise power
                added by the amplifier
        """
        if self._noise_power is None:
            if self._noise_density is not None:
                self._noise_power = self.noise_density * GHz_to_Hz(
                    self.transmitter.transmit.modulation.bandwidth
                )
        return self._noise_power

    @property
    def noise_density(self) -> float:
        """
        Calculate the noise density of the system

        Returns
        -------
            noise_density (float, W/Hz): the total noise power, normalised to a 1-Hz bandwidth
        """
        if self._noise_density is None:
            if self._noise_temperature is not None:
                self._noise_density = BOLTZMANN_CONSTANT * self.noise_temperature
        return self._noise_density

    @property
    def noise_temperature(self) -> float:
        """

        Returns
```

```
        -------
            noise_temperature (float, K): ambient temperature of the environment
        """
        if self._noise_temperature is None:
            if self._noise_power is not None:
                self._noise_temperature = (
                    self.noise_power
                    / GHz_to_Hz(self.transmitter.transmit.modulation.bandwidth)
                ) / BOLTZMANN_CONSTANT
        return self._noise_temperature

    @property
    def receiver(self) -> Communicator:
        return self._receiver

    @property
    def transmitter(self) -> Communicator:
        return self._transmitter

    @property
    def atmospheric_loss(self) -> float:
        return self._atmospheric_loss

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Carrier Power Density",
                    "unit": "dBW",
                    "value": watt_to_decibel(self.receiver_carrier_power),
                },
                {
                    "name": "Free-Space Path Loss",
                    "unit": "dB",
                    "value": watt_to_decibel(self.path_loss),
                },
                {
                    "name": "Atmospheric Loss",
                    "unit": "dB",
                    "value": watt_to_decibel(self.atmospheric_loss),
                },
                {
                    "name": "C/No Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.carrier_to_noise_density),
                },
                {
                    "name": "Eb/No Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.eb_no),
                },
                {
```

```
                    "name": "Bandwidth to Bit Rate Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.bandwidth_to_bit_rate),
                },
                {
                    "name": "C/N Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.carrier_to_noise),
                },
                {"name": "Central Angle", "unit": "  ", "value": self.central_angle},
                {"name": "Slant Range", "unit": "km", "value": self.slant_range},
            ]
        )
        receiver = self.receiver.summary()
        receiver.index = "Receiver " + receiver.index

        transmitter = self.transmitter.summary()
        transmitter.index = "Transmitter " + transmitter.index

        summary.set_index("name", inplace=True)
        summary = pd.concat([summary, transmitter, receiver])
        return summary

    def propagate_calculations(self) -> float:
        for _ in range(3):
            for var in type(self).__dict__:
                if not callable(var):
                    getattr(self, var)


class LinkBudget:
    def __init__(
        self,
        uplink: Link,
        downlink: Link,
    ):
        self._uplink = uplink
        self._downlink = downlink

    @property
    def eb_no(self) -> float:
        return (self.uplink.eb_no * self.downlink.eb_no) / (
            self.uplink.eb_no + self._downlink.eb_no
        )

    @property
    def uplink(self) -> Link:
        return self._uplink

    @property
    def downlink(self) -> Link:
        return self._downlink

```

```
    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Eb/No Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.eb_no),
                },
            ]
        )

        uplink = self.uplink.summary()
        uplink.rename(columns={"value": "Uplink", "unit": "Uplink unit"}, inplace=True)
        downlink = self.downlink.summary()
        downlink.rename(
            columns={"value": "Downlink", "unit": "Downlink unit"}, inplace=True
        )
        summary.set_index("name", inplace=True)
        summary.rename(columns={"value": "Overall"}, inplace=True)

        # Merge uplink and downlink summaries
        summary = pd.concat([summary, uplink, downlink], axis=1)

        # Merge unit columns
        summary["unit"] = summary["unit"].combine_first(
            summary["Uplink unit"].combine_first(summary["Downlink unit"])
        )
        summary.drop(columns=["Downlink unit", "Uplink unit"], inplace=True)

        # Clean and reformat columns
        summary = summary[
            [
                not (
                    idx.startswith("Transmitter Receive")
                    or idx.startswith("Receiver Transmit")
                )
                for idx in summary.index
            ]
        ]
        summary.rename(
            index={
                k: k.replace("Receiver Receive", "Receiver")
                if "Receiver Receive" in k
                else k
                for k in summary.index
            },
            inplace=True,
        )
        summary.rename(
            index={
                k: k.replace("Transmitter Transmit", "Transmitter")
                if "Transmitter Transmit" in k
```

```
                else k
                for k in summary.index
            },
            inplace=True,
        )
        return summary
```

Listing 2: link$_{b}$*udget.py*

```
from math import sqrt

import pandas as pd

from link_calculator.components.antennas import Amplifier, Antenna
from link_calculator.constants import BOLTZMANN_CONSTANT, EARTH_MU,
    ↪ EARTH_RADIUS
from link_calculator.orbits.utils import GeodeticCoordinate, Orbit
from link_calculator.propagation.conversions import watt_to_decibel
from link_calculator.signal_processing.conversions import GHz_to_Hz


class Communicator:
    def __init__(
        self,
        name: str,
        transmit: Antenna,
        receive: Antenna,
        ground_coordinate: GeodeticCoordinate = None,
        combined_gain: float = None,
        noise_figure: float = None,
        noise_temperature: float = None,
        noise_density: float = None,
        gain_to_equiv_noise_temp: float = None,
        equiv_noise_temp: float = None,
    ):
        self._name = name
        self._transmit = transmit
        self._receive = receive
        self._ground_coordinate = ground_coordinate
        self._noise_figure = noise_figure
        self._noise_density = noise_density
        self._noise_temperature = noise_temperature
        self._combined_gain = combined_gain
        self._gain_to_equiv_noise_temp = gain_to_equiv_noise_temp
        self._equiv_noise_temp = equiv_noise_temp
        self.propagate_calculations()

    @property
    def noise_figure(self) -> float:
        """
        Calculate the noise figure of the device

        Returns
        -------
            noise_figure (float, ): the ratio of the S/N ratio at the
```

```
   ↪ input to the
                S/N ratio at the output. Measure of the relative increase
   ↪ in noise
                power compared to increase in signal power
        """
        if self._noise_figure is None:
            if self.receive.signal_to_noise is not None:
                self.noise_figure = (
                    self.receive.signal_to_noise / self.transmit.
   ↪ signal_to_noise
                )
            elif self._noise_temperature is not None:
                self._noise_figure = 1 + (
                    self.equiv_noise_temp / self.noise_temperature
                )
        return self._noise_figure

    @property
    def output_noise_power(self) -> float:
        return self.receive.amplifier.gain * (
            self.receive.noise_power + self.receive.amplifier.noise_power
        )

    @property
    def receive_noise_power(self) -> float:
        return (
            BOLTZMANN_CONSTANT
            * self.noise_temperature
            * GHz_to_Hz(self.receive.modulation.bandwidth)
        )

    @property
    def equiv_noise_temp(self):
        """
        TODO
        """
        if self._equiv_noise_temp is None:
            if self._noise_temperature is not None:
                self._equiv_noise_temp = self.noise_temperature * (
                    self.noise_figure - 1
                )
        return self._equiv_noise_temp

    @equiv_noise_temp.setter
    def equiv_noise_temp(self, value):
        """
        TODO
        """
        self._equiv_noise_temp = value

    @property
    def combined_gain(self) -> float:
        if self._combined_gain is None:
            if self.receive.amplifier is not None:
                self._combined_gain = self.receive.amplifier.gain * self.
```

```
    ↪ receive.gain
            else:
                self._combined_gain = self.receive.gain
        return self._combined_gain

    @property
    def gain_to_equiv_noise_temp(self) -> float:
        if self._gain_to_equiv_noise_temp is None:
            if self._equiv_noise_temp is not None:
                self._gain_to_equiv_noise_temp = (
                    self.combined_gain / self.equiv_noise_temp
                )
            elif self._noise_density is not None:
                self._gain_to_equiv_noise_temp = (
                    self.carrier_power * BOLTZMANN_CONSTANT
                ) / (self.noise_density * self.receive_carrier_power)
        return self._gain_to_equiv_noise_temp

    @property
    def ground_coordinate(self) -> GeodeticCoordinate:
        return self._ground_coordinate

    @property
    def receive(self) -> Antenna:
        return self._receive

    @property
    def transmit(self) -> Antenna:
        return self._transmit

    @property
    def noise_temperature(self) -> float:
        return self._noise_temperature

    @property
    def noise_density(self) -> float:
        return self._noise_density

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {"name": "Noise Figure", "unit": "", "value": self.
    ↪ noise_figure},
                {
                    "name": "Equivalent Noise Temperature",
                    "unit": "K",
                    "value": self.equiv_noise_temp,
                },
                {
                    "name": "Noise Temperature",
                    "unit": "K",
                    "value": self.noise_temperature,
                },
                {
                    "name": "Combined Gain",
```

```
                    "unit": "dB",
                    "value": watt_to_decibel(self.combined_gain),
                },
                {
                    "name": "G/Te Ratio",
                    "unit": "dBK-1",
                    "value": watt_to_decibel(self.gain_to_equiv_noise_temp)
    ↪ ,
                },
            ]
        )
        transmitter = self.transmit.summary()
        transmitter.index = "Transmit " + transmitter.index

        receiver = self.receive.summary()
        receiver.index = "Receive " + receiver.index

        summary.set_index("name", inplace=True)
        summary = pd.concat([summary, transmitter, receiver])
        return summary

    def propagate_calculations(self) -> float:
        for _ in range(3):
            for var in type(self).__dict__:
                getattr(self, var)


class GroundStation(Communicator):
    def __init__(
        self,
        name: str,
        transmit: Antenna,
        receive: Antenna,
        ground_coordinate: GeodeticCoordinate = None,
        gain_to_equiv_noise_temp: float = None,
        combined_gain: float = None,
        noise_figure: float = None,
        noise_temperature: float = None,
        equiv_noise_temp: float = None,
    ):
        """

        Parameters
        ----------
            name (str,): name of the ground station
            latitude (str, deg): the latitude of the groundstation
            longitude (str, deg): the longitude of the groundstation
            altitude (str, km): the altitude of the groundstation above sea
    ↪  level
        """
        super().__init__(
            name=name,
            transmit=transmit,
            receive=receive,
            ground_coordinate=ground_coordinate,
```

```
            gain_to_equiv_noise_temp=gain_to_equiv_noise_temp,
            noise_figure=noise_figure,
            noise_temperature=noise_temperature,
            equiv_noise_temp=equiv_noise_temp,
        )

    def summary(self) -> pd.DataFrame:
        summary = super().summary()
        if self.ground_coordinate is not None:
            coordinate = self.ground_coordinate.summary()
            coordinate.index = "Earth Station " + coordinate.index
            summary = pd.concat([summary, coordinate])
        return summary


class Satellite(Communicator):
    def __init__(
        self,
        name: str,
        transmit: Antenna,
        receive: Antenna,
        orbit: Orbit = None,
        ground_coordinate: GeodeticCoordinate = None,
        gain_to_equiv_noise_temp: float = None,
        combined_gain: float = None,
        noise_figure: float = None,
        noise_temperature: float = None,
        equiv_noise_temp: float = None,
    ):
        self._orbit = orbit
        super().__init__(
            name=name,
            transmit=transmit,
            receive=receive,
            ground_coordinate=ground_coordinate,
            combined_gain=combined_gain,
            gain_to_equiv_noise_temp=gain_to_equiv_noise_temp,
            noise_figure=noise_figure,
            noise_temperature=noise_temperature,
            equiv_noise_temp=equiv_noise_temp,
        )

    def velocity(self, orbital_radius: float, mu: float = EARTH_MU) -> float:
        """
        Calculate the velocity of a satellite in orbit according to Kepler's second law

        Parameters
        ---------
            semi_major_axis (float, km): The semi-major axis of the orbit
                orbital_radius (float, km): distance from the centre of mass to the satellite
            mu (float, optional): Kepler's gravitational constant

```

```
        Returns
        -------
            velocity (float, km/s): the orbit speed of the satellite
        """
        return sqrt(mu * (2 / orbital_radius - 1 / self.semi_major_axis))

    @property
    def semi_major_axis(self) -> float:
        if self.orbit is not None:
            return self.orbit.semi_major_axis
        return None

    @property
    def orbit(self) -> float:
        return self._orbit

    def summary(self) -> pd.DataFrame:
        summary = super().summary()
        if self.ground_coordinate is not None:
            coordinate = self.ground_coordinate.summary()
            coordinate.index = "Sub-Satellite " + coordinate.index
            summary = pd.concat([summary, coordinate])
        return summary
```

Listing 3: components/communicators.py

```
from math import cos, degrees, exp, pi, radians, sin

import numpy as np
import pandas as pd

from link_calculator.constants import BOLTZMANN_CONSTANT
from link_calculator.propagation.conversions import (
    frequency_to_wavelength,
    watt_to_decibel,
    wavelength_to_frequency,
)
from link_calculator.signal_processing.modulation import Modulation


class Amplifier:
    def __init__(
        self, power: float, gain: float = 1, loss: float = 1, noise_power:
    ↪ float = None
    ):
        """
        Parameters
        ----------
            loss (float, ): losses in the amplifier (e.g. back-off loss)
            power (float, W): the total output amplifier power
        """
        self._power = power
        self._gain = gain
        self._loss = loss
        self._noise_power = noise_power
```

```
    @property
    def power(self) -> float:
        """
        TODO
        Returns
        -------
            power (float, W): the total output amplifier power
        """
        return self._power

    @property
    def gain(self) -> float:
        """
        TODO
        Returns
        -------
        """
        return self._gain

    @property
    def loss(self) -> float:
        """
        TODO
        Returns
        -------
        """
        return self._loss

    @property
    def noise_power(self) -> float:
        """
        TODO
        Returns
        -------
        """
        return self._noise_power

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Power",
                    "unit": "dBW",
                    "value": watt_to_decibel(self.power),
                },
                {
                    "name": "Gain",
                    "unit": "dB",
                    "value": watt_to_decibel(self.gain),
                },
                {
                    "name": "Back-Off Loss",
                    "unit": "dB",
                    "value": watt_to_decibel(self.loss),
```

```
                },
                {
                    "name": "Noise Power",
                    "unit": "dB",
                    "value": watt_to_decibel(self.noise_power),
                },
            ]
        )
        summary.set_index("name", inplace=True)
        return summary


class Antenna:
    def __init__(
        self,
        gain: float = None,
        loss: float = 1,
        eirp: float = None,
        frequency: float = None,
        wavelength: float = None,
        effective_aperture: float = None,
        cross_sect_area: float = None,
        cross_sect_diameter: float = None,
        half_beamwidth: float = None,
        efficiency: float = None,
        roughness_factor: float = None,
        carrier_to_noise: float = None,
        signal_to_noise: float = None,
        carrier_power: float = None,
        modulation: Modulation = None,
        combined_loss: float = None,
        amplifier: Amplifier = None,
        gain_to_noise_temperature=None,
        power_density: float = None,
    ):
        """
        Instantiate an Antenna object

        Parameters
        ----------
            name (str): Name of antenna
            gain (float, ): ratio of maximum power densiry to that of an isotropic radiatior
                at the same distance in the direction of the receiving antenna
            loss (float, ): coupling loss between transmitter and antenna
                in the range [0, 1]
            frequency (float, GHz): the transmit frequency of the antenna
            wavelength (float, m): the radiation wavelength
            cross_sect_area (float, m^2): cross sectional area of the antenna aperture
            cross_sect_diameter (float, m): cross sectional diameter of the  antenna aperture
            effective_aperture (float, m^2): The effective collecting area of a receiving antenna
```

```
            half_beamwidth (float, deg): The angle between the directions providing half maximum power
                on either side of the maximum power direction.
            efficiency (float, ): the efficiency with which the antenna radiates all
              energy fed into it
            roughness_factor (float, m): rms roughness of the antenna dish surface
        """
        self._amplifier = amplifier
        self._gain = gain
        self._loss = loss
        self._eirp = eirp
        self._efficiency = efficiency
        self._half_beamwidth = half_beamwidth
        self._cross_sect_area = cross_sect_area
        self._cross_sect_diameter = cross_sect_diameter
        self._frequency = frequency
        self._wavelength = wavelength
        self._modulation = modulation
        self._effective_aperture = effective_aperture
        self._roughness_factor = roughness_factor
        self._carrier_to_noise = carrier_to_noise
        self._signal_to_noise = signal_to_noise
        self._carrier_power = carrier_power
        self._gain_to_noise_temperature = gain_to_noise_temperature
        self._combined_loss = combined_loss
        self._power_density = power_density

    def power_density_eirp(self, distance: float, atmospheric_loss: float = 1) -> float:
        """
        Calculate the power density of the wavefront using EIRP

        Parameters
        ----------
            eirp (float, dB)
            distance (float, km): the distance between the transmit and receive antennas
            atmospheric_loss (float, ): the total losses due to the atmosphere

        Returns
        -------
            power_density (float, W/m^2): the power density at distance d
        """
        distance = distance * 1000  # convert to m
        return self.eirp / (4 * np.pi * distance**2) * atmospheric_loss

    def power_density_distance(self, distance: float) -> float:
        """
        Calculate the power density of the wavefront

        Parameters
        ----------
```

```
            power (float, W): the transmitted power
            distance (float, km): the distance between the transmit and receive antennas

        Returns
        ------
            power_density (float, W/m^2): the power density at distance d
        """
        distance = distance * 1000  # convert to m
        return (self.amplifier.power * self.gain) / (4 * np.pi * distance **2)

    @property
    def combined_loss(self) -> float:
        if self._combined_loss is None:
            if self.amplifier is not None:
                self._combined_loss = self.loss * self.amplifier.loss
            elif self._eirp is not None:
                self._combined_loss = self.eirp / (self.amplifier.power * self.gain)
        return self._combined_loss

    @property
    def eirp(self) -> float:
        """
        Calculate the Effetive Isotropic Radiated Power


        Returns
        -------
            eirp (float, dB): power incident at the receiver that would have had to be radiated
                from an isotropic antenna to achieve the same power incident at the
                receiver  as that of a transmitter with a specific antenna gain
        """
        if self._eirp is None:
            self._eirp = self.amplifier.power * self.combined_loss * self.gain
        return self._eirp

    @property
    def half_beamwidth(self) -> float:
        if self._half_beamwidth is None:
            self._half_beamwidth = self.wavelength / (
                self.cross_sect_diameter * np.sqrt(self.efficiency)
            )
        return self._half_beamwidth

    @property
    def carrier_to_noise(self) -> float:
        if self._carrier_to_noise is None:
            self._carrier_to_noise = self.carrier_power / self.noise_density
```

```
        return self._carrier_to_noise

    @property
    def effective_aperture(self) -> float:
        """
        Calculate the effective area of the receiving antenna

        Parameters
        ----------

        Returns
        -------
            effective_aperture (float, m^2): the effective aperture of the receive antenna
        """
        if self._effective_aperture is None:
            self._effective_aperture = self.gain * self.wavelength**2 / (4 * np.pi)
        return self._effective_aperture

    @property
    def directive_gain(self) -> float:
        return self.gain / self.efficiency

    @property
    def directivity(self) -> float:
        """
        TODO
        """
        return 4 * pi * self.cross_sect_area / self.wavelength**2

    def pointing_loss(self, pointing_error: float) -> float:
        """
        TODO
        Parameters
        ---------
          pointing_error (float, deg): angle off nominal pointing direction

        Returns
        ------
          pointing_loss (float, ??):
        """
        return exp(
            -2.76 * (radians(pointing_error) / radians(self.half_beamwidth)) ** 2
        )

    def surface_roughness_loss(self) -> float:
        """
        TODO
        Parameters
        ---------

        Returns
        ------
```

```
            pointing_loss (float, ??):
        """
        if self._surface_roughness_loss is None:
            self._surface_roughness_loss = exp(
                -(4 * pi * self.roughness_factor / self.wavelength)
            )
        return self._surface_roughness_loss

    @property
    def gain(self):
        """
        TODO
        Returns
        -------
            gain (float, ): ratio of maximum power densiry to that of an isotropic radiatior
                at the same distance in the direction of the receiving antenna
        """
        if self._gain is None:
            if self._eirp is not None:
                self._gain = self.eirp / (self.amplifier.power * self.combined_loss)
            elif self._efficiency is not None:
                self._gain = (
                    self.efficiency
                    * 4
                    * np.pi
                    * self.cross_sect_area
                    / self.wavelength**2
                )
        return self._gain

    @property
    def carrier_power(self) -> float:
        return self._carrier_power

    @property
    def power_density(self) -> float:
        return self._power_density

    @power_density.setter
    def power_density(self, value):
        self._power_density = value

    @property
    def frequency(self):
        """
        TODO
        Returns
        -------
            frequency (float, GHz): the transmit frequency of the antenna
        """
        if self._frequency is None:
            self._frequency = wavelength_to_frequency(self._wavelength)
```

```python
        return self._frequency

    @property
    def wavelength(self):
        """
        TODO
        """
        if self._wavelength is None:
            self._wavelength = frequency_to_wavelength(self._frequency)
        return self._wavelength

    @property
    def efficiency(self):
        """
        TODO
        """
        if self._efficiency is None:
            if self._gain is not None and self._cross_sect_area is not None:
                self._efficiency = self.gain / (
                    4 * np.pi * self.cross_sect_area / self.wavelength**2
                )
        return self._efficiency

    @property
    def cross_sect_diameter(self):
        """
        TODO
        """
        return self._cross_sect_diameter

    @property
    def cross_sect_area(self):
        """
        TODO
        """
        return self._cross_sect_area

    @property
    def loss(self):
        """
        TODO
        Returns
        -------
            loss (float, ): coupling loss between transmitter and antenna
                in the range [0, 1]
        """
        if self._loss is None:
            if self._combined_loss is not None:
                if self.amplifier is not None:
                    self._loss = self.combined_loss / self.amplifier.loss
                else:
                    self._loss = self.combined_loss
        return self._loss

```

```
    @property
    def amplifier(self):
        """
        TODO
        """
        return self._amplifier

    @property
    def modulation(self):
        """
        TODO
        """
        return self._modulation

    @property
    def transmit_loss(self):
        """
        TODO
        Returns
        -------
            transmit_loss (float, ): the feeder and branching losses from
        ↪ the amplifier
                to the transmit antenna
        """
        return self._transmit_loss

    def receive_power(
        self,
        transmit_antenna: "Antenna",
        distance: float,
        atmospheric_loss: float = 1,
    ) -> float:
        """
        Calculate the power collected by the receive antenna

        Parameters
        ----------
            distance (float, km): the distance between the transmit and
        ↪ receive antennas
            atmospheric_loss (float, ): The loss due to the atmosphere

        Returns
        -------
            receive_power (float, W): the total collected power at the
        ↪ receiver's terminals
        """
        pow_density = transmit_antenna.power_density_eirp(distance,
        ↪ atmospheric_loss)
        return pow_density * self.effective_aperture * self.loss

    @property
    def roughness_factor(self):
        """
        TODO
        """
```

```
        return self._roughness_factor

    @property
    def signal_to_noise(self):
        """
        calculate S/N knowing G/T, wavelength, bandwidth and the field
        strength of the signal (Duffy 2007).

        Signal/Noise=S(  **2/4  )(G/T)(1/kbB) where:

        S is power flux density;
           is wavelength;
        kb is Boltzmanns constant; and
        B is receiver quivalent noise bandwidth
        """
        if self._signal_to_noise is None:
            if (
                self._gain_to_noise_temperature is not None
                and self._power_density is not None
            ):
                self._signal_to_noise = (
                    self.power_density
                    * (self.wavelength**2 / (4 * pi))
                    * self.gain_to_noise_temperature
                    * (1 / (BOLTZMANN_CONSTANT * self.modulation.bandwidth))
                )
        return self._signal_to_noise

    @property
    def gain_to_noise_temperature(self):
        """
        TODO
        """
        return self._gain_to_noise_temperature

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Frequency",
                    "unit": "GHz",
                    "value": self.frequency,
                },
                {
                    "name": "Wavelength",
                    "unit": "m",
                    "value": self.wavelength,
                },
                {
                    "name": "Efficiency",
                    "unit": "%",
                    "value": self.efficiency,
                },
                {
```

```
                    "name": "Feeder Loss",
                    "unit": "dB",
                    "value": watt_to_decibel(self.loss),
                },
                {
                    "name": "Gain",
                    "unit": "dB",
                    "value": watt_to_decibel(self.gain),
                },
                {
                    "name": "EIRP",
                    "unit": "dBW",
                    "value": watt_to_decibel(self.eirp),
                },
                {
                    "name": "S/N",
                    "unit": "dBW",
                    "value": watt_to_decibel(self.signal_to_noise),
                },
            ]
        )
        summary.set_index("name", inplace=True)

        if self.amplifier is not None:
            amplifier = self.amplifier.summary()
            amplifier.index = "Amplifier " + amplifier.index
            summary = pd.concat([summary, amplifier])

        if self.modulation is not None:
            modulation = self.modulation.summary()
            modulation.index = "Modulation " + modulation.index
            summary = pd.concat([summary, modulation])
        return summary

    def propagate_calculations(self) -> float:
        for _ in range(3):
            for var in type(self).__dict__:
                getattr(self, var)


class HalfWaveDipole(Antenna):
    """
    Class for omnidirectuinal radiation pattern
    """

    def __init__(
        self,
        amplifier: Amplifier = None,
        gain: float = None,
        loss: float = None,
        frequency: float = None,
        effective_aperture: float = None,
        half_beamwidth: float = None,  # deg
    ):
        super().__init__(
```

```
            amplifier=amplifier,
            gain=gain,
            loss=loss,
            frequency=frequency,
            effective_aperture=effective_aperture,
            half_beamwidth=half_beamwidth,
        )

    @property
    def effective_aperture(self) -> float:
        if self._effective_aperture is None:
            self._effective_aperture = 0.13 * self.wavelength
        return self._effective_aperture

    def off_sight_gain(self, theta: float) -> float:
        """
        theta (float, deg): ??
        """
        return cos(np.pi / 2 * cos(radians(theta))) ** 2 / sin(radians(
    ↪ theta)) ** 2


class ConicalHornAntenna(Antenna):
    def __init__(
        self,
        amplifier: Amplifier = None,
        gain: float = None,
        loss: float = None,
        frequency: float = None,
        effective_aperture: float = None,
        half_beamwidth: float = 20,  # deg
    ):
        super().__init__(
            amplifier=amplifier,
            gain=gain,
            loss=loss,
            frequency=frequency,
            effective_aperture=effective_aperture,
            half_beamwidth=half_beamwidth,
        )


class SquareHornAntenna(Antenna):
    def __init__(
        self,
        cross_sect_diameter: float,
        amplifier: Amplifier = None,
        efficiency: float = 1,
        half_beamwidth: float = None,  # deg
        gain: float = None,
        loss: float = 1,
        frequency: float = None,
        effective_aperture: float = None,
    ):
        effective_aperture = efficiency * cross_sect_diameter**2
```

```
        super().__init__(
            amplifier=amplifier,
            gain=gain,
            loss=loss,
            frequency=frequency,
            efficiency=efficiency,
            effective_aperture=effective_aperture,
            half_beamwidth=half_beamwidth,
            cross_sect_diameter=cross_sect_diameter,
        )

    @property
    def gain(self):
        """
        TODO
        """
        if self._gain is None:
            self._gain = (
                self.efficiency
                * 4
                * pi
                * self.cross_sect_diameter**2
                / self.wavelength**2
            )
        return self._gain

    @property
    def half_beamwidth(self) -> float:
        """
        TODO
        """
        if self._half_beamwidth is None:
            self._half_beamwidth = degrees(
                0.88 * self.wavelength / self.cross_sect_diameter
            )
        return self._half_beamwidth


class ParabolicAntenna(Antenna):
    def __init__(
        self,
        circular_diameter: float,
        amplifier: Amplifier = None,
        gain: float = None,
        loss: float = None,
        frequency: float = None,
        wavelength: float = None,
        effective_aperture: float = None,
        efficiency: float = None,
        beamwidth_scale_factor: float = None,
        half_beamwidth: float = None,  # deg
        roughness_factor: float = None,
        carrier_to_noise: float = None,
        signal_to_noise: float = None,
        modulation: Modulation = None,
```

```
        combined_loss: float = None,
    ):
        self._beamwidth_scale_factor = beamwidth_scale_factor
        super().__init__(
            amplifier=amplifier,
            gain=gain,
            loss=loss,
            frequency=frequency,
            wavelength=wavelength,
            efficiency=efficiency,
            effective_aperture=effective_aperture,
            half_beamwidth=half_beamwidth,
            cross_sect_diameter=circular_diameter,
            modulation=modulation,
            carrier_to_noise=carrier_to_noise,
            signal_to_noise=signal_to_noise,
            combined_loss=combined_loss,
        )

    @property
    def gain(self) -> float:
        """
        TODO
        """
        if self._gain is None:
            self._gain = (
                self.efficiency * (pi * self.cross_sect_diameter / self.wavelength) ** 2
            )
        return self._gain

    @property
    def cross_sect_area(self):
        """
        TODO
        """
        if self._cross_sect_area is None:
            self._cross_sect_area = pi / 4 * self.circular_diameter**2
        return self._cross_sect_area

    @property
    def half_beamwidth(self) -> float:
        """
        TODO
        """
        if self._half_beamwidth is None:
            self._half_beamwidth = self._beamwidth_scale_factor * (
                self.wavelength / self.cross_sect_diameter
            )
        return self._half_beamwidth

    def off_sight_gain(self, k: float, theta: float):
        """
        TODO
        """
```

```
        return self.gain * self.pointing_loss(theta)

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Diameter",
                    "unit": "m",
                    "value": self.cross_sect_diameter,
                },
                {
                    "name": "Gain",
                    "unit": "dB",
                    "value": watt_to_decibel(self.gain),
                },
                {
                    "name": "Half Beamwidth",
                    "unit": "dBW",
                    "value": watt_to_decibel(self.eirp),
                },
            ]
        )
        summary.set_index("name", inplace=True)

        antenna = super().summary()
        antenna.drop("Gain", inplace=True)
        summary = pd.concat([summary, antenna])
        return summary


class HelicalAntenna(Antenna):
    def __init__(
        self,
        circular_diameter: float,
        n_helix_turns: float,
        turn_spacing: float,
        amplifier: Amplifier = None,
        gain: float = None,
        loss: float = None,
        frequency: float = None,
        effective_aperture: float = None,
        efficiency: float = None,
        half_beamwidth: float = 20,  # deg
    ):
        self.n_helix_turns = n_helix_turns
        self.turn_spacing = n_helix_turns
        super().__init__(
            amplifier=amplifier,
            gain=gain,
            loss=loss,
            frequency=frequency,
            effective_aperture=effective_aperture,
            half_beamwidth=half_beamwidth,
            cross_sect_diameter=circular_diameter,
        )
```

```python

    @property
    def gain(self) -> float:
        """
        TODO
        """
        if self._gain is None:
            self._gain = (
                15
                * self.n_helix_turns
                * self.turn_spacing
                * (pi**2)
                * (self.cross_sect_diameter**2)
                / self.wavelength**3
            )
        return self._gain
```

Listing 4: components/antennas.py

```python
from math import erfc, log2, log10, pi, sin, sqrt

import pandas as pd
from scipy.special import erfcinv

from link_calculator.propagation.conversions import watt_to_decibel
from link_calculator.signal_processing.conversions import (
    GHz_to_Hz,
    Hz_to_GHz,
    Hz_to_MHz,
    bit_to_mbit,
    mbit_to_bit,
)


class Modulation:
    def __init__(
        self,
        bandwidth: float = None,
        carrier_to_noise: float = None,
    ):
        self._bandwidth = bandwidth
        self._carrier_to_noise = carrier_to_noise

    @property
    def channel_capacity(self) -> float:
        return self.bandwidth * log2(1 + self.carrier_to_noise)

    @property
    def channel_capacity_to_bandwidth_ideal(self) -> float:
        return log2(1 + self.eb_no * self.channel_capacity / self.bandwidth
    ↪ )

    @property
    def channel_capacity_to_bandwidth(self) -> float:
        return log2(1 + self.eb_no * self.bit_rate / self.bandwidth)
```

```python

    @property
    def eb_no(self) -> float:
        return (2 ** (self.channel_capacity / self.bandwidth) - 1) / (
            self.channel_capacity / self.bandwidth
        )


class FrequencyModulation(Modulation):
    def __init__(
        self,
        bandwidth: float = None,
        baseband_bandwidth: float = None,
        carrier_to_noise: float = None,
        signal_to_noise: float = None,
        deviation_ratio: float = None,
        frequency_deviation: float = None,
        threshold: float = None,
        link_margin: float = None,
        deemphasis_improvement: float = 1,
        preemphasis_improvement: float = 1,
    ):
        """
        bandwidth (float, GHz): the range of frequencies required to transmit the signal
        baseband_bandwidth (float, GHz): the range of frequencies required to
            transmit the baseband signal
        carrier_to_noise (float, W):
        deemphasis_improvement (float, W):
        preemphasis_improvement (float, W):
        """
        self._bandwidth = bandwidth
        self._baseband_bandwidth = baseband_bandwidth
        self._carrier_to_noise = carrier_to_noise
        self._signal_to_noise = signal_to_noise
        self._deviation_ratio = deviation_ratio
        self._frequency_deviation = frequency_deviation
        self._deemphasis_improvement = deemphasis_improvement
        self._preemphasis_improvement = preemphasis_improvement
        self._threshold = threshold
        self._link_margin = link_margin

    @property
    def bandwidth(self) -> float:
        if self._bandwidth is None:
            self._bandwidth = 2 * (self.frequency_deviation + self.baseband_bandwidth)
        return self._bandwidth

    @property
    def frequency_deviation(self) -> float:
        if self._frequency_deviation is None:
            self._frequency_deviation = self.bandwidth / 2 - self.baseband_bandwidth
```

```
        return self._frequency_deviation

    @property
    def deviation_ratio(self) -> float:
        if self._deviation_ratio is None:
            self._deviation_ratio = self.frequency_deviation / self.
    ↪ baseband_bandwidth
        return self._deviation_ratio

    @property
    def signal_to_noise(self) -> float:
        if self._signal_to_noise is None:
            self._signal_to_noise = (
                (3 / 2)
                * self.carrier_to_noise
                * (self.bandwidth / self.baseband_bandwidth)
                * self.deviation_ratio**2
                * self.preemphasis_improvement
                * self.deemphasis_improvement
            )
        return self._signal_to_noise

    @property
    def link_margin(self) -> float:
        if self._link_margin is None:
            self._link_margin = self.carrier_to_noise / self.threshold
        return self._link_margin

    @property
    def threshold(self) -> float:
        return self._threshold

    @property
    def baseband_bandwidth(self) -> float:
        return self._baseband_bandwidth

    @property
    def carrier_to_noise(self) -> float:
        return self._carrier_to_noise

    @property
    def deemphasis_improvement(self) -> float:
        return self._deemphasis_improvement

    @property
    def preemphasis_improvement(self) -> float:
        return self._preemphasis_improvement


class Waveform:
    def __init__(
        self, frequency: float = None, amplitude: float = None, phase:
    ↪ float = None
    ):
        self.frequency = frequency
```

```
        self.amplitude = amplitude
        self.phase = phase


class ConvolutionalCode:
    def __init__(
        self,
        coding_rate: float = None,
        coding_gain: float = None,
        min_distance: float = None,
    ):
        """
        coding_rate (float, bps):
        coding_gain (float, W):
        """
        self._coding_rate = coding_rate
        self._coding_gain = coding_gain
        self._min_distance = min_distance

    @property
    def coding_rate(self) -> float:
        return self._coding_rate

    @property
    def coding_gain(self) -> float:
        if self._coding_gain is None:
            if self._min_distance is not None:
                self._coding_gain = self.code_rate * self.min_distance
        return self._coding_gain

    @staticmethod
    def coding_gain_eb_no(eb_no_coded: float, eb_no_uncoded: float) -> float:
        """
        calculate the difference in Eb/No required to produce the same error rate for coded and
        uncoded signals

        Parameters
        ----------
            eb_no_coded (float, W); the Eb/No of the coded signal
            eb_no_coded (float, W): the Eb/No of the uncoded signal

        Returns
        -------
          coding_gain (float, ):
        """
        return eb_no_uncoded / eb_no_coded

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Coding Rate",
                    "unit": "mbps",
```

```
                    "value": self.coding_rate,
                },
                {
                    "name": "Coding Gain",
                    "unit": "dB",
                    "value": watt_to_decibel(self.coding_gain),
                },
            ]
        )
        summary.set_index("name", inplace=True)
        return summary


class MPhaseShiftKeying(Modulation):
    def __init__(
        self,
        levels: int,
        bandwidth: float = None,
        carrier_signal: Waveform = None,
        modulating_signal: Waveform = None,
        energy_per_symbol: float = None,
        energy_per_bit: float = None,
        symbol_rate: float = None,
        symbol_period: float = None,
        bit_rate: float = None,
        bit_error_rate: float = None,
        bit_error_rate_coded: float = None,
        bit_period: float = None,
        bits_per_symbol: int = None,
        carrier_power: float = None,
        coding_rate: float = None,
        carrier_to_noise: float = None,
        carrier_to_noise_coded: float = None,
        spectral_efficiency: float = None,
        eb_no: float = None,
        eb_no_coded: float = None,
        es_no: float = None,
        es_no_coded: float = None,
        rolloff_rate: float = None,
        frequency_range: list = None,
        noise_probability: float = None,
        noise_probability_coded: float = None,
        noise_power_density: float = None,
        noise_power_density_coded: float = None,
        code: ConvolutionalCode = None,
        data_rate: float = None,
    ):
        """
        Parameters
        ----------
            levels (int,): number of levels in the waveform (equivalent to
    ↪ number of symbols)
            bandwidth (float, GHz): the range of frequencies required to
    ↪ transmit the signal
            energy_per_symbol (float, J): energy required to transmit one
```

```
↪ symbol
            energy_per_bit (float, J): energy required to transmit one bit
            symbol_rate (float, symbols/s): number of symbols transmitted
↪ per second
            symbol_period (float, s/symbol): seconds to transmit a symbol
            bit_rate (float, bits/s): number of bits transmitted per second
            bit_period (float, s/bit): number of seconds to transmit a bit
            carrier_power (float, W): total transmit power of signal
            spectral_efficiency (float bit/s/Hz):
        """
        self._levels = levels
        self._bandwidth = bandwidth
        self._energy_per_symbol = energy_per_symbol
        self._energy_per_bit = energy_per_bit
        self._symbol_rate = symbol_rate
        self._symbol_period = symbol_period
        self._bits_per_symbol = bits_per_symbol
        self._bit_rate = bit_rate
        self._bit_error_rate = bit_error_rate
        self._bit_period = bit_period
        self._carrier_power = carrier_power
        self._carrier_to_noise = carrier_to_noise
        self._eb_no = eb_no
        self._es_no = es_no
        self._rolloff_rate = rolloff_rate
        self._carrier_signal = carrier_signal
        self._modulating_signal = modulating_signal
        self._frequency_range = frequency_range
        self._spectral_efficiency = spectral_efficiency
        self._noise_probability = noise_probability
        self._code = code
        self._data_rate = data_rate
        self._bit_error_rate_coded = bit_error_rate_coded
        self._carrier_to_noise_coded = carrier_to_noise_coded
        self._eb_no_coded = eb_no_coded
        self._noise_power_density_coded = noise_power_density_coded
        self._noise_power_density = noise_power_density
        self._noise_probability_coded = noise_probability_coded
        self._es_no_coded = es_no_coded
        self.propagate_calculations()

    @property
    def es_no(self) -> float:
        if self._es_no is None:
            if self._carrier_to_noise is not None:
                self._es_no = (
                    self.carrier_to_noise * GHz_to_Hz(self.bandwidth) /
↪ self.symbol_rate
                )
            if self._eb_no is not None:
                return self.eb_no * self.bits_per_symbol
        return self._es_no

    @property
    def es_no_coded(self) -> float:
```

```
        if self._es_no_coded is None:
            if self._carrier_to_noise_coded is not None:
                self._es_no = (
                    self.carrier_to_noise_coded
                    * GHz_to_Hz(self.bandwidth)
                    / self.symbol_rate
                )
            elif self._eb_no_coded is not None:
                return self.eb_no_coded * self.bits_per_symbol
        return self._es_no_coded

    @property
    def eb_no(self) -> float:
        if self._eb_no is None:
            if self._bit_error_rate is not None:
                self._eb_no = (
                    erfcinv(self.bit_error_rate * self.bits_per_symbol)
                    / sin(pi / self.levels)
                ) ** 2 / self.bits_per_symbol
            elif self._es_no is not None:
                self._eb_no = self.es_no / self.bits_per_symbol
            elif (
                self._energy_per_bit is not None
                and self._noise_power_density is not None
            ):
                self._eb_no = self.energy_per_bit / self.noise_power_density
        return self._eb_no

    @property
    def eb_no_coded(self) -> float:
        if self._eb_no_coded is None:
            if self._code is not None:
                self._eb_no_coded = self.eb_no * self.code.coding_gain
            else:
                self._eb_no_coded = self.eb_no
        return self._eb_no_coded

    @eb_no.setter
    def eb_no(self, value) -> None:
        self._eb_no = value

    @property
    def carrier_power(self) -> float:
        return self._carrier_power

    @carrier_power.setter
    def carrier_power(self, value) -> None:
        self._carrier_power = value

    @property
    def bit_rate(self) -> float:
        if self._bit_rate is None:
            if self._bit_period is not None:
                self._bit_rate = 1.0 / self.bit_period
```

```
            elif self._rolloff_rate is not None:
                self._bit_rate = (
                    self.bits_per_symbol
                    * GHz_to_Hz(self.bandwidth)
                    / (1 + self.rolloff_rate)
                )
        return self._bit_rate

    @property
    def data_rate(self) -> float:
        if self._data_rate is None:
            if self._code is not None:
                self._data_rate = self.bit_rate * self.code.coding_rate
            else:
                self._data_rate = self.bit_rate
        return self._data_rate

    @property
    def bit_period(self) -> float:
        if self._bit_period is None:
            self._bit_period = self.symbol_period / self.bits_per_symbol
        return self._bit_period

    @property
    def symbol_rate(self) -> float:
        if self._symbol_rate is None:
            if self._bandwidth is not None and self._rolloff_rate is not
↪ None:
                self._symbol_rate = (GHz_to_Hz(self._bandwidth)) / (
                    1 + self.rolloff_rate
                )
            else:
                self._symbol_rate = self.bit_rate / self.bits_per_symbol
        return self._symbol_rate

    @property
    def symbol_period(self) -> float:
        if self._symbol_period is None:
            self._symbol_period = 1.0 / self.symbol_rate
        return self._symbol_period

    @property
    def spectral_efficiency(self) -> float:
        if self._spectral_efficiency is None:
            if self._bit_rate is not None and self._bandwidth is not None:
                self._spectral_efficiency = self.bit_rate / GHz_to_Hz(self.
↪ bandwidth)
        return self._spectral_efficiency

    @property
    def energy_per_symbol(self) -> float:
        if self._energy_per_symbol is None:
            if self._carrier_power is not None and self._symbol_period is
↪ not None:
                self._energy_per_symbol = self.carrier_power * self.
```

```
symbol_period
        return self._energy_per_symbol

    @property
    def bits_per_symbol(self):
        if self._bits_per_symbol is None:
            self._bits_per_symbol = log2(self.levels)
        return self._bits_per_symbol

    @property
    def energy_per_bit(self) -> float:
        """
        Returns
            energy_per_bit (float, J/s)
        """
        if self._energy_per_bit is None:
            if self._carrier_power is not None:
                self._energy_per_bit = self.carrier_power * self.bit_period
        return self._energy_per_bit

    @property
    def noise_power_density(self) -> float:
        if self._noise_power_density is None:
            if (
                self._symbol_rate is not None
                and self._carrier_to_noise is not None
                and self._energy_per_symbol is not None
            ):
                self._noise_power_density = self.symbol_rate / (
                    self.bandwidth * self.carrier_to_noise * self.energy_per_symbol
                )
        return self._noise_power_density

    @property
    def noise_power_density_coded(self) -> float:
        if self._noise_power_density_coded is None:
            if (
                self._symbol_rate is not None
                and self._carrier_to_noise_coded is not None
                and self._energy_per_symbol is not None
            ):
                self._noise_power_density_coded = self.symbol_rate / (
                    self.bandwidth
                    * self.carrier_to_noise_coded
                    * self.energy_per_symbol
                )
        return self._noise_power_density_coded

    @property
    def noise_probability(self) -> float:
        if self._noise_probability is None:
            if self._es_no is not None:
                self._noise_probability = erfc(sqrt(self.es_no) * sin(pi / self.levels))
```

```
        return self._noise_probability

    @property
    def noise_probability_coded(self) -> float:
        if self._noise_probability_coded is None:
            if self._eb_no_coded is not None:
                self._noise_probability_coded = erfc(
                    sqrt(self.bits_per_symbol * self.eb_no_coded)
                    * sin(pi / self.levels)
                )
        return self._noise_probability_coded

    @property
    def bit_error_rate(self) -> float:
        if self._bit_error_rate is None:
            if self._noise_probability is not None:
                self._bit_error_rate = self.noise_probability / self.bits_per_symbol
        return self._bit_error_rate

    @property
    def bit_error_rate_coded(self) -> float:
        if self._bit_error_rate_coded is None:
            if self._noise_probability_coded is not None:
                self._bit_error_rate_coded = (
                    self.noise_probability_coded / self.bits_per_symbol
                )
        return self._bit_error_rate_coded

    @property
    def levels(self) -> float:
        return self._levels

    @property
    def rolloff_rate(self) -> float:
        return self._rolloff_rate

    @property
    def carrier_to_noise(self) -> float:
        if self._carrier_to_noise is None:
            if self._eb_no is not None:
                self._carrier_to_noise = (
                    self.eb_no * self.bit_rate / GHz_to_Hz(self._bandwidth)
                )
        return self._carrier_to_noise

    @property
    def carrier_to_noise_coded(self) -> float:
        if self._carrier_to_noise_coded is None:
            if self._eb_no_coded is not None:
                self._carrier_to_noise_coded = (
                    self.eb_no_coded * self.bit_rate / GHz_to_Hz(self._bandwidth)
                )
        return self._carrier_to_noise_coded
```

```

    @property
    def carrier_signal(self) -> Waveform:
        return self._carrier_signal

    @property
    def code(self) -> ConvolutionalCode:
        return self._code

    @property
    def bandwidth(self) -> float:
        if self._bandwidth is None:
            if self._symbol_rate is not None and self._rolloff_rate is not
↪ None:
                self._bandwidth = self.symbol_rate * (1 + self.rolloff_rate
↪ )
        return self._bandwidth

    @property
    def frequency_range(self) -> float:
        if self._frequency_range is None:
            self._frequency_range = [
                self.carrier_signal.frequency - self.bandwidth / 2,
                self.carrier_signal.frequency + self.bandwidth / 2,
            ]
        return self._frequency_range

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {
                    "name": "Maximum Bit Rate",
                    "unit": "mbps",
                    "value": bit_to_mbit(self.bit_rate),
                },
                {
                    "name": "Data Rate",
                    "unit": "mbps",
                    "value": bit_to_mbit(self.data_rate),
                },
                {"name": "Bandwidth", "unit": "GHz", "value": self.
↪ bandwidth},
                {
                    "name": "Spectral Efficiency",
                    "unit": "bits/s/Hz",
                    "value": self.spectral_efficiency,
                },
                {
                    "name": "C/N Ratio",
                    "unit": "bits/s/GHz",
                    "value": watt_to_decibel(self.carrier_to_noise),
                },
                {
                    "name": "Coded C/N Ratio",
                    "unit": "bits/s/GHz",
```

```
                    "value": watt_to_decibel(self.carrier_to_noise_coded),
                },
                {
                    "name": "Eb/No Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.eb_no),
                },
                {
                    "name": "Coded Eb/No Ratio",
                    "unit": "dB",
                    "value": watt_to_decibel(self.eb_no_coded),
                },
                {
                    "name": "Bit Error Rate",
                    "unit": "",
                    "value": self.bit_error_rate,
                },
                {
                    "name": "Coded Bit Error Rate",
                    "unit": "",
                    "value": self.bit_error_rate_coded,
                },
                {"name": "Roll-Off Factor", "unit": "", "value": self.
    ↪ rolloff_rate},
            ]
        )
        summary.set_index("name", inplace=True)

        if self.code is not None:
            code = self.code.summary()
            summary = pd.concat([summary, code])
        else:
            summary = summary[[not ("Coded" in idx) for idx in summary.
    ↪ index]]

        return summary

    def propagate_calculations(self) -> float:
        for _ in range(4):
            for var in type(self).__dict__:
                getattr(self, var, None)


class BinaryPhaseShiftKeying(MPhaseShiftKeying):
    def __init__(
        self,
        bandwidth: float = None,
        carrier_signal: Waveform = None,
        modulating_signal: Waveform = None,
        energy_per_symbol: float = None,
        energy_per_bit: float = None,
        symbol_rate: float = None,
        symbol_period: float = None,
        bit_rate: float = None,
        bit_period: float = None,
```

```
        bits_per_symbol: float = None,
        carrier_power: float = None,
        rolloff_rate: float = None,
        frequency_range: list = None,
    ):
        super().__init__(
            levels=2,
            bandwidth=bandwidth,
            carrier_signal=carrier_signal,
            modulating_signal=modulating_signal,
            energy_per_bit=energy_per_bit,
            rolloff_rate=rolloff_rate,
            frequency_range=frequency_range,
            carrier_power=carrier_power,
            energy_per_symbol=energy_per_symbol,
            symbol_rate=symbol_rate,
            bit_rate=bit_rate,
            bit_period=bit_period,
            bits_per_symbol=bits_per_symbol,
        )

    @property
    def noise_probability(self) -> float:
        if self._noise_probability is None:
            if self._carrier_to_noise is not None:
                self._noise_probability = 0.5 * erfc(sqrt(self.
    ↪ carrier_to_noise))
        return self._noise_probability


class QuadraturePhaseShiftKeying(MPhaseShiftKeying):
    def __init__(
        self,
        bandwidth: float = None,
        carrier_signal: Waveform = None,
        modulating_signal: Waveform = None,
        energy_per_symbol: float = None,
        symbol_rate: float = None,
        symbol_period: float = None,
        carrier_power: float = None,
        bit_rate: float = None,
        bit_period: float = None,
        bits_per_symbol: float = None,
        rolloff_rate: float = None,
    ):
        super().__init__(
            levels=4,
            bandwidth=bandwidth,
            carrier_signal=carrier_signal,
            modulating_signal=modulating_signal,
            energy_per_symbol=energy_per_symbol,
            symbol_rate=symbol_rate,
            carrier_power=carrier_power,
            bit_rate=bit_rate,
            bit_period=bit_period,
```

```
            bits_per_symbol=bits_per_symbol,
            rolloff_rate=rolloff_rate,
        )

    @property
    def noise_probability(self) -> float:
        if self._noise_probability is None:
            if self._carrier_to_noise is not None:
                self._noise_probability = 0.5 * erfc(sqrt(self.
    ↪ carrier_to_noise * 0.5))
        return self._noise_probability
```

Listing 5: signal$_{p}$*rocessing/modulation.py*

```
from math import atan2, cos, degrees, radians

import numpy as np

from link_calculator.components.antennas import Antenna
from link_calculator.constants import EARTH_RADIUS


def slant_path(
    elevation_angle: float,
    rain_altitude: float,
    station_altitude: float,
) -> float:
    """
    Calculate the slant path

    Parameters
    ----------
        angle_of_elevation (float, deg): the angle between the Earth
    ↪ station and the satellite
        rain_height (float, km): the rain height
        station_altitude (float, km): the rain height of the Earth station
    ↪ above sea level
        refraction_radius (float, km): The modified radius of the Earth to
    ↪ account for the
            refraction of the wave by thr troposphere

    Returns
    -------
        d_s (float, km): The slant height
    """
    refraction_radius = 8500 if station_altitude < 1.0 else EARTH_RADIUS
    elevation_angle_rad = radians(elevation_angle)
    if elevation_angle < 5:
        return (
            2
            * (rain_altitude - station_altitude)
            / np.sqrt(
                np.sin(elevation_angle_rad) ** 2
                + 2 * (rain_altitude - station_altitude) /
    ↪ refraction_radius
```

```python
            )
        )
    else:
        return (rain_altitude - station_altitude) / np.sin(elevation_angle_rad)


def rain_specific_attenuation(frequency: float, rain_rate: float, polarization: str):
    """
    TODO()

    Parameters
    ----------

    Returns
    -------

    """
    _f = [
        1,
        2,
        4,
        6,
        7,
        8,
        10,
        12,
        15,
        20,
        25,
        30,
        35,
        40,
        45,
        50,
        60,
        70,
        80,
        90,
        100,
        120,
        150,
        200,
        300,
        400,
    ]

    _kH = [
        0.0000387,
        0.000154,
        0.00065,
        0.00175,
        0.00301,
        0.00454,
```

```
        0.0101,
        0.0188,
        0.0367,
        0.0751,
        0.124,
        0.187,
        0.263,
        0.35,
        0.442,
        0.536,
        0.707,
        0.851,
        0.975,
        1.06,
        1.12,
        1.18,
        1.31,
        1.45,
        1.36,
        1.32,
    ]

    _kV = [
        0.0000352,
        0.000138,
        0.000591,
        0.00155,
        0.00265,
        0.00395,
        0.00887,
        0.0168,
        0.0335,
        0.0691,
        0.113,
        0.167,
        0.233,
        0.31,
        0.393,
        0.479,
        0.642,
        0.784,
        0.906,
        0.999,
        1.06,
        1.13,
        1.27,
        1.42,
        1.35,
        1.31,
    ]

    _alphaH = [
        0.912,
        0.963,
        1.121,
```

```
        1.308,
        1.332,
        1.327,
        1.276,
        1.217,
        1.154,
        1.099,
        1.061,
        1.021,
        0.979,
        0.939,
        0.903,
        0.873,
        0.826,
        0.793,
        0.769,
        0.753,
        0.743,
        0.731,
        0.71,
        0.689,
        0.688,
        0.683,
    ]

    _alphaV = [
        0.88,
        0.923,
        1.075,
        1.265,
        1.312,
        1.31,
        1.264,
        1.2,
        1.128,
        1.065,
        1.03,
        1,
        0.963,
        0.929,
        0.897,
        0.868,
        0.824,
        0.793,
        0.769,
        0.754,
        0.744,
        0.732,
        0.711,
        0.69,
        0.689,
        0.684,
    ]

    KH = np.exp(np.interp(np.log(frequency), np.log(_f), np.log(_kH)))
```

```
    KV = np.exp(np.interp(np.log(frequency), np.log(_f), np.log(_kV)))

    alphaH = np.interp(np.log(frequency), np.log(_f), _alphaH)
    alphaV = np.interp(np.log(frequency), np.log(_f), _alphaV)

    if polarization == "circular":
        k = (KH + KV) / 2
        alpha = (KH * alphaH + KV * alphaV) / (2 * k)
    elif polarization == "vertical":
        k = KV
        alpha = alphaV
    elif polarization == "horizontal":
        k = KH
        alpha = alphaH
    else:
        raise Exception("Invalid Polarization")

    return k, alpha, k * rain_rate**alpha


def horizontal_reduction(
    horizontal_projection: float, specific_attenuation: float, frequency:
    ↪ float
) -> float:
    """
    TODO()

    Parameters
    ----------

    Returns
    -------
    """
    return 1 / (
        1
        + 0.78 * np.sqrt(horizontal_projection * specific_attenuation /
    ↪ frequency)
        - 0.38 * (1 - np.exp(-2 * horizontal_projection))
    )


def vertical_adjustment(
    elevation_angle: float,
    specific_attenuation: float,
    d_r: float,
    frequency: float,
    chi: float,
) -> float:
    """
    Calculate the vertical adjustment factor

    Parameters
    ----------
        elevation_angle (float, deg):
        specific_attenation (float, dBKm-1)
```

```
        d_r (float, km):
        frequency (float, GHz):
        chi (float, deg):

    Returns
    -------
        vert_adj (float, )
    """
    return 1 / (
        1
        + np.sqrt(np.sin(radians(elevation_angle)))
        * (
            31
            * (1 - np.exp(-elevation_angle / (1 + chi)))
            * (np.sqrt(d_r * specific_attenuation) / (frequency**2))
            - 0.45
        )
    )


def zeta(
    rain_altitude: float,
    station_altitude: float,
    horizontal_projection: float,
    horizontal_reduction: float,
) -> float:
    """
    Calculate interim vertical adjustment value

    Parameters
    ----------

    Returns
    -------
    """
    return degrees(
        atan2(
            rain_altitude - station_altitude,
            horizontal_projection * horizontal_reduction,
        )
    )


def rain_attenuation(
    elevation_angle: float,
    slant_path: float,
    frequency: float,
    rain_altitude: float,
    station_altitude: float,
    station_latitude: float,
    rain_rate: float = 0.01,
    polarization: str = "vertical",
) -> float:
    """

```

```
    Parameters
    ----------

    Returns
    -------

    """
    elevation_angle_rad = radians(elevation_angle)
    horiz_proj = slant_path * np.cos(elevation_angle_rad)

    _, _, specific_att = rain_specific_attenuation(frequency, rain_rate, polarization)

    horiz_reduction = horizontal_reduction(horiz_proj, specific_att, frequency)

    zeta_ = zeta(rain_altitude, station_altitude, horiz_proj, horiz_reduction)

    if zeta_ > elevation_angle:
        d_r = horiz_proj * horiz_reduction / np.cos(elevation_angle_rad)
    else:
        d_r = slant_path

    if abs(station_latitude) < 36:
        chi = 36 - abs(station_latitude)
    else:
        chi = 0

    vert_adj = vertical_adjustment(elevation_angle, specific_att, d_r, frequency, chi)
    effective_path = slant_path * vert_adj

    return specific_att * effective_path


def worst_rain_rate(rain_rate: float) -> float:
    return (rain_rate / 0.3) ** 0.87


def polarization_loss(faraday_rotation: float) -> float:
    rotation_rad = radians(faraday_rotation)
    return 20 * np.log(cos(rotation_rad))
```

Listing 6: propagation/utils.py

```
from math import acos, atan, cos, degrees, pi, radians, sin, sqrt, tan

import pandas as pd

from link_calculator.constants import EARTH_MU, EARTH_RADIUS


class Orbit:
    def __init__(
```

```
        self,
        semi_major_axis: float = None,
        eccentricity: float = None,
        inclination: float = float,
        raan: float = None,
        arg_of_perigee: float = None,
        true_anomaly: float = None,
        period: float = None,
        orbital_radius: float = None,
    ):
        """

        Parameters
        ---------
            semi_major_axis (float, km): The semi-major axis of the orbit
        """
        self._semi_major_axis = semi_major_axis
        self._eccentricity = eccentricity
        self._inclination = inclination
        self._raan = raan
        self._arg_of_perigee = arg_of_perigee
        self._true_anomaly = true_anomaly
        self._period = period
        self._orbital_radius = orbital_radius

    def period(self, mu: float = EARTH_MU) -> float:
        """
        Calculate the period of the satellite's orbit according to Kepler's third law

            mu (float, km^3/s^-2, optional): Kepler's gravitational constant

        Returns
        ------
            period (float, s): the time taken for the satellite to complete a revolution
        """
        if self._period is None:
            self._period = 2 * pi * sqrt(self.semi_major_axis**3 / mu)
        return self._period

    @property
    def semi_major_axis(self) -> float:
        return self._semi_major_axis

    @property
    def orbital_radius(self) -> float:
        if self._orbital_radius is None:
            self._orbital_radius = (
                self.semi_major_axis
                * (1 - self.eccentricity**2)
                / (1 + self.eccentricity * cos(radians(self.true_anomaly)))
            )
        return self._orbital_radius
```

```python


class GeodeticCoordinate:
    def __init__(self, latitude: float, longitude: float, altitude: float = 0):
        self._latitude = latitude
        self._longitude = longitude
        self._altitude = altitude

    @property
    def latitude(self) -> float:
        return self._latitude

    @property
    def longitude(self) -> float:
        return self._longitude

    @property
    def altitude(self) -> float:
        return self._altitude

    def central_angle(
        self,
        point: "GeodeticCoordinate",
    ) -> float:
        """
        Calculate angle gamma at the centre of the ground, between the Earth station and the satellite

        Parameters
        ----------
            ground_station_lat (float, deg): the latitude of the ground station
            ground_station_long (float, deg): the longitude of the ground station
            sat_lat (float, deg): the latitude of the satellite
            sat_long (float, deg): the longitude of the satellite

        Returns
        ------
            gamma (float, rad): angle between satellite and ground station
        """
        gamma = acos(
            cos(radians(self.latitude))
            * cos(radians(point.latitude))
            * cos(radians(point.longitude) - radians(self.longitude))
            + sin(radians(self.latitude)) * sin(radians(point.latitude))
        )
        return degrees(gamma)

    def summary(self) -> pd.DataFrame:
        summary = pd.DataFrame.from_records(
            [
                {"name": "Latitude", "unit": "  ", "value": self.latitude},
                {"name": "Longitude", "unit": "  ", "value": self.longitude
```

```python
},
                {"name": "Altitude", "unit": "  ", "value": self.altitude},
            ]
        )
        summary.set_index("name", inplace=True)
        return summary


def central_angle_orbital_radius(
    orbital_radius: float, planet_radius: float = EARTH_RADIUS, elevation: float = 0
):
    """
    Calculate angle gamma at the centre of the ground, between the Earth
    station and the satellite, given the orbital radius of the satellite

    Parameters
    ---------
        orbital_radius (float, km): distance from the centre of mass to the satellite
        planet_radius (float, km, optional): radius of the planet
        elevation (float, deg): the angle of elevation over the horizon

    Returns
    ------
        gamma (float, deg): angle between satellite and ground station
    """
    elevation_rad = radians(elevation)
    gamma = acos((planet_radius * cos(elevation_rad)) / orbital_radius) - elevation_rad
    return degrees(gamma)


def slant_range(
    orbital_radius: float, central_angle: float, planet_radius: float = EARTH_RADIUS
) -> float:
    """
    Calculate the slant range from the ground station to the satellite

    Parameters
    ----------
        orbital_radius (float, km): distance from the centre of mass to the satellite
        gamma (float, deg): central_angle; angle from the satellite to the ground station, centred at the centre of mass
        planet_radius (float, km, optional): radius of the planet

    Return
    ------
        slant_range (float, km): the distance from the ground station to the satellite

    """
    return sqrt(
```

```
        planet_radius**2
        + orbital_radius**2
        - 2 * planet_radius * orbital_radius * cos(radians(central_angle))
    )


def elevation_angle(
    orbital_radius: float, central_angle: float, planet_radius: float = EARTH_RADIUS
) -> float:
    """
    Calculate the elevation angle from the ground station to the satellite

    Parameters
    ----------
        orbital_radius (float, km): distance from the centre of mass to the satellite
        central_angle (float, rad): angle from the satellite to the ground station, centred at the centre of mass
        planet_radius (float, km, optional): radius of the planet

    Return
    ------
        elevation_angle (float, rad): the distance from the ground station to the satellite

    """
    gamma_rad = radians(central_angle)
    elev = acos(
        sin(gamma_rad)
        / sqrt(
            1
            + (planet_radius / orbital_radius) ** 2
            - 2 * (planet_radius / orbital_radius) * cos(gamma_rad)
        )
    )
    return degrees(elev)


def area_of_coverage(central_angle: float, planet_radius: float = EARTH_RADIUS):
    """
    Calculate the surface area coverage of the Earth from a satellite

    Parameters
    ----------
        central_angle (float, deg): angle from the satellite to the ground station, centred at the centre of mass
        planet_radius (float, km, optional): radius of the planet

    Return
    ------
        area_coverage (float, km): the area of the Earth's surface visible from a satellite

```

```
    """
    return 2 * pi * (planet_radius**2) * (1 - cos(radians(central_angle)))


def percentage_of_coverage(
    central_angle: float, planet_radius: float = EARTH_RADIUS
) -> float:
    """
    Calculate the surface area coverage of the Earth from a satellite as a percentage of the total
        surface area

    Parameters
    ----------
        central_angle (float, rad): angle from the satellite to the ground station, centred at the centre of mass
        planet_radius (float, km, optional): radius of the planet

    Return
    ------
        area_coverage (float, %): the percentage of the Earth's surface visible from a satellite

    """
    return (
        area_of_coverage(central_angle, planet_radius)
        / (4 * pi * planet_radius**2)
        * 100
    )


def percentage_of_coverage_gamma(central_angle: float) -> float:
    """
    Calculate the surface area coverage of the Earth from a satellite as a percentage of the total
        surface area

    Parameters
    ----------
        central_angle (float, rad): angle from the satellite to the ground station, centred at the centre of mass

    Return
    ------
        area_coverage (float, %): the percentage of the Earth's surface visible from a satellite

    """
    gamma_rad = radians(central_angle)
    return 50 * (1 - cos(gamma_rad))


def azimuth_intermediate(
    ground_station_lat: float, ground_station_long: float, sat_long: float
) -> float:
```

```
    """
    Calculate the azimuth of a geostationary satellite

    Parameters
    ----------
        ground_station_lat (float, deg): the latitude of the ground station
        ground_station_long (float, deg): the longitude of the ground station
        sat_long (float, deg): the longitude of the satellite

    Returns
    ------
        azimuth (float, rad): horizontal pointing angle of the ground station antenna to
            the satellite. The azimuth angle is usually measured in clockwise direction
            in degrees from true north.
    """
    gs_lat_rad = radians(ground_station_lat)
    gs_long_rad = radians(ground_station_long)
    sat_long_rad = radians(sat_long)
    az = atan(tan(abs(sat_long_rad - gs_long_rad)) / sin(gs_lat_rad))
    return degrees(az)


def max_visible_distance(
    orbital_radius: float,
    ground_station_lat: float,
    min_angle: float = 0.0,
    planet_radius: float = EARTH_RADIUS,
) -> float:
    """
    Calculate the radius of visibility for a satellite and a ground station

    Parameters
    ----------
        orbital_radius (float, km): the radius of the satellite from the centre of the Earth
        ground_station_lat (float, deg): the latitude of the ground station
        min_angle (float, deg): the minimum angle of visibility over the horizon
        planet_radius (float, km, optional): radius of the planet

    Returns
    ------

    """
    gs_lat_rad = radians(ground_station_lat)
    min_angle_rad = radians(min_angle)
    return acos(
        cos(
            (
                acos((planet_radius / orbital_radius) * cos(min_angle_rad))
                - min_angle_rad
            )
```

```python
                / cos(gs_lat_rad)
            )
        )


def azimuth(
        ground_station_lat: float,
        ground_station_long: float,
        sat_lat: float,
        sat_long: float,
) -> float:
        """
        Calculate the azimuth of a satellite

        Parameters
        ----------
            ground_station_lat (float, deg): the latitude of the ground station
            ground_station_long (float, deg): the longitude of the ground station
            sat_lat (float, deg): the latitude of the satellite
            sat_long (float, deg): the longitude of the satellite

        Returns
        ------
            azimuth (float, rad): horizontal pointing angle of the ground station antenna to
                the satellite. The azimuth angle is usually measured in clockwise direction
                in degrees from true north.
        """
        # Ground station is in northern hemisphere
        # rel_pos = ground_station_long - sat_long
        #
        # if 90 > gs_long_rad > 0:
        #
        # At the equator
        # In the southern hemisphere
        # elif 0 > gs_long_rad >= -`90:
        #    if (sat_lat - ground_station_lat)
        #        return pi / 2
        #    else:
        # raise ValueError("Invalid value for the ground station's longitude: must be in range -90 < L < 90")
```

Listing 7: orbits/utils.py

```python
# Orbits
EARTH_GRAVITATIONAL_CONSTANT = 6.6743e-11  # Nm^2 / kg^2
EARTH_MASS = 5.98e24  # km^3/s^2
EARTH_MU = 3.986004418e5  # km^3/s^-2
EARTH_RADIUS = 6378.14  # km
EARTH_POLAR_RADIUS = 6357  # km
EARTH_SOLAR_YEAR = 365.25  # days
SIDEREAL_DAY = 23.935  # hours
SIDEREAL_DAY_S = SIDEREAL_DAY * 60 * 60
```

```

# Propagation
SPEED_OF_LIGHT = 299792458  # m/s
BOLTZMANN_CONSTANT = 1.38e-23  # JK-1
```

Listing 8: constants.py

```
import numpy as np

from link_calculator.constants import SPEED_OF_LIGHT


def decibel_to_watt(value: float):
    """
    Convert decibel watt units to watts

    Parameters
    ----------
      decibels (float, dBW): decibel value

    Returns
    -------
      watts (float, W): Watt value
    """
    if value is None:
        return None
    return 10 ** (value / 10)


def watt_to_decibel(value: float):
    """
    Convert watts to decibel watts

    Parameters
    ----------
      watts (float, W): Watt value

    Returns
    -------
      decibels (float, dBW): decibel value
    """
    if value is None:
        return None
    return 10 * np.log10(value)


def frequency_to_wavelength(value: float) -> float:
    """
    Convert frequency to wavelength

    Parameters
    ----------
      frequency (float, GHz): the frequency of the wave

    Returns
```

```
    -------
      wavelength (float, m): length of the wave between peaks
    """
    if value is None:
        return None
    return SPEED_OF_LIGHT / (value * 1e9)


def wavelength_to_frequency(value: float) -> float:
    """
    Convert frequency to wavelength

    Parameters
    ----------
      wavelength (float, m): length of the wave between peaks

    Returns
    -------
      frequency (float, GHz): the frequency of the wave
    """
    if value is None:
        return None
    return (SPEED_OF_LIGHT / value) * 1e-9
```

Listing 9: propagation/conversions.py

```
def MHz_to_GHz(value: float) -> float:
    if value is None:
        return None
    return value * 1e-3


def GHz_to_MHz(value: float) -> float:
    if value is None:
        return None
    return value * 1e3


def GHz_to_Hz(value: float) -> float:
    if value is None:
        return None
    return value * 1e9


def Hz_to_GHz(value: float) -> float:
    if value is None:
        return None
    return value * 1e-9


def MHz_to_Hz(value: float) -> float:
    if value is None:
        return None
    return value * 1e6

```

```python

def Hz_to_MHz(value: float) -> float:
    if value is None:
        return None
    return value * 1e-6


def mbit_to_bit(value: float) -> float:
    if value is None:
        return None
    return value * 1e6


def bit_to_mbit(value: float) -> float:
    if value is None:
        return None
    return value * 1e-6
```

Listing 10: signal$_{p}$*rocessing/conversions.py*

```python
from math import log10


def joules_to_decibel_joules(value: float) -> float:
    if value is None:
        return None
    return 10 * log10(value)


def decibel_joules_to_joules(value: float) -> float:
    if value is None:
        return None
    return 10 ** (value / 10)
```

Listing 11: components/conversions.py

```python
import numpy as np


def axes_to_eccentricity(a: float, b: float):
    return np.sqrt(a**2, b**2) / a
```

Listing 12: orbits/conversions.py