Link_Budget_Modeling

March 15, 2025

1 An Earth-Venus-Earth Link Budget from Open Research Institute

This document contains a detailed link budget analysis for Earth-Venus-Earth (EVE) communications. It begins with a Python dataclass for each fixed earth station. Dataclasses are a type of Python object where only variables are declared. No methods are included. The site-specific dataclasses have parameters that are true for the site regardless of the target. Following the site-specific dataclasses are link budget classes, which contain target-specific values and methods that return various gains and losses. The most important output of a link budget object is a carrier to noise ratio at a particular receive bandwidth. A link budget class inherits a particular site dataclass. The site-specific dataclasses and the link budgets can be mixed and matched. This gives flexibility, as a link budget for a particular target, such as EVE, can be calculated for different sites by having that link budget inherit different site-specific dataclasses.

1.0.1 to do from 4 March 2025 meetup:

- ☒ Michelle to update the link budget with 5 log10 for non-coherent averaging instead of 10 log10.
- ☒ Michelle to document the change to 5 log10 for non-coherent averaging in the link budget, explaining the practical reasons for this derating.
- ☒ Michelle to reach out to retired scientists with EME stations to help validate the link budget model using moon bounce data.
- ☐ Thomas to analyze the 30-second Zadoff-Chu transmission data to evaluate incoherent averaging performance in practice.
- ☐ Team to work on validating link budget assumptions by comparing model predictions for moon bounces against real EME data.
- ☒ Team to review and refine the Doppler spread model in the link budget.
- ☒ Team to consider lowering the target SNR requirement (Zadoff-Chu paper) to reduce required transmission duration.
- ☒ Team to continue refining and improving the EVE link budget and transmission proposal.

1.0.2 to do from 11 March 2025 meetup:

- ☒ Michelle to consolidate imports
- ☒ Michelle to move Tsys up and include calculation instead of hand edit in link budget
- ☒ Michelle to do a worksheet for Temporal Spread
- ☐ Michelle to consolidate pointing and tracking error worksheet
- ☐ Michelle to add Dwingeloo and other potential sites to the dataclass set in the link budget.

1.1 EVE compared to EME

The communications mode most similar to EVE is Earth-Moon-Earth (EME). EVE is more complex than EME communications due to several factors:

1. Much greater distances involved
2. Greater variability in the distances involved
3. Doppler rate of change due orbital positions of Earth and Venus changing with respect to each other
4. Different signal reflection characteristics from Venus compared to the Moon
5. Doppler spreading

1.2 Contributors

Including but not limited to Michelle Thompson, Matthew Wishek, Paul Williamson, Rose Easton, Thomas Telkamp, Pete Wyckoff, Gary K6MG, and Lee Blanton. Questions or comments about this document? Please file an issue in the repository or write hello at operesearch dot institute.

1.3 Function to Display SVG

[61]:
```
from IPython.display import display, SVG, Image, HTML
import os

def display_svg(svg_file, width=None):
    """Display SVG with proper width control and fallback options"""
    # Check if file exists
    if not os.path.exists(svg_file):
        print(f"Error: SVG file not found: {svg_file}")
        return

    try:
        # For SVG, we need to use HTML to control the width
        if width:
            # Use HTML with img tag to control width
            html = f'<img src="{svg_file}" width="{width}px">'
            return HTML(html)
        else:
            # Use standard SVG display
            return SVG(filename=svg_file)
    except Exception as e:
        print(f"Warning: Could not display {svg_file} as SVG. Error: {e}")

        # Check if PNG version exists
        png_file = svg_file.replace('.svg', '.png')
        if os.path.exists(png_file):
            if width:
                html = f'<img src="{png_file}" width="{width}px">'
                return HTML(html)
            else:
```

```
            return Image(filename=png_file)
        else:
            print(f"No PNG fallback found at {png_file}")

            # Let's help create a PNG version if inkscape is available
            try:
                import subprocess
                print(f"Attempting to convert SVG to PNG using inkscape...")
                subprocess.run(['inkscape', '-z', '-e', png_file, svg_file],
                               check=True, capture_output=True)
                if os.path.exists(png_file):
                    print(f"Successfully created PNG: {png_file}")
                    return Image(filename=png_file)
                else:
                    print("Conversion didn't produce a PNG file")
            except Exception as e2:
                print(f"Could not convert SVG to PNG: {e2}")

            # If all else fails, just embed the raw SVG content
            try:
                with open(svg_file, 'r') as f:
                    svg_content = f.read()
                    return HTML(svg_content)
            except:
                print("All display methods failed.")
```

2 ORI Earth-Venus-Earth Communications Development Roadmap

Note: the checkboxes for each task listed below *may not* automatically render if you are viewing this document on GitHub. If you want to see completed vs. uncompleted tasks, then you will need to see the PDF rendering of this document, or render it to a viewer yourself.

2.1 Phase 1: Foundation and Research (3-4 months)

2.1.1 Documentation and Requirements

- ☒ Document full link budget analysis (this notebook)
- ☒ Compile SDR Capabilities (zc706/adrv9009 or any partner SDR system)
- ☒ Create detailed Doppler analysis (in this notebook)
- ☒ Define custom mode specifications (CNR in 1Hz now known)
- ☐ Write FPGA requirements document

zc706/adrv9009 FPGA Development Board Capabilities ADRV9009 capabilities:

-Frequency range: 75 MHz to 6 GHz (covers our 1296 MHz) -Observation bandwidth: up to 200 MHz (plenty for our ~600 kHz Doppler range) -12-bit ADC -Low noise performance -Direct sampling capability

The Zynq 7 FPGA on the zc706 provides:

-DSP slices: 900 -Block RAM: 555 × 36 Kb -LUTs: 218,600 -Flip-flops: 437,200

zcu102/adrv9009 FPGA Development Board Capabilities ADRV9009 capabilities:

-Frequency range: 75 MHz to 6 GHz (covers our 1296 MHz) -Observation bandwidth: up to 200 MHz (plenty for our ~600 kHz Doppler range) -12-bit ADC -Low noise performance -Direct sampling capability

UltraScale+ XCZU9EG on the zcu102 provides:

DSP Slices: 2,520 (vs 900 on zc706) Block RAM: 1,824 × 36Kb (vs 555 on zc706) UltraRAM: 960 Kb LUTs: 548,160 (vs 218,600 on zc706) Flip-flops: 1,096,320 (vs 437,200 on zc706)

-More parallel processing channels possible -Better timing closure likely with UltraScale+ architecture -UltraRAM provides additional buffer space for sample processing -Higher achievable clock rates -More room for future expansion/features -Could implement more parallel demodulators for Doppler tracking

USRP X310

Detailed Doppler Analysis Document

Custom Mode Definition In the case that no existing communications mode closes the link (see Evaluation of Modes cell below) we could design a custom "EVE-Mode" with:

- QPSK modulation as a baseline (good balance of spectral efficiency and robustness)
- ~5 kHz bandwidth (more than wide enough for short-term Doppler)
- Strong FEC (LDPC, Polar)
- Synchronization sequence designed for high Doppler rates
- Multiple parallel decoding paths

Custom Mode SDR Requirements

- Need high sample rate (>1.2 MSPS)
- Need Excellent phase noise performance
- 16-bit ADC minimum
- Very stable frequency reference

Custom Mode FPGA Specficiation

2.1.2 Initial GNU Radio Development

- ☐ Basic flow graph implementation
- ☐ Simple Doppler tracking prototype
- ☐ Test with recorded or simulated signals/simulations
- ☐ Document processing chain

2.1.3 Community Engagement

- ☒ Present project on ORI channels
- ☒ Recruit additional developers (mailing list, Slack)
- ☒ Set up project definition and documentation (this document)

2.2 Phase 2: Core Development (6-8 months)

2.2.1 GNU Radio Implementation

- ☐ Complete mode implementation
- ☐ Doppler tracking refinement
- ☐ orbital prediction integration
- ☐ Performance monitoring tools
- ☐ User interface development

2.2.2 FPGA Development

- ☐ DDC implementation?
- ☐ Initial Doppler tracking
- ☐ Memory interface design
- ☐ Basic demodulator implementation
- ☐ Testing framework development
- ☐ Does our AI/ML team have a role?

2.2.3 Integration Planning

- ☐ SDR selection and acquisition: zc706/adrv9009, zcu102/adrv9009 or 9002, new hardware
- ☐ Interface definition for hardware
- ☒ Test equipment requirements (Remote Labs confirmed)
- ☐ Performance measurement plans

2.3 Phase 3: Integration and Testing (4-6 months)

2.3.1 Hardware Integration

- ☐ SDR integration (do the work)
- ☐ FPGA bitstream testing (do the work)
- ☐ Timing verification (do the work)
- ☐ Performance measurements (yes, do the work)

2.3.2 Software Integration

- ☐ GNU Radio to FPGA interface?
- ☐ Control software development
- ☐ Monitoring tools - [ ] probably not required due to brevity of opportunities
- ☐ User interface refinement (standing orders)

2.3.3 Testing Framework

- ☐ Automated test development (probably not needed but need to talk about)

- ☐ Performance verification (did it work or not)
- ☐ Doppler simulation testing (did our model turn out to be accurate)
- ☐ Link budget verification (did we close the link with expected margins)

2.4 Phase 4: Optimization and Documentation (3-4 months)

2.4.1 Performance Optimization (Vivado)

- ☐ FPGA resource optimization and utilization reports
- ☐ Processing chain refinement
- ☐ Doppler tracking improvements?
- ☐ Memory usage optimization

2.4.2 Documentation

- ☐ User manual creation (HTML5 interface?)
- ☐ Installation guides
- ☐ Development documentation, articles, reports
- ☐ Performance reports, papers

2.4.3 Community Resources

- ☐ Example configurations
- ☐ Tutorial development
- ☐ Test data publication
- ☐ Contribution guidelines

2.5 Phase 5: Deployment and Validation (3-4 months)

2.5.1 Field Testing

- ☐ Initial station setup (may be very narrow cases if remote access not available)
- ☐ Performance validation
- ☐ Doppler tracking testing
- ☐ System stability testing

2.5.2 Final Documentation

- ☐ Test results publication
- ☐ Configuration guides
- ☐ Troubleshooting guides
- ☐ Performance reports

2.5.3 Community Support

- ☐ Training materials
- ☐ Support documentation
- ☐ Schedule of next opportunity for communications attempts
- ☐ Future development plans

2.6 Ongoing Activities

2.6.1 Community Engagement

- ☒ Regular progress updates (Inner Circle, email, published video recordings)
- ☒ Technical presentations (IEEE vTools events)
- ☐ Conference participation
- ☒ Developer meetings

2.6.2 Development Support

- ☒ Code review process (weekly meetups)
- ☒ Bug tracking (Github Issues Tracker)
- ☒ Feature requests (Slack, email, meetups)
- ☐ Performance monitoring

2.6.3 Documentation Maintenance

- ☐ Regular documentation updates
- ☐ Performance reports
- ☐ Configuration Artifacts Recorded
- ☐ Best practices documentation

2.7 Key Milestones

0. Jupyter Lab Notebook passes review
1. Initial GNU Radio prototype functional
2. First FPGA implementation complete
3. Basic Doppler tracking working
4. Full system integration achieved
5. First successful field tests
6. Release candidate testing complete
7. Production release with documentation

2.8 Success Criteria

- ☐ Successful Doppler tracking at specified rates
- ☐ Reliable demodulation under varying conditions
- ☐ Meeting specified link budget parameters
- ☐ Complete, maintainable documentation
- ☐ Active community engagement
- ☐ Reproducible build process
- ☐ Comprehensive test coverage

2.9 Risk Management

2.9.1 Technical Risks

- ☐ SDR performance limitations
- ☐ FPGA resource constraints
- ☐ Doppler tracking challenges

- ☐ Integration complexities

2.9.2 Mitigation Strategies

- ☐ Early prototyping
- ☐ Regular testing
- ☐ Performance monitoring
- ☐ Community feedback
- ☒ Incremental development
- ☒ Regular reviews

2.10 Resource Requirements

2.10.1 Hardware

- ☒ Development SDRs
- ☒ FPGA development boards
- ☒ Test equipment
- ☒ Antenna systems

2.10.2 Software

- ☒ FPGA development tools
- ☒ GNU Radio environment
- ☒ Testing frameworks
- ☒ Documentation tools

2.10.3 Personnel

- ☒ FPGA developers
- ☐ GNU Radio developers
- ☒ Documentation writers
- ☒ Test engineers
- ☒ Project coordinators

2.11 Review Points

- ☒ Weekly progress reviews
- ☒ Monthly milestone assessments (Inner Circle Newsletter)
- ☒ Community feedback sessions (vTools crosslisting, email list)
- ☐ Performance validation checks
- ☐ Documentation reviews

2.12 Imports and Dataclass Definition

We import the python modules that we need for the project.

[62]:
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
```

```
from dataclasses import dataclass
from typing import Optional, Dict, Any
from skyfield.api import load, wgs84
from datetime import datetime, timedelta, timezone
import matplotlib.pyplot as plt
import pandas as pd
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import math
import plotly.io as pio
pio.renderers.default = "notebook"
```

We set up dataclasses using the naming format LinkParameters. We use numbers we have been given from sites such as Deep Space Exploration Society (DSES), who first asked for assistance with this project. This first part of our analysis sets the values that are true for these sites regardless of the celestial target. For example, 1296 MHz (23cm band) is a common frequency for EME communications, and is the starting point for EVE at DSES. Using a frequency closer to ~2450 MHz would give a bit better performance, and there are plans to move to 2304 MHz in 2026. Therefore, variables such as the frequency used at DSES site (tx_frequency_mhz) can be set in DSESLinkParameters data classes.

[63]:
```
# Dataclass Definitions
@dataclass
class DSESLinkParameters: #things that are true for the site regardless of the␣
 ↪target
    # location of this dish in Google Earth is 38°22'51"N 103°09'22"W
    latitude: float = 38.380833 # from FCC converter
    longitude: float = -103.156111 # from FCC converter
    elevation: float = 1311 # meters, from online calculator
    tx_frequency_mhz: float = 1296.0 # Transmit frequency
#    tx_frequency_mhz: float = 2304.0 # Transmit frequency anticipated for␣
 ↪October 2026
    tx_power_w: float = 1500.0 # Transmit power in watts
    tx_antenna_diameter_m: float = 18.29 # Transmit antenna diameter
    tx_antenna_efficiency: float = 0.69 # Transmit antenna efficiency
    rx_antenna_diameter_m: float = 18.29 # Receive antenna diameter
    rx_antenna_efficiency: float = 0.69 # Receive antenna efficiency
    tx_line_loss_db: float = 0.5 # provided
    rx_line_loss_db: float = 0.5 # provided
    pointing_error_deg: float = 0.01 # provided
    lna_noise_figure_db: float = 0.4 # from datasheet for the Kuhne MKU LNA 132␣
 ↪AH SMA
    lna_noise_figure_K: float = 28.0 # calculated from lna_noise_figure_db
    receiver_noise_bandwidth: float = 100e3 # operational receiver bandwidth.
    c: float = 299792458  # Speed of light in m/s
    k: float = 1.380649e-23  # Boltzmann constant in J/K
    t0: float = 290  # Reference temperature in K
```

2.13 System Noise Temperature Worksheet

2.13.1 Introduction

The system noise temperature is an important factor in determining the sensitivity of a radio telescope or communication system. It represents the total noise from all sources that affects the system's ability to detect weak signals. It is measured in Kelvin (K).

Unlike signal strength, which scales with dish diameter, system noise temperature is largely independent of the physical size of the antenna. Instead, it depends on factors related to the quality of the antenna construction, receiver electronics, and environmental conditions.

2.13.2 Key Components

This system noise temperature calculation has the following components

1. Sky Noise (T_sky): Background radiation from the whole universe anda model of atmospheric contributions. This varies with elevation angle (more atmosphere at lower angles) and weather conditions (clear, cloudy, rainy).
2. Spillover Noise (T_spillover): Noise caused by the antenna feed pattern transmitting energy beyond the dish edges. This means it also picks up thermal radiation from the ground. This is determined by the feed design and positioning.
3. Scatter Noise (T_scatter): Noise resulting from dish surface imperfections that scatter incoming signals. Calculated using the Ruze equation, which relates surface RMS errors to performance degradation at a given wavelength.
4. Receiver Noise (T_receiver): Noise generated by the receiver's electronic components, often specified as noise figure or noise temperature.

2.13.3 Total System Temperature

The total system noise temperature is calculated as:

T_sys = T_ant + T_receiver

Where T_ant (the antenna temperature) is the combination of sky noise, spillover noise, and scatter noise:

T_ant = (main_beam_efficiency * T_sky) + T_spillover + T_scatter

2.13.4 Impact on Performance

A lower system noise temperature directly translates to better sensitivity. For deep space communications, like those with Venus, minimizing each noise component is essential. The main ways to do this are to use high-quality low-noise amplifiers (LNAs). This is a very important factor in reducing receiver noise. Precise dish surfaces minimize scatter noise. If they're out of round or warped, then there's a degradation. Well-designed feeds reduce spillover noise. The feed needs to be matched as well as it can be to the dish dimension. Operating at higher elevation angles when possible reduces sky noise because we're going through less of the atmosphere.

This worksheet implements some calculations for each of these noise components. The goal is to produce a realistic system performance assessment, and to provide a solid system noise temperature to the Link Budget calculation.

[64]:
```
#import numpy as np
#from dataclasses import dataclass
#from typing import Optional, Dict, Any

class SystemNoiseTemperature:
    """
    A class to calculate system noise temperature for radio systems.
    """

    def __init__(self, params: 'DSESLinkParameters'):
        """
        Initialize the system noise temperature calculator.

        Parameters:
        params (DSESLinkParameters): The link parameters object containing␣
↪system settings
        """
        self.params = params

    def calculate_sky_noise(self, elevation_angle_deg: float,␣
↪atmospheric_conditions: str = 'clear') -> float:
        """
        Calculate sky noise temperature based on frequency and elevation angle.

        Parameters:
        elevation_angle_deg (float): Elevation angle in degrees
        atmospheric_conditions (str): Weather conditions ('clear', 'cloudy',␣
↪'rain')

        Returns:
        float: Sky noise temperature in Kelvin
        """
        # Convert elevation angle to radians
        elev_rad = np.radians(elevation_angle_deg)

        # Basic atmospheric attenuation model
        base_temp = 2.7  # cosmic background radiation

        # Atmospheric contribution increases at lower elevation angles
        air_mass = 1.0 / np.sin(elev_rad)

        # Frequency dependent atmospheric absorption
        freq_ghz = self.params.tx_frequency_mhz / 1000.0
```

```
        freq_factor = 0.1 * freq_ghz / 10.0

        # Weather condition factors
        weather_factors = {
            'clear': 1.0,
            'cloudy': 1.5,
            'rain': 3.0
        }

        weather_multiplier = weather_factors.get(atmospheric_conditions, 1.0)

        return base_temp + (270 * (1 - np.exp(-freq_factor * air_mass))) *␣
↪weather_multiplier

    def calculate_spillover_noise(self, spillover_efficiency: float,␣
↪ground_temp: float = 290.0) -> float:
        """
        Calculate spillover noise contribution.

        Parameters:
        spillover_efficiency (float): Efficiency of the antenna's spillover␣
↪(0-1)
        ground_temp (float): Ground temperature in Kelvin

        Returns:
        float: Spillover noise temperature in Kelvin
        """
        return ground_temp * (1 - spillover_efficiency)

    def calculate_scatter_noise(self, surface_rms_mm: float) -> float:
        """
        Calculate scattering noise due to surface imperfections using the Ruze␣
↪equation.

        Parameters:
        surface_rms_mm (float): Root mean square surface error in mm

        Returns:
        float: Scatter noise temperature in Kelvin
        """
        wavelength_mm = 300000 / self.params.tx_frequency_mhz  # Convert MHz to␣
↪wavelength in mm
        surface_efficiency = np.exp(-(4 * np.pi * surface_rms_mm /␣
↪wavelength_mm) ** 2)
        return 290 * (1 - surface_efficiency)
```

```
    def calculate_system_noise(self,
                               main_beam_efficiency: float,
                               spillover_efficiency: float,
                               surface_rms_mm: float,
                               receiver_temp: float,
                               elevation_angle_deg: float,
                               atmospheric_conditions: str = 'clear',
                               ground_temp: float = 290.0) -> Dict[str, float]:
        """
        Calculate total system noise temperature and its components.

        Parameters:
        main_beam_efficiency (float): Main beam efficiency of the antenna (0-1)
        spillover_efficiency (float): Spillover efficiency of the antenna (0-1)
        surface_rms_mm (float): RMS surface error in mm
        receiver_temp (float): Receiver noise temperature in Kelvin
        elevation_angle_deg (float): Elevation angle in degrees
        atmospheric_conditions (str): Weather conditions ('clear', 'cloudy',␣
↪'rain')
        ground_temp (float): Ground temperature in Kelvin

        Returns:
        Dict[str, float]: Dictionary containing total and component temperatures
        """
        # Calculate individual components
        sky_noise = self.calculate_sky_noise(elevation_angle_deg,␣
↪atmospheric_conditions)
        spillover_noise = self.calculate_spillover_noise(spillover_efficiency,␣
↪ground_temp)
        scatter_noise = self.calculate_scatter_noise(surface_rms_mm)

        # Calculate antenna temperature
        t_ant = (main_beam_efficiency * sky_noise +
                spillover_noise + scatter_noise)

        # Calculate total system temperature
        t_sys = t_ant + receiver_temp

        return {
            'T_sys': t_sys,
            'T_ant': t_ant,
            'T_sky': sky_noise,
            'T_spillover': spillover_noise,
            'T_scatter': scatter_noise,
            'T_receiver': receiver_temp
        }
```

```
    def get_noise_temperature_summary(self, **kwargs) -> Dict[str, Any]:
        """
        Returns a comprehensive summary of noise temperature calculations.

        Parameters:
        **kwargs: Keyword arguments that can override default parameters
                  (main_beam_efficiency, spillover_efficiency, surface_rms_mm,
                   receiver_temp, elevation_angle_deg, atmospheric_conditions)

        Returns:
        Dict[str, Any]: Dictionary with noise temperature results and␣
↪parameters used
        """
        # Default values (these could/should be stored in DSESLinkParameters␣
↪instead)
        defaults = {
            'main_beam_efficiency': 0.69,
            'spillover_efficiency': 0.95,
            'surface_rms_mm': 0.3,
            'receiver_temp': self.params.lna_noise_figure_K,  # from given 0.4␣
↪dB NF Kuhne MKU LNA 132 AH SMA - fairly confident on this temperature
            'elevation_angle_deg': 45.0,
            'atmospheric_conditions': 'clear',
            'ground_temp': 290.0
        }

        # Override defaults with any provided keyword arguments.
        # we got a lot of good advice on the defaults, but if we know
        # about some sort of update or change to the defaults, then we
        # provide them to the noise calculator get_noise_temperature_summary␣
↪method.
        params = {**defaults, **kwargs}

        # Calculate system noise
        noise_temps = self.calculate_system_noise(
            main_beam_efficiency=params['main_beam_efficiency'],
            spillover_efficiency=params['spillover_efficiency'],
            surface_rms_mm=params['surface_rms_mm'],
            receiver_temp=params['receiver_temp'],
            elevation_angle_deg=params['elevation_angle_deg'],
            atmospheric_conditions=params['atmospheric_conditions'],
            ground_temp=params['ground_temp']
        )

        # Add parameters used for calculation to results
        result = {
            'noise_temperatures': noise_temps,
```

```
            'parameters_used': params,
            'frequency_mhz': self.params.tx_frequency_mhz
        }

        return result


# Create calculator
params = DSESLinkParameters()
noise_calculator = SystemNoiseTemperature(params)

# Get full noise temperature analysis with default parameters
results = noise_calculator.get_noise_temperature_summary(
)

# Print results
print(f"System Noise Analysis at {results['frequency_mhz']} MHz:")
print(f"Elevation: {results['parameters_used']['elevation_angle_deg']}°")
print(f"Conditions: {results['parameters_used']['atmospheric_conditions']}")
print("\nNoise Temperatures:")
for key, value in results['noise_temperatures'].items():
    print(f"  {key}: {value:.1f} K")



# Get full noise temperature analysis with compromised session parameters
results = noise_calculator.get_noise_temperature_summary(
    elevation_angle_deg=5.0,  # Lower elevation than default - this is a␣
 ↪keyword argument change
    atmospheric_conditions='cloudy'  # Worse conditions than default - this is␣
 ↪a keyword argument change
)

# Print results
print(f"System Noise Analysis at {results['frequency_mhz']} MHz:")
print(f"Elevation: {results['parameters_used']['elevation_angle_deg']}°")
print(f"Conditions: {results['parameters_used']['atmospheric_conditions']}")
print("\nNoise Temperatures:")
for key, value in results['noise_temperatures'].items():
    print(f"  {key}: {value:.1f} K")
```

```
System Noise Analysis at 1296.0 MHz:
Elevation: 45.0°
Conditions: clear

Noise Temperatures:
  T_sys: 47.8 K
```

```
  T_ant: 19.8 K
  T_sky: 7.6 K
  T_spillover: 14.5 K
  T_scatter: 0.1 K
  T_receiver: 28.0 K
System Noise Analysis at 1296.0 MHz:
Elevation: 5.0°
Conditions: cloudy

Noise Temperatures:
  T_sys: 83.1 K
  T_ant: 55.1 K
  T_sky: 58.7 K
  T_spillover: 14.5 K
  T_scatter: 0.1 K
  T_receiver: 28.0 K
```

2.14 EVELinkBudget Class

This link budget class is EVELinkBudget. It targets Venus as a reflective surface. We inherit site specific link parameters (SiteNameLinkParameters) as params and can call upon them in the class. We then set up all our EVE/Venus specific values and define the functions that we need in order to calculate this specific link budget.

We add up all the gains and subtract the losses. This gives power at the receiver. We calculate the noise in our receive bandwidth, and subtract it from power at the receiver. This gives a carrier to noise ratio in dB. Our communications mode must be able to meet this CNR in order to close the link.

[65]:
```python
# Define the EVE link budget here - [ ] # Venus specific
class EVELinkBudget:
    def __init__(self, params: DSESLinkParameters):
        self.params = params

        # Venus characteristics
        self.venus_radius_km = 6051.8  # Venus radius in km
        self.venus_radar_albedo = 0.152  # see Venus radar albedo (Radio Echo␣
 ↪Observations of Venus and Mercury at 23 cm Wavelength, 1965)
                                         # see Variations in the Radar Cross␣
 ↪Section of Venus J. V. Evans Lincoln Laboratory,
                                         # Massachusetts Institute of␣
 ↪Technology (Received 19 December 1967)


    def wavelength(self) -> float:
        """Calculate wavelength in meters from frequency"""
        return self.params.c / (self.params.tx_frequency_mhz * 1e6)
```

```
    def venus_reflection_gain(self) -> float:
        """Calculate reflection gain from Venus surface
        Venus radar cross section =  * radius²"""
        # see also https://hamradio.engineering/
↪eme-path-loss-free-space-loss-passive-reflector-loss/
        venus_radius_m = self.venus_radius_km * 1000
        radar_cross_section = np.pi * venus_radius_m**2
        return 10 * np.log10(radar_cross_section)

    def venus_reflection_loss(self) -> float:
        # use radar albedo for Venus to get this loss
        return 10 * np.log10(self.venus_radar_albedo)

    def tx_antenna_gain(self) -> float:
        """Calculate transmitter antenna gain"""
        efficiency = self.params.tx_antenna_efficiency
        diameter = self.params.tx_antenna_diameter_m
        wavelength = self.wavelength()
        return 10 * np.log10(efficiency * (np.pi * diameter / wavelength) ** 2)

    def rx_antenna_gain(self) -> float:
        """Calculate receiver antenna gain"""
        efficiency = self.params.rx_antenna_efficiency
        diameter = self.params.rx_antenna_diameter_m
        wavelength = self.wavelength()
        return 10 * np.log10(efficiency * (np.pi * diameter / wavelength) ** 2)

    def free_space_loss(self, distance_km: float, round_trip: bool = True) ->␣
↪float:
        """Calculate free space loss, optionally for round trip

        Args:
            distance_km: Distance in kilometers
            round_trip: If True, calculate round trip loss (both directions)
        """
        wavelength = self.wavelength()
        distance_m = distance_km * 1000
        one_way_loss = 20 * np.log10(4 * np.pi * distance_m / wavelength)
        return one_way_loss * 2 if round_trip else one_way_loss

    def pointing_loss(self) -> float:
        """Calculate pointing loss"""
        pointing_error_rad = np.radians(self.params.pointing_error_deg)
        #tracking_error_rad = np.radians(self.params.tracking_error_deg) # if␣
↪we know tracking error
        total_error_rad = np.sqrt(pointing_error_rad**2)
```

```
        #total_error_rad = np.sqrt(pointing_error_rad**2 +␣
↪tracking_error_rad**2) # if we know tracking error
        return -12 * (total_error_rad / self.antenna_beamwidth_rad())**2

    def antenna_beamwidth_rad(self) -> float:
        """Calculate antenna beamwidth in radians"""
        return 1.22 * self.wavelength() / self.params.tx_antenna_diameter_m

    def calculate_link_budget(self, distance_km: float) -> dict:
        """Calculate complete link budget for a given distance"""
        # Convert transmit power to dBW
        tx_power_dbw = 10 * np.log10(self.params.tx_power_w)

        # Calculate antenna gains (only once each for TX and RX)
        tx_gain = self.tx_antenna_gain()
        rx_gain = self.rx_antenna_gain()
        venus_gain = self.venus_reflection_gain()

        # Calculate losses
        fs_loss = self.free_space_loss(distance_km, round_trip=True)
        point_loss = self.pointing_loss()
        venus_loss = self.venus_reflection_loss()

        # Calculate received power (passive reflection scenario)
        rx_power = (
            tx_power_dbw
            + tx_gain  # TX antenna gain
            + rx_gain  # RX antenna gain
            - fs_loss  # Two-way path loss
            - self.params.tx_line_loss_db  # reported TX line loss at site
            - self.params.rx_line_loss_db  # reported RX line loss at site
            - point_loss  # Pointing loss - needs attention
            - venus_loss  # Venus reflection loss
            + venus_gain  # Venus reflection gain
        )

        # Calculate system noise
        #t_sys = 47.8 # copied over by hand from Tsys worksheet cell

        # Get full noise temperature analysis from our noise_calculator and␣
↪then use the calculated Tsys
        results = noise_calculator.get_noise_temperature_summary(
            #elevation_angle_deg=5.0,  # Lower elevation than default - this is␣
↪a keyword argument change
            #atmospheric_conditions='cloudy'  # Worse conditions than default -␣
↪this is a keyword argument change
        )
```

```
        t_sys = results['noise_temperatures']['T_sys']

        noise_dbw = 10 * np.log10(self.params.k * t_sys * self.params.
↪receiver_noise_bandwidth)


        # Calculate CNR
        cnr = rx_power - noise_dbw

        # Calculate CNR in 1Hz
        bandwidth_factor_db = 10 * np.log10(self.params.
↪receiver_noise_bandwidth / 1)
        cnr_db_1hz = cnr + bandwidth_factor_db

        return {
            'tx_power_dbw': tx_power_dbw,
            'radius_venus_km': self.venus_radius_km,
            'venus_radar_albedo': self.venus_radar_albedo,
            'tx_gain_db': tx_gain,
            'rx_gain_db': rx_gain,
            'free_space_loss_db': fs_loss,
            'pointing_loss_db': point_loss,
            'venus_reflection_loss_db': venus_loss,
            'venus_reflection_gain_db': venus_gain,
            'system_noise_temperature': t_sys,
            'rx_power_dbw': rx_power,
            'noise_dbw': noise_dbw,
            'cnr_db': cnr,
            'cnr_db_1hz': cnr_db_1hz
        }
```

2.15 Link Budget Calculator Setup

We fetch all the siteLinkParameters and then create a calculator instance by saying EVELinkBudget(params). We've specified the class of EVELinkBudget. We're now creating an object where we can call the various methods inside of that class, which includes our calculate_link_budget(). Since the distance to Venus varies quite a bit, this is a parameter we need to provide.

Whenever we need to calculate the CNR for our link, we call our_results = calculator.calculate_link_budget(distance to Venus). our_results['cnr_db_1hz'] would be the resulting 1 Hz CNR.

[66]:
```
# Cell 2: Create calculator instance
params = DSESLinkParameters()
calculator = EVELinkBudget(params)
```

```python
min_distance_km = 38_000_000  # Minimum Earth-Venus distance
max_distance_km = 261_000_000  # Maximum Earth-Venus distance

# Print detailed results for minimum and maximum distances - can then be used␣
 ↪in following cells
max_results = calculator.calculate_link_budget(max_distance_km)
min_results = calculator.calculate_link_budget(min_distance_km)

print(f"Link Budget at Minimum Distance (38 million km) at {params.
 ↪receiver_noise_bandwidth/1e6} MHz receiver bandwidth")
for key, value in min_results.items():
    print(f"{key}: {value:.2f}")

print(f"\nLink Budget at Maximum Distance (261 million km) at {params.
 ↪receiver_noise_bandwidth/1e6} MHz receiver bandwidth")
for key, value in max_results.items():
    print(f"{key}: {value:.2f}")
```

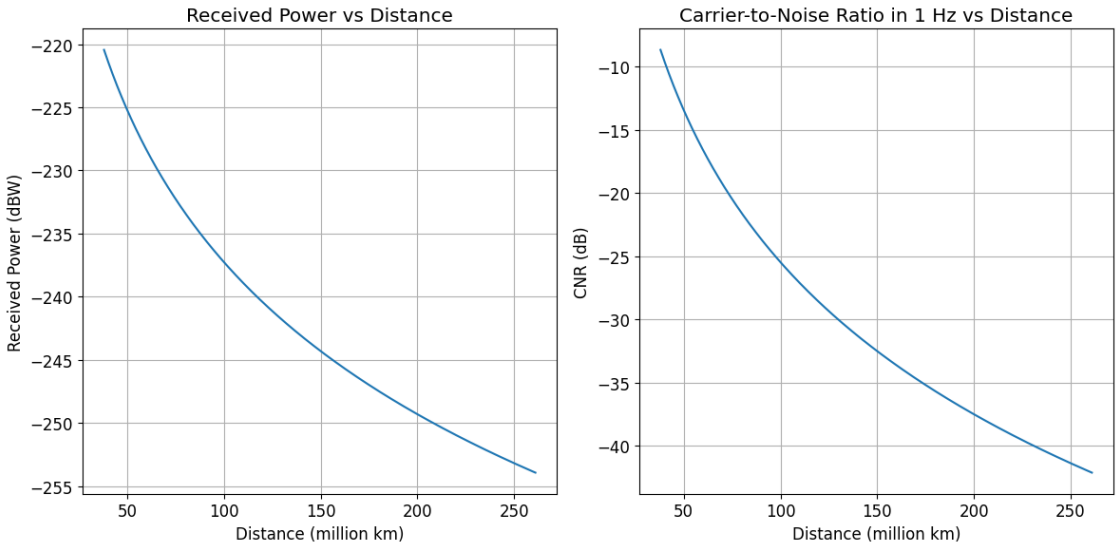
```
Link Budget at Minimum Distance (38 million km) at 0.1 MHz receiver bandwidth
tx_power_dbw: 31.76
radius_venus_km: 6051.80
venus_radar_albedo: 0.15
tx_gain_db: 46.29
rx_gain_db: 46.29
free_space_loss_db: 492.59
pointing_loss_db: -0.00
venus_reflection_loss_db: -8.18
venus_reflection_gain_db: 140.61
system_noise_temperature: 47.82
rx_power_dbw: -220.46
noise_dbw: -161.80
cnr_db: -58.65
cnr_db_1hz: -8.65

Link Budget at Maximum Distance (261 million km) at 0.1 MHz receiver bandwidth
tx_power_dbw: 31.76
radius_venus_km: 6051.80
venus_radar_albedo: 0.15
tx_gain_db: 46.29
rx_gain_db: 46.29
free_space_loss_db: 526.07
pointing_loss_db: -0.00
venus_reflection_loss_db: -8.18
venus_reflection_gain_db: 140.61
system_noise_temperature: 47.82
rx_power_dbw: -253.93
noise_dbw: -161.80
```

```
cnr_db: -92.13
cnr_db_1hz: -42.13
```

2.16 Effect of Distance on Received Power and Carrier-to-Noise Ratio

The variation in distance from Earth to Venus results in a large variation in the received power at the site and in the carrier to noise ratio at the site. This visualization shows the differences between the closest path and the furthest path for radio work.

[67]:
```
# Create distance array for plotting
distances = np.linspace(min_distance_km, max_distance_km, 1000)
cnrs = []
rx_powers = []

for dist in distances:
    results = calculator.calculate_link_budget(dist)
    cnrs.append(results['cnr_db_1hz'])
    rx_powers.append(results['rx_power_dbw'])

# Cell 4: Create plots
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(distances/1e6, rx_powers)
plt.grid(True)
plt.xlabel('Distance (million km)')
plt.ylabel('Received Power (dBW)')
plt.title('Received Power vs Distance')

plt.subplot(1, 2, 2)
plt.plot(distances/1e6, cnrs)
plt.grid(True)
plt.xlabel('Distance (million km)')
plt.ylabel('CNR (dB)')
plt.title('Carrier-to-Noise Ratio in 1 Hz vs Distance')
plt.tight_layout()
plt.show()
```



2.17 Pointing and Tracking Error Worksheets

Dish antennas have a narrow beamwidth. Pointing and tracking errors cost us in our link budget. This section models the pointing and tracking errors so that those amounts can be included in the link budget calculations. We return the pointing loss that corresponds to 1dB and 3dB signal loss. We plot pointing error vs. loss in a chart.

To ensure sensitivity and resolution, pointing errors need to be kept very small, ideally within 1/10th of the half-power beam width.

[68]:
```
# Cell 5: Parameters for dish analysis of pointing error
class AntennaAnalysis:
    def __init__(self, params: DSESLinkParameters):
        self.params = params

    def wavelength(self) -> float:
        """Calculate wavelength in meters from frequency in MHz"""
        frequency_hz = self.params.tx_frequency_mhz * 1e6  # Convert MHz to Hz
        return self.params.c / frequency_hz

    def beamwidth_deg(self) -> float:
        """Calculate 3dB beamwidth in degrees"""
        # Using 1.22 /D formula for circular aperture
        return np.degrees(1.22 * self.wavelength() / self.params.
 ↪tx_antenna_diameter_m)

    def pointing_loss_vs_error(self, max_error_deg: float = 0.5):
        """Calculate pointing loss for range of pointing errors"""
        errors = np.linspace(0, max_error_deg, 100)
```

```
        losses = -12 * (np.radians(errors) / np.radians(self.
 ↪beamwidth_deg()))**2
        return errors, losses

# Create analyzer using params from link budget
analyzer = AntennaAnalysis(params)

# Calculate key parameters
beamwidth = analyzer.beamwidth_deg()
print(f"For a {analyzer.params.tx_antenna_diameter_m:.2f}m dish at {analyzer.
 ↪params.tx_frequency_mhz:.1f} MHz:")
print(f"3dB Beamwidth: {beamwidth:.3f} degrees")
print(f"Recommended max tracking error (1dB loss): {beamwidth/5.66:.3f}␣
 ↪degrees")
print(f"Recommended max tracking error (3dB loss): {beamwidth/2:.3f} degrees")
```

```
For a 18.29m dish at 1296.0 MHz:
3dB Beamwidth: 0.884 degrees
Recommended max tracking error (1dB loss): 0.156 degrees
Recommended max tracking error (3dB loss): 0.442 degrees
```

Let's explain these calculations because I didn't find it intuitive at first.

1. `beamwidth = analyzer.beamwidth_deg()`
   - This calculates the 3dB beamwidth (where power drops to half)
   - Uses the formula 1.22 * /D for a circular aperture
   - This is the "width" of your main beam
2. The tracking error recommendations come from the pointing loss formula we used earlier:

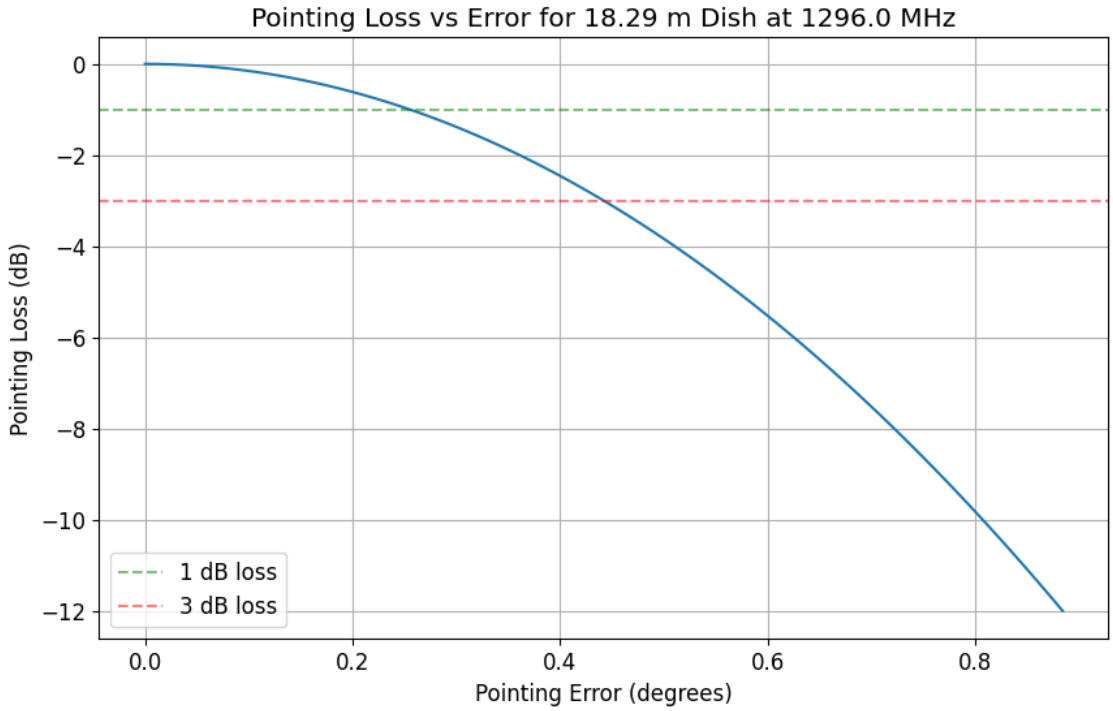
   `pointing_loss = -12 * (error_angle / beamwidth)**2`

   This wasn't intuitive but it's in a lot of papers and seems legit.
3. Working backwards from this formula:
   - For 1dB loss: -1 = -12 * (error/beamwidth)**2
   - Solving for error: error = beamwidth/√12 beamwidth/5.66
   - For 3dB loss: -3 = -12 * (error/beamwidth)**2
   - Solving for error: error = beamwidth/2

So if your beamwidth is (for example) 0.6 degrees: - For 1dB loss: max error should be 0.6°/5.66 0.106° - For 3dB loss: max error should be 0.6°/2 = 0.3°

[69]:
```
# Plot pointing loss vs tracking error normalized by beamwidth
errors, losses = analyzer.pointing_loss_vs_error(max_error_deg=beamwidth)

plt.figure(figsize=(10, 6))
plt.plot(errors, losses)
```

```
plt.grid(True)
plt.xlabel('Pointing Error (degrees)')
plt.ylabel('Pointing Loss (dB)')
plt.title(f'Pointing Loss vs Error for {params.tx_antenna_diameter_m:.2f} m␣
 ↪Dish at {analyzer.params.tx_frequency_mhz:.1f} MHz')

# Add horizontal lines for common loss thresholds
plt.axhline(y=-1, color='g', linestyle='--', alpha=0.5, label='1 dB loss')
plt.axhline(y=-3, color='r', linestyle='--', alpha=0.5, label='3 dB loss')
plt.legend()
plt.show()
```



2.18 Pointing Error as a function of Tracking Error Normalized by Beamwidth

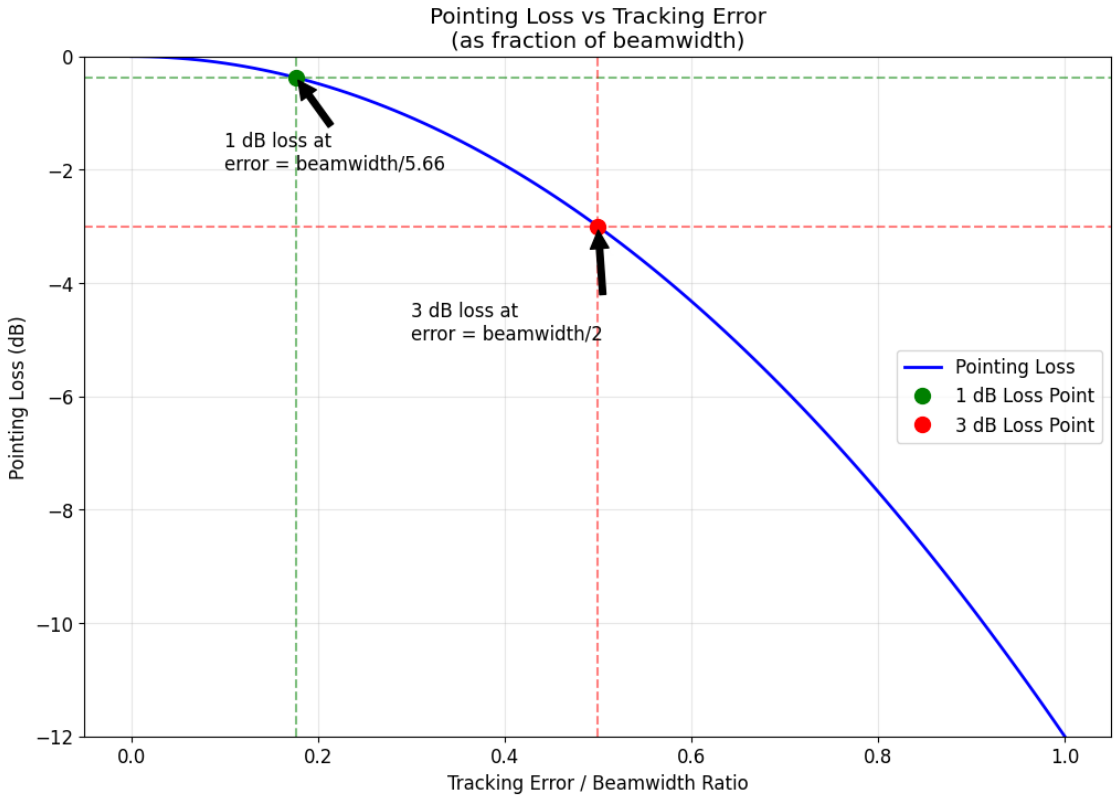
This is an experimental section to try and quantify the tracking error. Pointing error and tracking error are two different things, but notice that we get the same numerical results. Feedback here welcome and encouraged.

[70]:
```
# Cell 5: dish analysis of tracking error
#The visualization shows:
#The blue curve is the pointing loss vs tracking error (normalized to beamwidth)
#The green point shows where tracking error = beamwidth/5.66, giving 1 dB loss
#The red point shows where tracking error = beamwidth/2, giving 3 dB loss
```

```
#If your tracking error is less than beamwidth/5.66, your pointing loss is less␣
 ↪than 1 dB
#At beamwidth/2, you've lost half your power (3 dB)
#The loss increases with the square of the error

def calculate_pointing_loss(error_angle, beamwidth):
    """Calculate pointing loss in dB given error angle and beamwidth"""
    return -12 * (error_angle / beamwidth)**2

# Create a range of error angles as fraction of beamwidth
error_fractions = np.linspace(0, 1, 100)

# Calculate losses
losses = calculate_pointing_loss(error_fractions, 1)

# Create the plot
plt.figure(figsize=(12, 8))

# Main loss curve
plt.plot(error_fractions, losses, 'b-', linewidth=2, label='Pointing Loss')

#Calculate losses at our specific points
loss_at_166 = calculate_pointing_loss(1/5.66, 1)
loss_at_half = calculate_pointing_loss(1/2, 1)

print(f"Loss at error/beamwidth = 1/5.66: {loss_at_166:.2f} dB")
print(f"Loss at error/beamwidth = 1/2: {loss_at_half:.2f} dB")

# Add markers for specific loss points
plt.plot(1/5.66, loss_at_166, 'go', markersize=10, label='1 dB Loss Point')
plt.plot(1/2, loss_at_half, 'ro', markersize=10, label='3 dB Loss Point')

# Add horizontal lines for loss levels
plt.axhline(y=loss_at_166, color='g', linestyle='--', alpha=0.5)
plt.axhline(y=loss_at_half, color='r', linestyle='--', alpha=0.5)

# Add vertical lines for error ratios
plt.axvline(x=1/5.66, color='g', linestyle='--', alpha=0.5)
plt.axvline(x=1/2, color='r', linestyle='--', alpha=0.5)

# Customize the plot
plt.grid(True, alpha=0.3)
plt.xlabel('Tracking Error / Beamwidth Ratio')
plt.ylabel('Pointing Loss (dB)')
plt.title('Pointing Loss vs Tracking Error\n(as fraction of beamwidth)')

#Calculate losses at our specific points
```

```
loss_at_166 = calculate_pointing_loss(1/5.66, 1)
loss_at_half = calculate_pointing_loss(1/2, 1)

print(f"Loss at error/beamwidth = 1/5.66: {loss_at_166:.2f} dB")
print(f"Loss at error/beamwidth = 1/2: {loss_at_half:.2f} dB")

# Add text annotations
plt.annotate('1 dB loss at\nerror = beamwidth/5.66',
             xy=(1/5.66, loss_at_166), xytext=(0.1, -2),
             arrowprops=dict(facecolor='black', shrink=0.05))

plt.annotate('3 dB loss at\nerror = beamwidth/2',
             xy=(1/2, loss_at_half), xytext=(0.3, -5),
             arrowprops=dict(facecolor='black', shrink=0.05))

plt.legend()

# Set y-axis limits to focus on relevant range
plt.ylim(-12, 0)

plt.show()

# Print example calculations for your specific dish
beamwidth = analyzer.beamwidth_deg()
print(f"\nFor your {params.tx_antenna_diameter_m:.2f}m dish at {params.
  ↪tx_frequency_mhz/1e6:.1f} MHz:")
print(f"Beamwidth: {beamwidth:.3f}°")
print(f"Max tracking error for 1 dB loss: {beamwidth/5.66:.3f}°")
print(f"Max tracking error for 3 dB loss: {beamwidth/2:.3f}°")
```

```
Loss at error/beamwidth = 1/5.66: -0.37 dB
Loss at error/beamwidth = 1/2: -3.00 dB
Loss at error/beamwidth = 1/5.66: -0.37 dB
Loss at error/beamwidth = 1/2: -3.00 dB
```




```
For your 18.29m dish at 0.0 MHz:
Beamwidth: 0.884°
Max tracking error for 1 dB loss: 0.156°
Max tracking error for 3 dB loss: 0.442°
```

2.19 How much Doppler do we Have?

We will have to anticipate and "track out" Doppler and understand and tolerate a rate of change of this Doppler shift. A third factor is Doppler spread, which is what happens when a signal bounces off a rotating reflector.

A Doppler spread penalty calculation is currently in the Mode Evaluator cell.

How much Doppler do we have to deal with? This section calculates Doppler shift and the rate of change of the Doppler shift, and visualizes the results.

[71]:
```
#import numpy as np
#from skyfield.api import load, wgs84
#from datetime import datetime, timedelta, timezone
#import matplotlib.pyplot as plt

class DopplerCalculator:
```

```
    """
    A class to calculate Doppler shift between Earth and Venus.

    This class uses the Skyfield library to obtain planetary ephemeris data
    from publicly available sources and calculate the relative velocity and
    resulting Doppler shift and shift rate for radio communications.
    """

    def __init__(self, params: DSESLinkParameters):
        self.params = params
        """
        Initialize the Doppler calculator with one of our site-specific␣
↪dataclasses.

        Parameters:
        -----------
        tx_frequency_mhz : float from <site>LinkParameters
            The transmission frequency in MHz (eg. 1296.0 MHz)
        """

        # Convert MHz to Hz for calculations
        self.frequency_hz = self.params.tx_frequency_mhz * 1e6

        # Load ephemeris data
        self.ephemeris = load('de421.bsp')
        self.earth = self.ephemeris['earth']
        self.venus = self.ephemeris['venus']

        # Time scale
        self.ts = load.timescale()

        # Observer location (initially None)
        self.has_observer = False

    def set_frequency(self, frequency_mhz):
        """
        Update the transmission frequency, outside of the dataclass.

        Parameters:
        -----------
        frequency_mhz : float
            The new transmission frequency in MHz
        """
        self.frequency_hz = frequency_mhz * 1e6

    def calculate_doppler(self, timestamp=None):
        """
```

```
        Calculate the Doppler shift at a specific time.

        Parameters:
        -----------
        timestamp : datetime, optional
            The time at which to calculate the Doppler shift (default: current␣
↪time)
            If timezone-naive, UTC will be used

        Returns:
        --------
        tuple:
            - Frequency shift in Hz
            - Received frequency in Hz
            - Relative velocity in m/s (positive: moving apart, negative:␣
↪moving closer)
        """
        if timestamp is None:
            # Make sure we use timezone-aware UTC time
            timestamp = datetime.now(timezone.utc)
        elif timestamp.tzinfo is None:
            # If the timestamp is naive (no timezone), assume it's in UTC
            timestamp = timestamp.replace(tzinfo=timezone.utc)

        # Convert to Skyfield time. This is called a "decomposed" approach
        t = self.ts.utc(timestamp.year, timestamp.month, timestamp.day,
                        timestamp.hour, timestamp.minute, timestamp.second +␣
↪timestamp.microsecond/1000000.0)

        # Get the position and velocity vectors
        earth_pos = self.earth.at(t)
        venus_pos = self.venus.at(t)

        # Calculate relative position and velocity
        relative = earth_pos.observe(self.venus)
        distance = relative.distance().km
        relative_velocity = relative.velocity.km_per_s

        # The radial velocity component (positive means moving apart)
        radial_velocity = np.dot(relative_velocity, relative.position.km /␣
↪distance)

        # Convert km/s to m/s
        radial_velocity_m_s = radial_velocity * 1000

        # Calculate Doppler shift
        # If objects are moving apart (positive velocity), frequency decreases
```

```
        doppler_shift = -self.frequency_hz * radial_velocity_m_s / self.params.c
        received_frequency = self.frequency_hz + doppler_shift

        return doppler_shift, received_frequency, radial_velocity_m_s

    def calculate_doppler_rate(self, timestamp=None, delta_hours=24):
        """
        Calculate the rate of change of the Doppler shift over a specified time␣
↪period.

        Parameters:
        -----------
        timestamp : datetime, optional
            The time at which to start the calculation (default: current time)
            If timezone-naive, UTC will be assumed
        delta_hours : float, optional
            The time period in hours over which to calculate the rate (default:␣
↪24 hours)

        Returns:
        --------
        tuple:
            - Rate of Doppler shift change in Hz/hour
            - Rate of Doppler shift change in Hz/second
            - Doppler shift at start time in Hz
            - Doppler shift at end time in Hz
        """
        if timestamp is None:
            timestamp = datetime.now(timezone.utc)
        elif timestamp.tzinfo is None:
            timestamp = timestamp.replace(tzinfo=timezone.utc)

        # Calculate Doppler shift at the start time
        doppler_start, _, _ = self.calculate_doppler(timestamp)

        # Calculate Doppler shift at the end time
        end_time = timestamp + timedelta(hours=delta_hours)
        doppler_end, _, _ = self.calculate_doppler(end_time)

        # Calculate the rate of change
        delta_doppler = doppler_end - doppler_start
        rate_per_hour = delta_doppler / delta_hours
        rate_per_second = rate_per_hour / 3600

        return rate_per_hour, rate_per_second, doppler_start, doppler_end

    def generate_doppler_curve(self, start_date, end_date, num_points=100):
```

```
        """
        Generate a curve of Doppler shift over a time period.

        Parameters:
        -----------
        start_date : datetime
            The starting date for the curve (if timezone-naive, UTC will be␣
↪assumed)
        end_date : datetime
            The ending date for the curve (if timezone-naive, UTC will be␣
↪assumed)
        num_points : int, optional
            Number of points to calculate (default: 100)

        Returns:
        --------
        tuple:
            - List of datetime objects
            - List of Doppler shifts in Hz
            - List of relative velocities in m/s
        """
        # Ensure both dates have timezone information
        if start_date.tzinfo is None:
            start_date = start_date.replace(tzinfo=timezone.utc)
        if end_date.tzinfo is None:
            end_date = end_date.replace(tzinfo=timezone.utc)

        time_delta = (end_date - start_date) / num_points

        times = []
        doppler_shifts = []
        velocities = []

        for i in range(num_points + 1):
            current_time = start_date + i * time_delta
            try:
                doppler_shift, _, velocity = self.
↪calculate_doppler(current_time)

                times.append(current_time)
                doppler_shifts.append(doppler_shift)
                velocities.append(velocity)
            except Exception as e:
                print(f"Error calculating Doppler at time {current_time}: {e}")
                # Continue with the loop but skip this problematic point
                continue
```

```
        return times, doppler_shifts, velocities

    def calculate_worst_case_doppler(self, start_date=None, duration_days=584):
        """
        Calculate the worst-case Doppler shift and rate of change over a␣
↪complete synodic period.

        The synodic period of Venus is approximately 584 days (the time it␣
↪takes for Earth
        and Venus to return to the same relative positions).

        Parameters:
        -----------
        start_date : datetime, optional
            The starting date for the analysis (default: current time)
        duration_days : int, optional
            The duration in days to analyze (default: 584, one synodic period)

        Returns:
        --------
        dict:
            Dictionary containing:
            - max_doppler_shift: Maximum absolute Doppler shift in Hz
            - min_doppler_shift: Minimum Doppler shift in Hz (most negative)
            - max_doppler_shift_date: Date of maximum Doppler shift
            - min_doppler_shift_date: Date of minimum Doppler shift
            - max_rate: Maximum absolute rate of change in Hz/hour
            - max_rate_date: Date of maximum rate of change
            - max_rate_per_second: Maximum rate in Hz/second
            - total_doppler_range: Total range of Doppler shift in Hz
        """
        if start_date is None:
            start_date = datetime.now(timezone.utc)
        elif start_date.tzinfo is None:
            start_date = start_date.replace(tzinfo=timezone.utc)

        end_date = start_date + timedelta(days=duration_days)

        # Use enough points to get good resolution (at least one point per day)
        num_points = max(1000, duration_days * 4)

        times, shifts, velocities = self.generate_doppler_curve(start_date,␣
↪end_date, num_points)

        # Calculate rates of change
        rates = []
        rate_times = []
```

```
        for i in range(len(shifts) - 1):
            delta_t = (times[i+1] - times[i]).total_seconds() / 3600  # in hours
            rate = (shifts[i+1] - shifts[i]) / delta_t  # Hz per hour
            rates.append(rate)
            rate_times.append(times[i])

        # Find extremes for Doppler shift
        max_shift_idx = np.argmax(shifts)
        min_shift_idx = np.argmin(shifts)
        max_shift = shifts[max_shift_idx]
        min_shift = shifts[min_shift_idx]
        max_shift_date = times[max_shift_idx]
        min_shift_date = times[min_shift_idx]

        # Find extreme for rate of change (maximum absolute value)
        abs_rates = np.abs(rates)
        max_rate_idx = np.argmax(abs_rates)
        max_rate = rates[max_rate_idx]
        max_rate_date = rate_times[max_rate_idx]
        max_rate_per_second = max_rate / 3600

        # Calculate total Doppler range
        total_doppler_range = max_shift - min_shift

        return {
            'max_doppler_shift': max_shift,
            'min_doppler_shift': min_shift,
            'max_doppler_shift_date': max_shift_date,
            'min_doppler_shift_date': min_shift_date,
            'max_rate': max_rate,
            'max_rate_date': max_rate_date,
            'max_rate_per_second': max_rate_per_second,
            'total_doppler_range': total_doppler_range
        }

    def plot_doppler_curve(self, start_date, end_date, num_points=100,␣
↪save_path=None, include_rate=True):
        """
        Generate and plot a curve of Doppler shift over a time period.

        Parameters:
        -----------
        start_date : datetime
            The starting date for the curve
        end_date : datetime
            The ending date for the curve
```

```
        num_points : int, optional
            Number of points to calculate (default: 100)
        save_path : str, optional
            Path to save the plot (default: None, plot is displayed instead)
        include_rate : bool, optional
            Whether to include a subplot showing the rate of change (default:␣
↪True)
        """
        times, shifts, velocities = self.generate_doppler_curve(start_date,␣
↪end_date, num_points)

        # Calculate rate of change for each point except the last one
        rates = []
        for i in range(len(shifts) - 1):
            delta_t = (times[i+1] - times[i]).total_seconds() / 3600  # in hours
            rate = (shifts[i+1] - shifts[i]) / delta_t  # Hz per hour
            rates.append(rate)

        # Add a placeholder for the last point to keep the arrays the same␣
↪length
        rates.append(rates[-1] if rates else 0)

        # Determine the number of subplots
        n_plots = 3 if include_rate else 2

        fig, axes = plt.subplots(n_plots, 1, figsize=(12, 4*n_plots),␣
↪sharex=True)

        # Plot Doppler shift
        axes[0].plot(times, [s/1000 for s in shifts], 'b-')
        axes[0].set_ylabel('Doppler Shift (kHz)')
        axes[0].set_title(f'Earth-Venus Doppler Shift at {self.frequency_hz/1e6:
↪.3f} MHz')
        axes[0].grid(True)

        # Plot velocity
        axes[1].plot(times, velocities, 'r-')
        axes[1].set_ylabel('Relative Velocity (m/s)')
        axes[1].set_title('Earth-Venus Relative Velocity')
        axes[1].grid(True)

        # Plot rate of change if requested
        if include_rate:
            axes[2].plot(times, rates, 'g-')
            axes[2].set_xlabel('Date')
            axes[2].set_ylabel('Rate of Change (Hz/hour)')
            axes[2].set_title('Doppler Shift Rate of Change')
```

```
            axes[2].grid(True)
        else:
            axes[1].set_xlabel('Date')

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path)
        else:
            plt.show()


    # fixed this with a lot of help - it wasn't updating past the first location
    def set_observer_location(self, latitude, longitude, elevation=0,␣
↪location_name=None):
            """
            Set the Earth-based observer's location.

            Parameters:
            -----------
            latitude : float
                Observer's latitude in degrees (positive for north, negative␣
↪for south)
            longitude : float
                Observer's longitude in degrees (positive for east, negative␣
↪for west)
            elevation : float, optional
                Observer's elevation above sea level in meters (default: 0)
            location_name : str, optional
                Name of the location (e.g., 'Site 1', 'New York', etc.)
            """
            # Create a completely new observer_location object EVERY time
            self.observer_location = None  # Clear the old one first
            self.observer_location = wgs84.latlon(latitude, longitude,␣
↪elevation_m=elevation)
            self.has_observer = True
            self.location_name = location_name if location_name else␣
↪f"({latitude:.2f}°, {longitude:.2f}°)"

            # Force the location to be recreated the next time it's used
            if hasattr(self, '_location_object'):
                del self._location_object


    def calculate_doppler_from_location(self, timestamp=None):
        """
```

```
        Calculate the Doppler shift at a specific time from the observer's location.
        This includes the additional effect of Earth's rotation.

        Parameters:
        -----------
        timestamp : datetime, optional
            The time at which to calculate the Doppler shift (default: current time)
            If timezone-naive, UTC will be assumed

        Returns:
        --------
        tuple:
            - Frequency shift in Hz
            - Received frequency in Hz
            - Relative velocity in m/s
            - Venus altitude in degrees (for visibility determination)
            - Venus azimuth in degrees
        """
        if not self.has_observer:
            raise ValueError("Observer location not set. Use set_observer_location() first.")

        if timestamp is None:
            timestamp = datetime.now(timezone.utc)
        elif timestamp.tzinfo is None:
            timestamp = timestamp.replace(tzinfo=timezone.utc)

        # Convert to Skyfield time
        t = self.ts.utc(timestamp.year, timestamp.month, timestamp.day,
                        timestamp.hour, timestamp.minute, timestamp.second + timestamp.microsecond/1000000.0)

        # Create a Skyfield location object for the observer
        location = self.earth + self.observer_location

        # Get topocentric position (from observer's location)
        venus_apparent = location.at(t).observe(self.venus).apparent()

        # Get altitude and azimuth for visibility determination
        alt, az, distance = venus_apparent.altaz()

        # Get the velocity component
        relative_velocity = venus_apparent.velocity.km_per_s

        # The radial velocity component (positive means moving apart)
```

```
        radial_velocity = np.dot(relative_velocity, venus_apparent.position.km /
↪ venus_apparent.distance().km)

        # Convert km/s to m/s
        radial_velocity_m_s = radial_velocity * 1000

        # Calculate Doppler shift
        doppler_shift = -self.frequency_hz * radial_velocity_m_s / self.params.c
        received_frequency = self.frequency_hz + doppler_shift

        return doppler_shift, received_frequency, radial_velocity_m_s, alt.
↪degrees, az.degrees

    def is_visible_from_location(self, timestamp=None, min_altitude=10):
        """
        Determine if Venus is visible from the observer's location.

        Parameters:
        -----------
        timestamp : datetime, optional
            The time to check visibility (default: current time)
        min_altitude : float, optional
            Minimum altitude in degrees for visibility (default: 10)

        Returns:
        --------
        bool:
            True if Venus is above the minimum altitude, False otherwise
        """
        if not self.has_observer:
            raise ValueError("Observer location not set. Use␣
↪set_observer_location() first.")

        _, _, _, altitude, _ = self.calculate_doppler_from_location(timestamp)
        return altitude > min_altitude

    def generate_location_doppler_curve(self, start_date, end_date,␣
↪num_points=100,
                                        include_visibility=True, min_altitude=10):
        """
        Generate a curve of Doppler shift from the observer's location over a␣
↪time period.

        Parameters:
        -----------
        start_date, end_date : datetime
            The time range to generate the curve for
```

```
        num_points : int, optional
            Number of points to calculate (default: 100)
        include_visibility : bool, optional
            Whether to mask points where Venus is not visible (default: True)
        min_altitude : float, optional
            Minimum altitude in degrees for visibility (default: 10)

        Returns:
        --------
        tuple:
            - List of datetime objects
            - List of Doppler shifts in Hz
            - List of relative velocities in m/s
            - List of Venus altitudes in degrees
            - List of visibility flags
        """
        if not self.has_observer:
            raise ValueError("Observer location not set. Use␣
↪set_observer_location() first.")

        # Ensure both dates have timezone information
        if start_date.tzinfo is None:
            start_date = start_date.replace(tzinfo=timezone.utc)
        if end_date.tzinfo is None:
            end_date = end_date.replace(tzinfo=timezone.utc)

        time_delta = (end_date - start_date) / num_points

        times = []
        doppler_shifts = []
        velocities = []
        altitudes = []
        visibilities = []

        for i in range(num_points + 1):
            current_time = start_date + i * time_delta
            try:
                doppler_shift, _, velocity, alt, _ = self.
↪calculate_doppler_from_location(current_time)
                is_visible = alt > min_altitude

                times.append(current_time)
                doppler_shifts.append(doppler_shift)
                velocities.append(velocity)
                altitudes.append(alt)
                visibilities.append(is_visible)
            except Exception as e:
```

```
                print(f"Error calculating Doppler at time {current_time}: {e}")
                continue

        # If including visibility, mask points where Venus is not visible
        if include_visibility:
            masked_shifts = []
            masked_velocities = []
            for i in range(len(visibilities)):
                if not visibilities[i]:
                    # Set to None to create gaps in the plot
                    masked_shifts.append(None)
                    masked_velocities.append(None)
                else:
                    masked_shifts.append(doppler_shifts[i])
                    masked_velocities.append(velocities[i])
            doppler_shifts = masked_shifts
            velocities = masked_velocities

        return times, doppler_shifts, velocities, altitudes, visibilities

    def plot_location_doppler_curve(self, start_date, end_date, num_points=100,
                                    save_path=None, include_visibility=True,␣
↪min_altitude=10):
        """
        Generate and plot a curve of Doppler shift from the observer's location.

        Parameters:
        -----------
        start_date, end_date : datetime
            The time range to generate the curve for
        num_points : int, optional
            Number of points to calculate (default: 100)
        save_path : str, optional
            Path to save the plot (default: None, plot is displayed instead)
        include_visibility : bool, optional
            Whether to mask points where Venus is not visible (default: True)
        min_altitude : float, optional
            Minimum altitude in degrees for visibility (default: 10)
        """
        if not self.has_observer:
            raise ValueError("Observer location not set. Use␣
↪set_observer_location() first.")

        times, shifts, velocities, altitudes, visibilities = \
            self.generate_location_doppler_curve(start_date, end_date,␣
↪num_points,
                                                 include_visibility, min_altitude)
```

```
        # Calculate rates of change
        rates = []
        rate_times = []

        for i in range(len(shifts) - 1):
            # Skip None values (when Venus is not visible)
            if shifts[i] is None or shifts[i+1] is None:
                continue

            delta_t = (times[i+1] - times[i]).total_seconds() / 3600  # in hours
            rate = (shifts[i+1] - shifts[i]) / delta_t  # Hz per hour
            rates.append(rate)
            rate_times.append(times[i])

        # Create subplots: Doppler, Velocity, Altitude, Rate
        fig, axes = plt.subplots(4, 1, figsize=(12, 16), sharex=True)

        # Plot Doppler shift
        axes[0].plot(times, [s/1000 if s is not None else None for s in␣
↪shifts], 'b-')
        axes[0].set_ylabel('Doppler Shift (kHz)')
        axes[0].set_title(f'Earth-Venus Doppler Shift at {self.frequency_hz/1e6:
↪.3f} MHz from {self.location_name}')
        axes[0].grid(True)

        # Plot velocity
        axes[1].plot(times, velocities, 'r-')
        axes[1].set_ylabel('Relative Velocity (m/s)')
        axes[1].grid(True)

        # Plot altitude
        axes[2].plot(times, altitudes, 'g-')
        axes[2].set_ylabel('Venus Altitude (°)')
        if include_visibility:
            axes[2].axhline(y=min_altitude, color='r', linestyle='--',
                            label=f'Minimum altitude ({min_altitude}°)')
            axes[2].legend()
        axes[2].grid(True)

        # Plot rate of change
        if rate_times:  # Only if we have valid rate data
            axes[3].plot(rate_times, rates, 'm-')
            axes[3].set_xlabel('Date')
            axes[3].set_ylabel('Rate of Change (Hz/hour)')
            axes[3].grid(True)
```

```
        plt.tight_layout()

        if save_path:
            plt.savefig(save_path)
        else:
            plt.show()

    def calculate_location_worst_case_doppler(self, start_date=None,␣
↪duration_days=30,
                                              include_visibility=True,␣
↪min_altitude=10):
        """
        Calculate the worst-case Doppler shift and rate of change for a␣
↪specific location.

        Parameters:
        -----------
        start_date : datetime, optional
            The starting date for the analysis (default: current time)
        duration_days : int, optional
            The duration in days to analyze (default: 30 days)
        include_visibility : bool, optional
            Whether to only consider times when Venus is visible (default: True)
        min_altitude : float, optional
            Minimum altitude in degrees for visibility (default: 10)

        Returns:
        --------
        dict:
            Dictionary containing worst-case metrics for the visible periods
        """
        if not self.has_observer:
            raise ValueError("Observer location not set. Use␣
↪set_observer_location() first.")

        if start_date is None:
            start_date = datetime.now(timezone.utc)
        elif start_date.tzinfo is None:
            start_date = start_date.replace(tzinfo=timezone.utc)

        end_date = start_date + timedelta(days=duration_days)

        # Generate data with higher resolution
        num_points = max(1000, duration_days * 24)  # At least one point per␣
↪hour
        times, shifts, velocities, altitudes, visibilities = \
```

```
            self.generate_location_doppler_curve(start_date, end_date,␣
↪num_points,
                                                 include_visibility, min_altitude)

        # Calculate rates for valid points
        rates = []
        rate_times = []

        for i in range(len(shifts) - 1):
            if shifts[i] is None or shifts[i+1] is None:
                continue

            delta_t = (times[i+1] - times[i]).total_seconds() / 3600  # in hours
            rate = (shifts[i+1] - shifts[i]) / delta_t  # Hz per hour
            rates.append(rate)
            rate_times.append(times[i])

        # Filter out None values (when Venus is not visible)
        valid_shifts = [s for s in shifts if s is not None]

        if not valid_shifts:
            return {
                'no_visibility': True,
                'message': f"Venus is not visible above {min_altitude}° during␣
↪the specified period"
            }

        # Find extremes for Doppler shift
        max_shift = max(valid_shifts)
        min_shift = min(valid_shifts)

        # Find times of extremes
        max_shift_idx = shifts.index(max_shift)
        min_shift_idx = shifts.index(min_shift)
        max_shift_date = times[max_shift_idx]
        min_shift_date = times[min_shift_idx]

        # Max altitude
        max_alt = max(altitudes)
        max_alt_idx = altitudes.index(max_alt)
        max_alt_date = times[max_alt_idx]

        # Find extreme for rate of change
        if rates:
            abs_rates = [abs(r) for r in rates]
            max_rate_idx = abs_rates.index(max(abs_rates))
            max_rate = rates[max_rate_idx]
```

```
            max_rate_date = rate_times[max_rate_idx]
            max_rate_per_second = max_rate / 3600
        else:
            max_rate = 0
            max_rate_date = None
            max_rate_per_second = 0

        # Calculate total Doppler range
        total_doppler_range = max_shift - min_shift

        # Calculate visibility statistics
        if include_visibility:
            visible_periods = []
            current_period = None

            for i, visible in enumerate(visibilities):
                if visible and current_period is None:
                    # Start of a new visibility period
                    current_period = {'start': times[i]}
                elif not visible and current_period is not None:
                    # End of a visibility period
                    current_period['end'] = times[i-1]
                    visible_periods.append(current_period)
                    current_period = None

            # Handle case where Venus is still visible at the end
            if current_period is not None:
                current_period['end'] = times[-1]
                visible_periods.append(current_period)

            # Calculate visibility statistics
            total_visible_hours = 0
            for period in visible_periods:
                duration = (period['end'] - period['start']).total_seconds() /␣
↪3600
                total_visible_hours += duration

            visibility_percentage = (total_visible_hours / (duration_days *␣
↪24)) * 100
        else:
            visible_periods = []
            visibility_percentage = 100  # Not considering visibility

        return {
            'max_doppler_shift': max_shift,
            'min_doppler_shift': min_shift,
            'max_doppler_shift_date': max_shift_date,
```

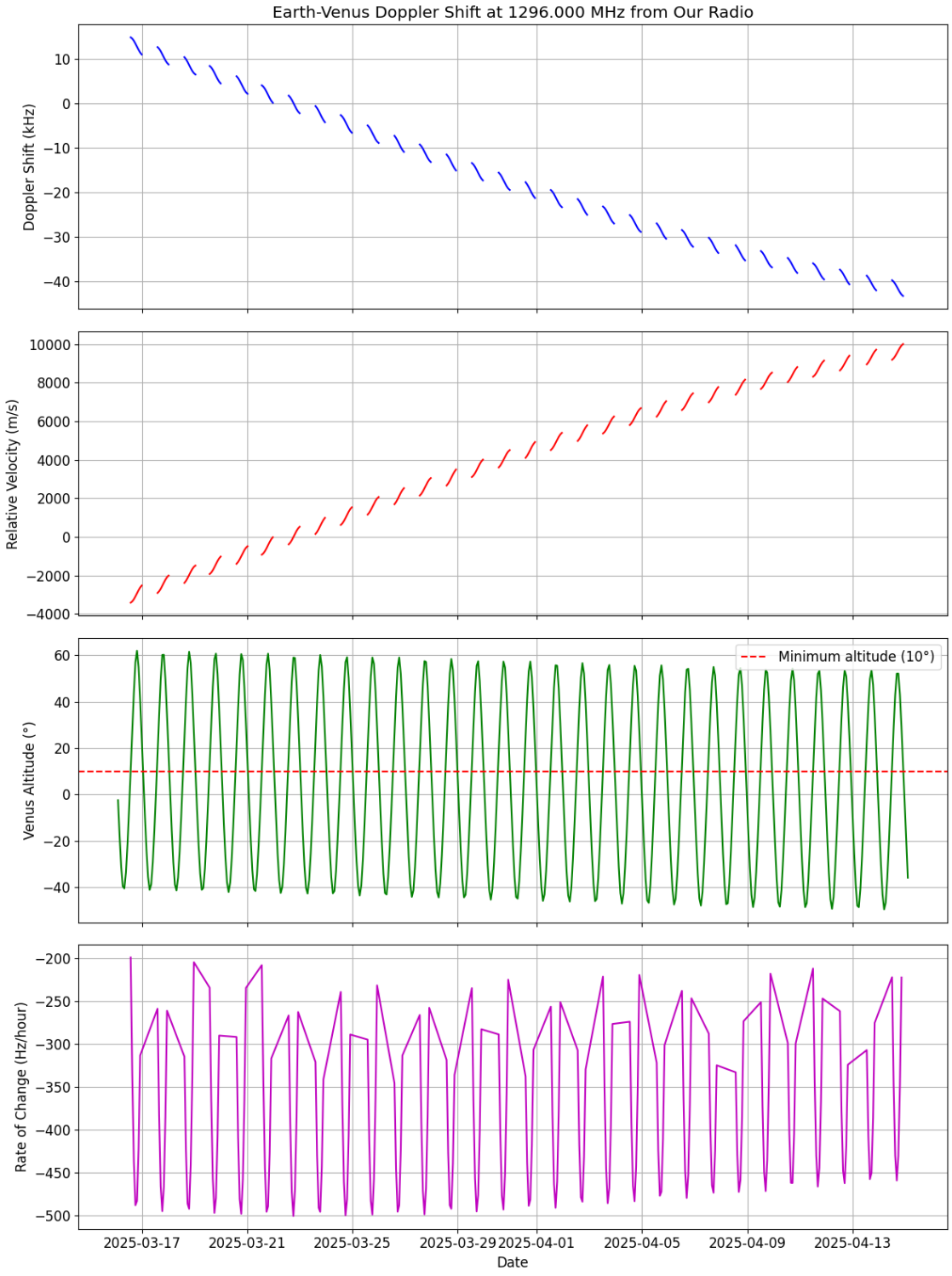
```
            'min_doppler_shift_date': min_shift_date,
            'max_rate': max_rate,
            'max_rate_date': max_rate_date,
            'max_rate_per_second': max_rate_per_second,
            'total_doppler_range': total_doppler_range,
            'max_altitude': max_alt,
            'max_altitude_date': max_alt_date,
            'visibility_percentage': visibility_percentage,
            'visible_periods': visible_periods,
            'no_visibility': False
        }



# Example usage

# Create a calculator using our transmit frequency from dataclass␣
 ↪<site>LinkParameters
DopplerCalculator = DopplerCalculator(params)

# Example using From Center-of-the-Earth calculations
print("\n-------------------------------------------------------------")
print("From Center-of-the-Earth Doppler calculations")
print("-------------------------------------------------------------")

# Calculate current Doppler shift - explicitly using UTC time
current_time_utc = datetime.now(timezone.utc)
print(f"Calculating Doppler for current time: {current_time_utc}")
shift, freq, velocity = DopplerCalculator.calculate_doppler(current_time_utc)
print(f"Current Earth-Venus:")
print(f"  Relative velocity: {velocity:.2f} m/s")
print(f"  Doppler shift: {shift:.2f} Hz")
print(f"  Received frequency: {freq/1e6:.6f} MHz")

# Calculate the rate of change of Doppler shift over the next 24 hours
rate_hour, rate_sec, doppler_start, doppler_end = DopplerCalculator.
 ↪calculate_doppler_rate(current_time_utc)
print(f"\nDoppler shift rate of change (next 24 hours):")
print(f"  Rate: {rate_hour:.2f} Hz/hour or {rate_sec:.4f} Hz/second")
print(f"  Starting Doppler: {doppler_start:.2f} Hz")
print(f"  Ending Doppler: {doppler_end:.2f} Hz")
print(f"  Total change: {doppler_end - doppler_start:.2f} Hz")

# Calculate worst-case Doppler scenarios
print(f"\nCalculating worst-case Doppler scenarios for a complete orbit cycle...
 ↪")
worst_case = DopplerCalculator.calculate_worst_case_doppler()
```

```
print(f"Worst-case Doppler shift:")
print(f"  Maximum (positive) shift: {worst_case['max_doppler_shift']:.2f} Hz on␣
 ↪{worst_case['max_doppler_shift_date']}")
print(f"  Minimum (negative) shift: {worst_case['min_doppler_shift']:.2f} Hz on␣
 ↪{worst_case['min_doppler_shift_date']}")
print(f"  Total Doppler range: {worst_case['total_doppler_range']:.2f} Hz")

print(f"\nWorst-case Doppler rate of change:")
print(f"  Maximum rate: {worst_case['max_rate']:.2f} Hz/hour or␣
 ↪{worst_case['max_rate_per_second']:.6f} Hz/second")
print(f"  Occurs on: {worst_case['max_rate_date']}")

# Example using Earth location-specific calculations
print("\n----------------------------------------------------------------")
print("Earth location-specific Doppler calculations")
print("----------------------------------------------------------------")

# We can define a set of sites - first one is from our site-specific dataclass
sites = [
    {"name": "Our Radio", "latitude": params.latitude, "longitude": params.
 ↪longitude, "elevation": params.elevation},  # from dataclass
    {"name": "London", "latitude": 51.5074, "longitude": -0.1278, "elevation":␣
 ↪25},    # London
    {"name": "Sidney", "latitude": -33.8688, "longitude": 151.2093, "elevation":
 ↪ 3}   # Sydney
]

# Test calculations for each site
for site in sites:
    print(f"\nSetting location to {site['name']} ({site['latitude']}°,␣
 ↪{site['longitude']}°)")
    DopplerCalculator.set_observer_location(site['latitude'],␣
 ↪site['longitude'], site['elevation'], location_name=site['name'])

    # Calculate current Doppler from this location
    try:
        doppler, freq, velocity, altitude, azimuth = DopplerCalculator.
 ↪calculate_doppler_from_location(current_time_utc)
        print(f"Current Earth-Venus from {site['name']}:")
        print(f"  Venus altitude: {altitude:.2f}° / azimuth: {azimuth:.2f}°")
        print(f"  Relative velocity: {velocity:.2f} m/s")
        print(f"  Doppler shift: {doppler:.2f} Hz")
        print(f"  Received frequency: {freq/1e6:.6f} MHz")
        print(f"  Venus is {'visible' if altitude > 10 else 'not visible'}␣
 ↪(assuming min altitude of 10°)")
```

```
        # Calculate worst case for this location over next 30 days
        print(f"\n  Calculating worst-case scenarios for {site['name']} over␣
↪next 30 days...")
        location_worst_case = DopplerCalculator.
↪calculate_location_worst_case_doppler()

        if location_worst_case.get('no_visibility', False):
            print(f"  {location_worst_case['message']}")
        else:
            print(f"  Maximum visible altitude:␣
↪{location_worst_case['max_altitude']:.2f}° on␣
↪{location_worst_case['max_altitude_date']}")
            print(f"  Venus visibility:␣
↪{location_worst_case['visibility_percentage']:.1f}% of time")
            print(f"  Maximum Doppler shift:␣
↪{location_worst_case['max_doppler_shift']:.2f} Hz")
            print(f"  Minimum Doppler shift:␣
↪{location_worst_case['min_doppler_shift']:.2f} Hz")
            print(f"  Maximum rate of change: {location_worst_case['max_rate']:.
↪2f} Hz/hour or {location_worst_case['max_rate_per_second']:.6f} Hz/second")


            # Uncomment to generate plots for each site
            DopplerCalculator.plot_location_doppler_curve(
                start_date=current_time_utc,
                end_date=current_time_utc + timedelta(days=30),
                num_points=500
            )

    except Exception as e:
        print(f"Error calculating for {site['name']}: {e}")

# Generate a Doppler curve for the next 30 days - explicitly using UTC
start = datetime.now(timezone.utc)
end = start + timedelta(days=30)

print(f"\nGenerating center-of-Earth Doppler curve from {start} to {end}")
DopplerCalculator.plot_doppler_curve(start, end, num_points=500)
```

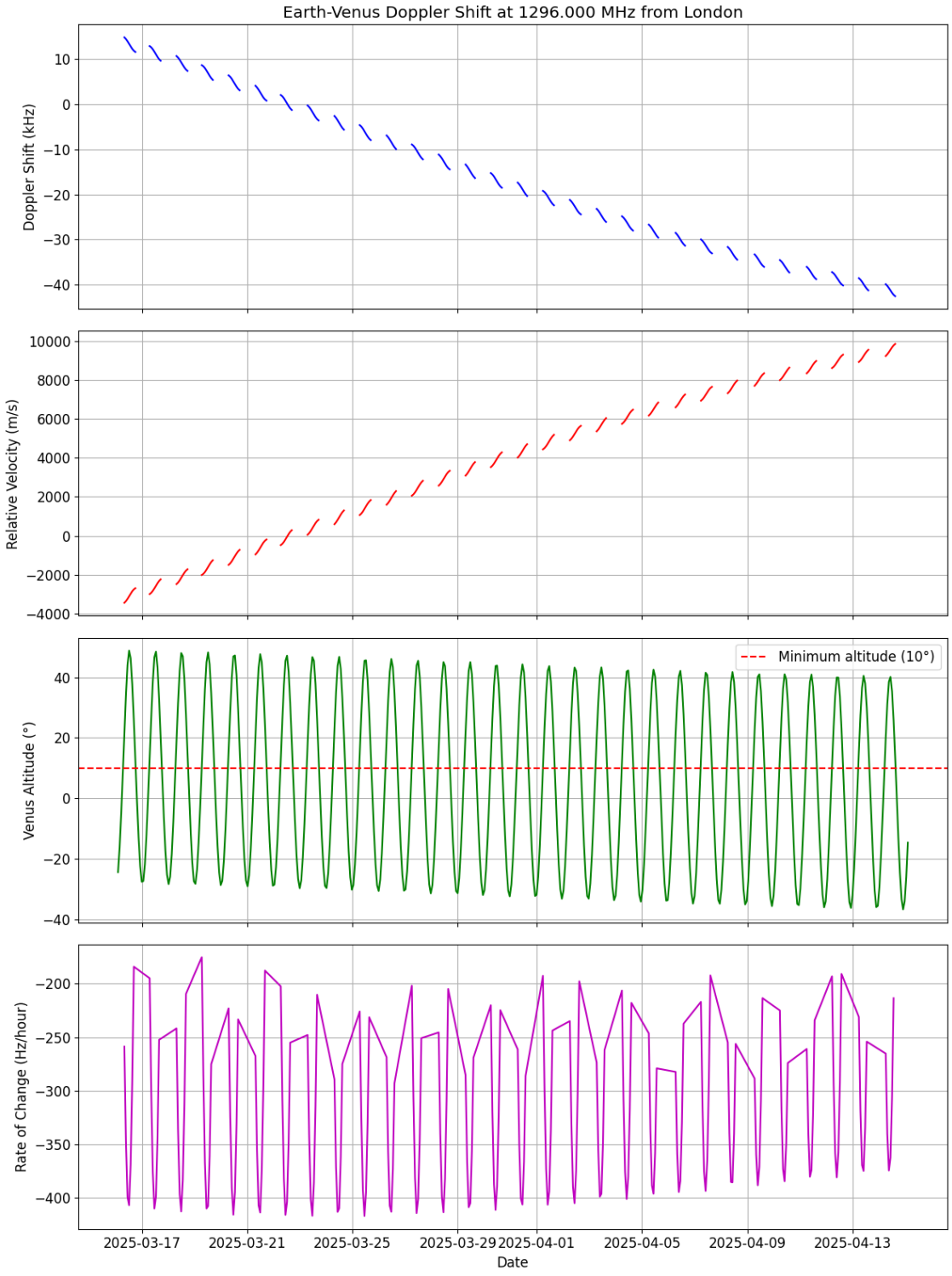
```
------------------------------------------------------------
From Center-of-the-Earth Doppler calculations
------------------------------------------------------------
Calculating Doppler for current time: 2025-03-16 02:15:05.100958+00:00
Current Earth-Venus:
  Relative velocity: -3290.08 m/s
```

```
  Doppler shift: 14222.98 Hz
  Received frequency: 1296.014223 MHz

Doppler shift rate of change (next 24 hours):
  Rate: -86.69 Hz/hour or -0.0241 Hz/second
  Starting Doppler: 14222.98 Hz
  Ending Doppler: 12142.49 Hz
  Total change: -2080.49 Hz

Calculating worst-case Doppler scenarios for a complete orbit cycle…
Worst-case Doppler shift:
  Maximum (positive) shift: 59965.12 Hz on 2026-08-04 08:15:05.110849+00:00
  Minimum (negative) shift: -60143.27 Hz on 2025-06-03 08:15:05.110849+00:00
  Total Doppler range: 120108.39 Hz

Worst-case Doppler rate of change:
  Maximum rate: -92.75 Hz/hour or -0.025764 Hz/second
  Occurs on: 2025-03-22 20:15:05.110849+00:00

------------------------------------------------------------
Earth location-specific Doppler calculations
------------------------------------------------------------

Setting location to Our Radio (38.380833°, -103.156111°)
Current Earth-Venus from Our Radio:
  Venus altitude: -2.46° / azimuth: 285.49°
  Relative velocity: -2937.43 m/s
  Doppler shift: 12698.49 Hz
  Received frequency: 1296.012698 MHz
  Venus is not visible (assuming min altitude of 10°)

  Calculating worst-case scenarios for Our Radio over next 30 days…
  Maximum visible altitude: 61.94° on 2025-03-16 19:31:54.146979+00:00
  Venus visibility: 42.6% of time
  Maximum Doppler shift: 14775.41 Hz
  Minimum Doppler shift: -43273.41 Hz
  Maximum rate of change: -502.85 Hz/hour or -0.139680 Hz/second
```



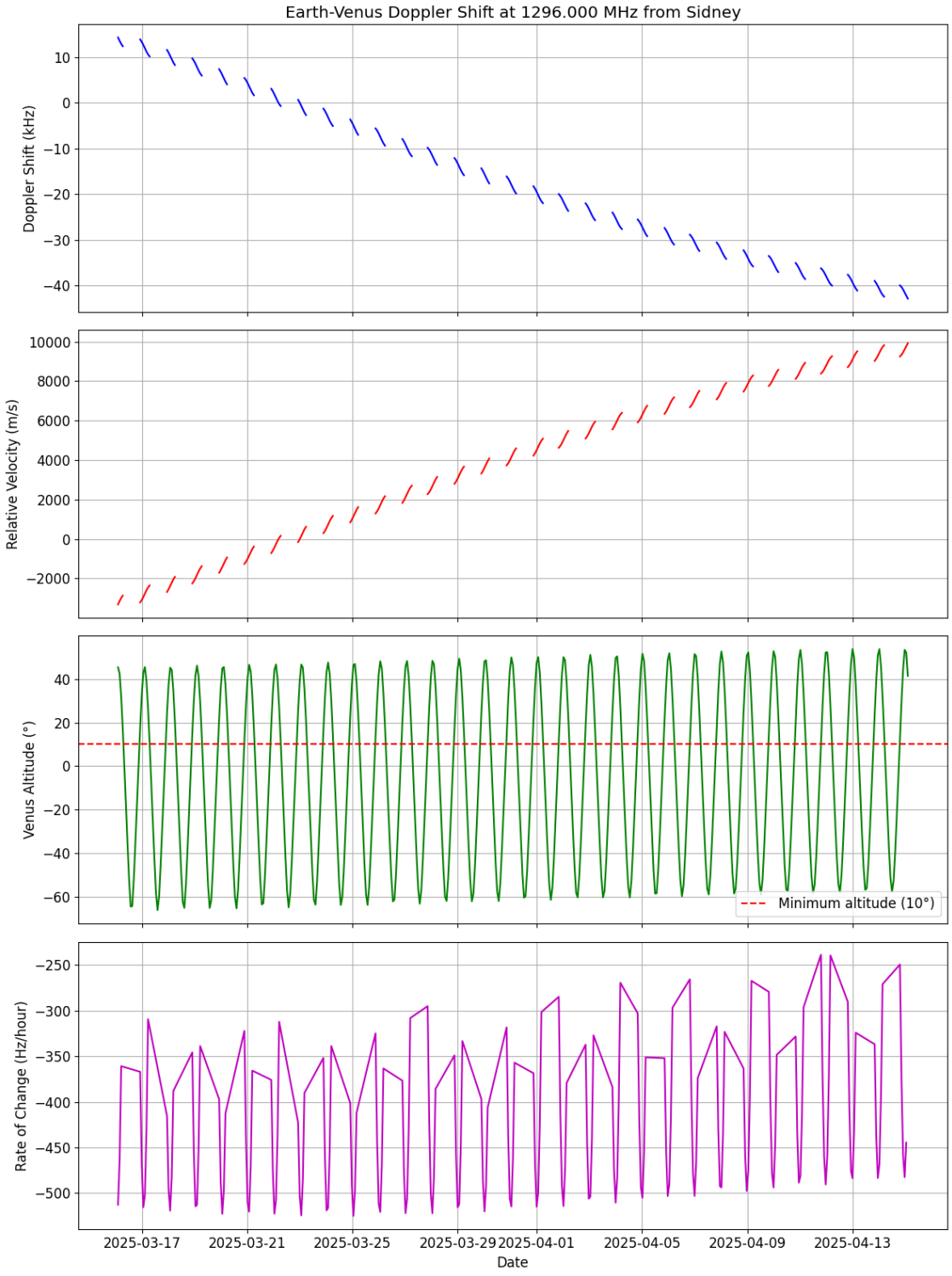

```
Setting location to London (51.5074°, -0.1278°)
Current Earth-Venus from London:
```

```
Venus altitude: -24.40° / azimuth: 26.84°
Relative velocity: -3408.86 m/s
Doppler shift: 14736.48 Hz
Received frequency: 1296.014736 MHz
Venus is not visible (assuming min altitude of 10°)

Calculating worst-case scenarios for London over next 30 days…
Maximum visible altitude: 48.83° on 2025-03-16 12:19:55.419312+00:00
Venus visibility: 42.1% of time
Maximum Doppler shift: 15000.62 Hz
Minimum Doppler shift: -42580.56 Hz
Maximum rate of change: -418.56 Hz/hour or -0.116266 Hz/second
```

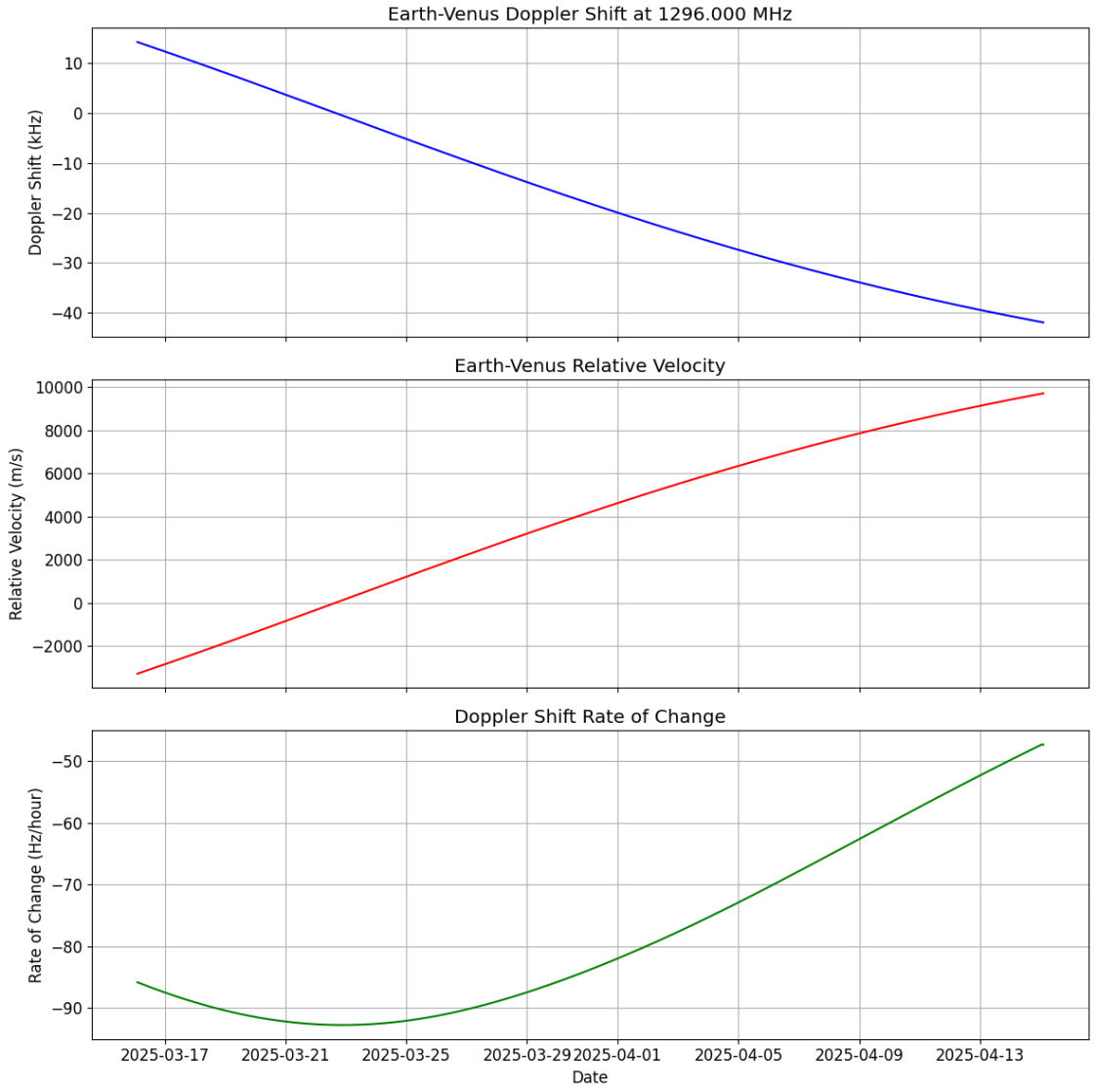



```
Setting location to Sidney (-33.8688°, 151.2093°)
Current Earth-Venus from Sidney:
```

```
Venus altitude: 45.45° / azimuth: 5.53°
Relative velocity: -3315.95 m/s
Doppler shift: 14334.84 Hz
Received frequency: 1296.014335 MHz
Venus is visible (assuming min altitude of 10°)

Calculating worst-case scenarios for Sidney over next 30 days…
Maximum visible altitude: 54.28° on 2025-04-15 00:05:32.694734+00:00
Venus visibility: 38.2% of time
Maximum Doppler shift: 14334.32 Hz
Minimum Doppler shift: -42907.21 Hz
Maximum rate of change: -526.72 Hz/hour or -0.146312 Hz/second
```




```
Generating center-of-Earth Doppler curve from 2025-03-16 02:15:09.947247+00:00
to 2025-04-15 02:15:09.947247+00:00
```



2.20 Doppler Spread for Venus

What is Doppler Spread and why do we care? Doppler spread happens when our signal reflects off a rotating structure, like Venus. Part of Venus is moving towards us and part of it is moving away from us. The signal reflected off the center of the planet is reflected back with little to no frequency change.

For the visualization below, which is a combination from Stack Overflow posts, my code, reviewer feedback, and Claude.ai advice, we calculate the Doppler spread. Once that is set, then we manually put in the different mode bandwidths with mode_bandwidth to get a bar chart of Doppler Spread loss in dB.

Gary K6MG writes "A rough calculation of venus limb-to-limb doppler spreading @ 1296MHz: 4 * venus rotation velocity 1.8 m/s / 3e8 m/s * 1.296e9 c/s = 31 c/s. This calculation is the same

as K1JT uses for EME in Frequency-Dependent Characteristics of the EME Path and is motivated from first principles. One edge of venus is approaching the earth at a 1.8m/s velocity relative to the center of venus, the other edge is receding at the same velocity giving one factor of 2. The other factor of 2 is due to the reflection, the wave is shortened or lengthened on both the approach and the retreat."

This would be super useful if it was interactive with sliders, but I don't (yet) know how to do that. I tried to get Claude.ai to help make that happen, but none of the solutions worked for me.

[88]:
```
#import plotly.graph_objects as go
#from plotly.subplots import make_subplots
#import numpy as np
#import math

class EVEDopplerSpread:
    def __init__(self, params: DSESLinkParameters):
        self.params = params

    def calculate_penalty(self, doppler_spread, mode_bandwidth):
        if doppler_spread > mode_bandwidth:
            penalty = 10 * math.log10(doppler_spread / mode_bandwidth) # our␣
 ↪simple model of Doppler spread penalty
            return min(penalty, 20) # limit the damage to 20 dB - after that it␣
 ↪doesn't really matter.
        return 0

    def venus_doppler_spread(self):
        # Convert MHz to Hz for calculations
        frequency_hz = self.params.tx_frequency_mhz * 1e6
        doppler_spread = (4 * 1.8 * frequency_hz) / self.params.c
        print(f"We calculated a Doppler spread of: {doppler_spread:.2f} Hz")
        return doppler_spread

# Create a calculator for Doppler Spread
DopplerSpreadCalculator = EVEDopplerSpread(params)

# Set parameters
doppler_spread = DopplerSpreadCalculator.venus_doppler_spread()
mode_bandwidth = 90  # Change this value to see different scenarios (Hz)

# Calculate penalty
penalty = DopplerSpreadCalculator.calculate_penalty(doppler_spread,␣
 ↪mode_bandwidth)
max_doppler = 300 # put a limit of the penalty

# Create figure with subplots
fig = make_subplots(
    rows=3, cols=1,
```

```
    subplot_titles=(
        "Frequency Domain Visualization",
        "Doppler Penalty vs. Spread",
        "Mode Comparison with Current Doppler Spread"
    ),
    vertical_spacing=0.1,
    specs=[[{"type": "xy"}], [{"type": "xy"}], [{"type": "xy"}]]
)

# Plot 1: Frequency domain visualization
x = np.linspace(-200, 200, 400)
center_freq = 0

# Signal with Doppler spread represented as a Gaussian function
# The limits of the Doppler spread Gaussian are set by the physical size of␣
 ↪Venus
# This value comes from DopplerSpreadCalculator.venus_doppler_spread()
signal = np.exp(-((x - center_freq) ** 2) / (2 * (doppler_spread/2) ** 2))

# Mode bandwidth indicator
bandwidth_x = x[(np.abs(x) <= mode_bandwidth/2)]
bandwidth_y = np.ones_like(bandwidth_x) * 0.5

fig.add_trace(
    go.Scatter(
        x=x, y=signal,
        fill='tozeroy',
        name='Signal Power',
        line=dict(color='blue'),
        fillcolor='rgba(0, 0, 255, 0.2)'
    ),
    row=1, col=1
)

fig.add_trace(
    go.Scatter(
        x=bandwidth_x, y=bandwidth_y,
        fill='tozeroy',
        name='Mode Bandwidth',
        line=dict(color='green'),
        fillcolor='rgba(0, 255, 0, 0.2)'
    ),
    row=1, col=1
)

# Plot 2: Doppler Penalty vs. Spread calculated and shown
spreads = np.linspace(0, max_doppler, 1000)
```

```
penalties = np.array([DopplerSpreadCalculator.calculate_penalty(s,␣
 ↪mode_bandwidth) for s in spreads])

fig.add_trace(
    go.Scatter(
        x=spreads, y=penalties,
        name='Penalty Function',
        line=dict(color='orange', width=3)
    ),
    row=2, col=1
)

fig.add_trace(
    go.Scatter(
        x=[mode_bandwidth, mode_bandwidth], y=[0, 20],
        name='Bandwidth Threshold',
        line=dict(color='green', width=2, dash='dash')
    ),
    row=2, col=1
)

fig.add_trace(
    go.Scatter(
        x=[doppler_spread], y=[penalty],
        name='Current Setting',
        mode='markers',
        marker=dict(color='red', size=12)
    ),
    row=2, col=1
)

# Add annotation for formula
if doppler_spread > mode_bandwidth:
    formula_text = f"Penalty = 10*log ({doppler_spread}/{mode_bandwidth}) =␣
 ↪{penalty:.2f} dB"
else:
    formula_text = "No penalty applied"

fig.add_annotation(
    x=max_doppler * 0.85, y=15,
    text=formula_text,
    showarrow=False,
    bgcolor="white",
    bordercolor="black",
    borderwidth=1,
    xref="x2", yref="y2"
)
```

```
# Plot 3: Mode Comparison
modes = ["FT8", "Q65-60A", "Q65-60B", "Q65-60C", "JT65", "CW", "SSB"]
bandwidths = [50, 65, 90, 180, 2.7, 250, 2500]
mode_penalties = [DopplerSpreadCalculator.calculate_penalty(doppler_spread, bw)␣
 ↪for bw in bandwidths]

colors = ['red' if p > 10 else 'green' for p in mode_penalties]

fig.add_trace(
    go.Bar(
        y=modes, x=mode_penalties,
        orientation='h',
        marker_color=colors,
        text=[f"{p:.1f} dB" for p in mode_penalties],
        textposition='outside'
    ),
    row=3, col=1
)

# Use this figure layout for better PDF rendering
fig.update_layout(
    height=900,
    width=800,
    title_text=f"Doppler Spread Effects (Spread: {doppler_spread:.1f} Hz,␣
 ↪Bandwidth: {mode_bandwidth} Hz)",
    showlegend=False,
    margin=dict(l=50, r=50, t=100, b=50),
    paper_bgcolor='white',
    plot_bgcolor='white',
    font=dict(family="Arial, sans-serif", size=14)
)

fig.update_xaxes(title_text="Frequency Offset (Hz)", range=[-200, 200], row=1,␣
 ↪col=1)
fig.update_yaxes(title_text="Power", range=[0, 1.1], row=1, col=1)

fig.update_xaxes(title_text="Doppler Spread (Hz)", range=[0, max_doppler],␣
 ↪row=2, col=1)
fig.update_yaxes(title_text="Penalty (dB)", range=[0, 21], row=2, col=1)

fig.update_xaxes(title_text="Penalty (dB)", range=[0, 21], row=3, col=1)

# Show figure
fig.show()
```

```
We calculated a Doppler spread of: 31.13 Hz
```

2.21 Mode Analysis

This class does an analysis of potential modes and their suitability for the the link.

The relationship between the CNR from the link budget object (calculated at the operational receiver bandwidth) and SNR of particular signals (given at different bandwidths) may be needed. The conversion is not always straightforward. The listed SNRs for many amateur modes are given with a bandwidth, but that given SNR is not calculated at that bandwidth, but is calculated at another "normalized" bandwidth. The most common "normalized" bandwidth is 2500 Hz. When we know this is the case, we can list this number (2500 Hz) as the "real" noise bandwidth as the "noise_bandwidth_hz" value.

The mathematics to make sure we're

SNR = CNR × (BW_CNR / BW_SNR)

We've updated our Doppler spread model to use a Gaussian representation (as see in the Doppler Spread section above) which better reflects the physical reality of how Doppler affects signals. Instead of assuming a flat distribution of energy across frequencies (a linear model), we model the Doppler-shifted energy as following a bell curve centered at the carrier frequency.

The penalty is calculated by determining what percentage of the signal's energy falls outside the mode's bandwidth. For modes with bandwidth much wider than the Doppler spread, almost all energy is contained within the bandwidth and there's minimal penalty. As the bandwidth narrows relative to the spread, more energy falls outside the usable bandwidth, increasing the penalty. The modes most affected are the very narrow-band modes.

This Gaussian approach is more accurate because it accounts for the concentration of energy near the carrier frequency with decreasing energy at the edges, which corresponds to the probability distribution of relative velocities in the signal path.

[89]:
```
#import numpy as np
#import pandas as pd

class AmateurRadioModeEvaluator:
    """
    A class to evaluate the suitability of amateur radio modes based on link␣
 ↪parameters.
    """

    def __init__(self, params: DSESLinkParameters):
        self.params = params
        # Define common amateur radio modes with their characteristics
        self.modes = [
            {"name": "CW", "bandwidth_hz": 250, "required_snr_db": -15,␣
 ↪"noise_bandwidth_hz": 250},
            {"name": "FT8", "bandwidth_hz": 50, "required_snr_db": -20,␣
 ↪"noise_bandwidth_hz": 2500},
            {"name": "JT65", "bandwidth_hz": 2.7, "required_snr_db": -25,␣
 ↪"noise_bandwidth_hz": 2500},
```

```
            {"name": "SSB", "bandwidth_hz": 2500, "required_snr_db": 8,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "FM", "bandwidth_hz": 12500, "required_snr_db": 12,␣
↪"noise_bandwidth_hz": 12500},
            {"name": "RTTY", "bandwidth_hz": 250, "required_snr_db": 5,␣
↪"noise_bandwidth_hz": 250},
            {"name": "PSK31", "bandwidth_hz": 31, "required_snr_db": 4,␣
↪"noise_bandwidth_hz": 31},
            {"name": "FT4", "bandwidth_hz": 90, "required_snr_db": -17,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "JS8", "bandwidth_hz": 30, "required_snr_db": -18,␣
↪"noise_bandwidth_hz": 2500},

            # WSPR Modes
            {"name": "WSPR-15", "bandwidth_hz": 6, "required_snr_db": -32,␣
↪"noise_bandwidth_hz": 2500},   # 15-minute
            {"name": "WSPR-2", "bandwidth_hz": 6, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},    # 2-minute mode (classic WSPR mode??)
            {"name": "WSPR-120", "bandwidth_hz": 6, "required_snr_db": -37,␣
↪"noise_bandwidth_hz": 2500},  # 120-minute mode,
            {"name": "WSPR-LF", "bandwidth_hz": 6, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500},   # Low frequency band modes (??)
            {"name": "WSPR-H", "bandwidth_hz": 12, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500},   # High-speed variant with doubled bandwidth

            # Q65 modes (various submodes A-E with different tone spacing and␣
↪durations)
            {"name": "Q65-15A", "bandwidth_hz": 65, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500},  # 15-second mode A
            {"name": "Q65-30A", "bandwidth_hz": 65, "required_snr_db": -27,␣
↪"noise_bandwidth_hz": 2500},  # 30-second mode A
            {"name": "Q65-60A", "bandwidth_hz": 65, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},  # 60-second mode A
            {"name": "Q65-120A", "bandwidth_hz": 65, "required_snr_db": -29,␣
↪"noise_bandwidth_hz": 2500}, # 120-second mode A
            {"name": "Q65-300A", "bandwidth_hz": 65, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A

            {"name": "Q65-15B", "bandwidth_hz": 90, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500},  # Mode B - [ ] wider tone spacing
            {"name": "Q65-30B", "bandwidth_hz": 90, "required_snr_db": -27,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-60B", "bandwidth_hz": 90, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},
```

```
            {"name": "Q65-120B", "bandwidth_hz": 90, "required_snr_db": -29,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-300B", "bandwidth_hz": 90, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500},

            {"name": "Q65-15C", "bandwidth_hz": 180, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500}, # Mode C - [ ] even wider spacing
            {"name": "Q65-30C", "bandwidth_hz": 180, "required_snr_db": -27,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-60C", "bandwidth_hz": 180, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-120C", "bandwidth_hz": 180, "required_snr_db": -29,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-300C", "bandwidth_hz": 180, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500},

            {"name": "Q65-15D", "bandwidth_hz": 360, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500}, # Mode D
            {"name": "Q65-30D", "bandwidth_hz": 360, "required_snr_db": -27,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-60D", "bandwidth_hz": 360, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-120D", "bandwidth_hz": 360, "required_snr_db": -29,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-300D", "bandwidth_hz": 360, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500},

            {"name": "Q65-15E", "bandwidth_hz": 720, "required_snr_db": -26,␣
↪"noise_bandwidth_hz": 2500}, # Mode E - [ ] widest spacing
            {"name": "Q65-30E", "bandwidth_hz": 720, "required_snr_db": -27,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-60E", "bandwidth_hz": 720, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-120E", "bandwidth_hz": 720, "required_snr_db": -29,␣
↪"noise_bandwidth_hz": 2500},
            {"name": "Q65-300E", "bandwidth_hz": 720, "required_snr_db": -30,␣
↪"noise_bandwidth_hz": 2500},

            # FST4 Modes
            {"name": "FST4-15", "bandwidth_hz": 67, "required_snr_db": -21,␣
↪"noise_bandwidth_hz": 2500},  # 15-second mode A
            {"name": "FST4-30", "bandwidth_hz": 29, "required_snr_db": -24,␣
↪"noise_bandwidth_hz": 2500},  # 30-second mode A
            {"name": "FST4-60", "bandwidth_hz": 12, "required_snr_db": -28,␣
↪"noise_bandwidth_hz": 2500},  # 60-second mode A
```

```
            {"name": "FST4-120", "bandwidth_hz": 6, "required_snr_db": -31,␣
↪"noise_bandwidth_hz": 2500}, # 120-second mode A
            {"name": "FST4-300", "bandwidth_hz": 2, "required_snr_db": -35,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4-900", "bandwidth_hz": 0.7, "required_snr_db": -40,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4-1800", "bandwidth_hz": 0.4, "required_snr_db": -43,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4W-120", "bandwidth_hz": 6, "required_snr_db": -32,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4W-300", "bandwidth_hz": 2, "required_snr_db": -37,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4W-900", "bandwidth_hz": 0.7, "required_snr_db": -42,␣
↪"noise_bandwidth_hz": 2500}, # 300-second mode A
            {"name": "FST4W-1800", "bandwidth_hz": 0.4, "required_snr_db": -45,␣
↪"noise_bandwidth_hz": 2500} # 300-second mode A
        ]

    def add_mode(self, name, bandwidth_hz, required_snr_db, noise_bandwidth_hz):
        """
        Add a new mode to the evaluator.

        Parameters:
        name (str): Name of the mode
        bandwidth_hz (float): Bandwidth of the mode in Hz
        required_snr_db (float): Required SNR in dB for the mode to function
        noise_bandwidth_hz (float): over what bandwidth is the required_snr_db␣
↪given?
        """
        new_mode = {
            "name": name,
            "bandwidth_hz": bandwidth_hz,
            "required_snr_db": required_snr_db,
            "noise_bandwidth_hz": noise_bandwidth_hz
        }
        self.modes.append(new_mode)

    def evaluate_modes(self, doppler_spread_hz=None):
        """
        Evaluate suitability of amateur radio modes based on link CNR

        Parameters:
        cnr_db (float): Carrier-to-Noise Ratio in dB already calculated at␣
↪receiver_noise_bandwidth
        receiver_noise_bandwidth (float): Receiver noise bandwidth in Hz used␣
↪for the CNR calculation
```

```
        doppler_spread_hz (float, optional): Doppler spread in Hz, if we have it

        Returns:
        DataFrame: Modes with suitability assessment
        """
        results = []

        # Get the CNR in 1 Hz result from link budget calculator
        print(f"from the Link Budget calculator, our min cnr_db_1hz is␣
↪{min_results['cnr_db_1hz']}")

        for mode in self.modes:
            # Scale each mode SNR from noise bandwidth to 1Hz bandwidth
            # Some SNRs are taken really at 2500 Hz, and some are not.
            # We made a table column to handle this.
            # noise_bandwidth_hz is the "real" number to scale by.
            snr_db_1hz = mode["required_snr_db"] + 10 * np.
↪log10(mode["noise_bandwidth_hz"])

            # Apply Doppler spread penalty if applicable using a Gaussian model
            doppler_penalty_db = 0

            if doppler_spread_hz is not None and doppler_spread_hz > 0:
                # Gaussian model approach
                mode_bw = mode["bandwidth_hz"]

                # Calculate how much signal power falls outside the mode␣
↪bandwidth
                # For a Gaussian with standard deviation doppler_spread/2,
                # calculate power ratio inside vs. outside the mode bandwidth

                # Convert mode bandwidth to standard deviation units relative␣
↪to Doppler spread
                # The doppler_spread/2 is used as sigma in the Gaussian model
                sigma = doppler_spread_hz / 2

                if sigma > 0:  # Prevent division by zero
                    # Calculate normalized bandwidth (how many standard␣
↪deviations of the Gaussian)
                    norm_bw = mode_bw / (2 * sigma)

                    # Calculate error function to determine probability mass␣
↪within bandwidth
                    # Using error function approximation (accurate to ~1%)
                    import math
```

```
                    # Calculate power within the mode bandwidth (within +/- norm_bw standard deviations)
                    # erf(x) gives probability mass from -x to +x in standard normal distribution
                    if norm_bw < 4:  # If bandwidth is reasonably small compared to spread
                        # Calculate percentage of power within bandwidth
                        power_within = math.erf(norm_bw / math.sqrt(2))

                        # Convert to power ratio and then to dB
                        if power_within < 0.99:  # Avoid penalty for very small Doppler effects
                            power_ratio = 1.0 / power_within
                            doppler_penalty_db = 10 * math.log10(power_ratio)

                            # Cap the penalty at 20 dB as before
                            doppler_penalty_db = min(doppler_penalty_db, 20)
                    else:
                        # If bandwidth is much larger than spread, no significant penalty
                        doppler_penalty_db = 0

            # Calculate final effective SNR in 1Hz with Doppler penalty
            effective_snr_db_1hz = snr_db_1hz + doppler_penalty_db

            # Calculate margin between CNR in 1Hz and required SNR for mode in 1Hz
            margin_db = min_results['cnr_db_1hz'] - effective_snr_db_1hz

            # Determine reliability level
            if margin_db >= 10:
                reliability = "Excellent"
            elif margin_db >= 6:
                reliability = "Very Good"
            elif margin_db >= 3:
                reliability = "Good"
            elif margin_db >= 0:
                reliability = "Marginal"
            else:
                reliability = "Not Feasible"

            result = {
                "Mode": mode["name"],
                "Mode Bandwidth (Hz)": mode["bandwidth_hz"],
                "Mode SNR (dB)": mode["required_snr_db"],
                "Mode SNR 1hz (dB)": round(effective_snr_db_1hz, 1),
```

```
                "Margin (dB)": round(margin_db, 1),
                "Reliability": reliability,
                "Feasible": margin_db >= 0
            }

            # Add Doppler information if provided
            if doppler_spread_hz is not None:
                result["Doppler Spread (Hz)"] = doppler_spread_hz
                result["Doppler Penalty (dB)"] = round(doppler_penalty_db, 1)

            results.append(result)

        # Convert to DataFrame for easy display
        results_df = pd.DataFrame(results)

        # Sort by margin (highest first)
        results_df = results_df.sort_values(by="Margin (dB)", ascending=False)

        return results_df

    def filter_by_mode_type(self, results_df, mode_type):
        """
        Filter results by mode type (e.g., 'Q65', 'WSPR', etc.)

        Parameters:
        results_df (DataFrame): Results from evaluate_modes
        mode_type (str): Mode type to filter for

        Returns:
        DataFrame: Filtered results
        """
        return results_df[results_df['Mode'].str.contains(mode_type)]

    def get_feasible_modes(self, results_df):
        """
        Get only feasible modes from results

        Parameters:
        results_df (DataFrame): Results from evaluate_modes

        Returns:
        DataFrame: Only feasible modes
        """
        return results_df[results_df['Feasible'] == True]
```

```
# Example usage:
params = DSESLinkParameters()
evaluator = AmateurRadioModeEvaluator(params)
results = evaluator.evaluate_modes()

print('Results Without Considering Doppler Spread:\n')
print(results.to_string())

print(f"\n\n\n")

# # With Doppler:
print('Results Including Doppler Spread:\n')
# get extent of current Doppler Spread
doppler_spread_hz = DopplerSpreadCalculator.venus_doppler_spread()
# run evaluator with calculated Doppler Spread
results_with_doppler = evaluator.evaluate_modes(doppler_spread_hz)

print(results_with_doppler.to_string())

print(f"\n\n\n")

# # Filter for specific mode types:
print('Results Filtered for Q65 Modes:\n')
q65_modes = evaluator.filter_by_mode_type(results, 'Q65')
print(q65_modes.to_string())

print(f"\n\n\n")

# # Only feasible modes - no Doppler spread
print('Results Filtered for Feasible Modes Without Considering Doppler Spread:
 ↪\n')
only_feasible_results = evaluator.get_feasible_modes(results)
print(only_feasible_results.to_string())

print(f"\n\n\n")

# # Only feasible modes - with Doppler
print('Results Filtered for Feasible Modes Including Doppler Spread:\n')
only_feasible_results = evaluator.get_feasible_modes(results_with_doppler)
print(only_feasible_results.to_string())
```

```
from the Link Budget calculator, our min cnr_db_1hz is -8.652244288215712
Results Without Considering Doppler Spread:

          Mode  Mode Bandwidth (Hz)  Mode SNR (dB)  Mode SNR 1hz (dB)  Margin
(dB)   Reliability  Feasible
49  FST4W-1800                  0.4            -45              -11.0
2.4      Marginal      True
```

```
45   FST4-1800                0.4           -43              -9.0
0.4      Marginal      True
48   FST4W-900                0.7           -42              -8.0
-0.6  Not Feasible     False
44    FST4-900                0.7           -40              -6.0
-2.6  Not Feasible     False
47   FST4W-300                2.0           -37              -3.0
-5.6  Not Feasible     False
11    WSPR-120                6.0           -37              -3.0
-5.6  Not Feasible     False
43    FST4-300                2.0           -35              -1.0
-7.6  Not Feasible     False
9      WSPR-15                6.0           -32               2.0
-10.6  Not Feasible     False
46   FST4W-120                6.0           -32               2.0
-10.6  Not Feasible     False
42    FST4-120                6.0           -31               3.0
-11.6  Not Feasible     False
28    Q65-300C               180.0           -30               4.0
-12.6  Not Feasible     False
12     WSPR-LF                6.0           -30               4.0
-12.6  Not Feasible     False
38    Q65-300E               720.0           -30               4.0
-12.6  Not Feasible     False
33    Q65-300D               360.0           -30               4.0
-12.6  Not Feasible     False
23    Q65-300B                90.0           -30               4.0
-12.6  Not Feasible     False
18    Q65-300A                65.0           -30               4.0
-12.6  Not Feasible     False
22    Q65-120B                90.0           -29               5.0
-13.6  Not Feasible     False
27    Q65-120C               180.0           -29               5.0
-13.6  Not Feasible     False
32    Q65-120D               360.0           -29               5.0
-13.6  Not Feasible     False
37    Q65-120E               720.0           -29               5.0
-13.6  Not Feasible     False
17    Q65-120A                65.0           -29               5.0
-13.6  Not Feasible     False
21     Q65-60B                90.0           -28               6.0
-14.6  Not Feasible     False
36     Q65-60E               720.0           -28               6.0
-14.6  Not Feasible     False
16     Q65-60A                65.0           -28               6.0
-14.6  Not Feasible     False
26     Q65-60C               180.0           -28               6.0
-14.6  Not Feasible     False
```

```
41     FST4-60                12.0            -28                6.0
-14.6  Not Feasible     False
10      WSPR-2                 6.0            -28                6.0
-14.6  Not Feasible     False
31     Q65-60D              360.0            -28                6.0
-14.6  Not Feasible     False
35     Q65-30E              720.0            -27                7.0
-15.6  Not Feasible     False
30     Q65-30D              360.0            -27                7.0
-15.6  Not Feasible     False
25     Q65-30C              180.0            -27                7.0
-15.6  Not Feasible     False
20     Q65-30B               90.0            -27                7.0
-15.6  Not Feasible     False
15     Q65-30A               65.0            -27                7.0
-15.6  Not Feasible     False
29     Q65-15D              360.0            -26                8.0
-16.6  Not Feasible     False
13      WSPR-H                12.0            -26                8.0
-16.6  Not Feasible     False
24     Q65-15C              180.0            -26                8.0
-16.6  Not Feasible     False
34     Q65-15E              720.0            -26                8.0
-16.6  Not Feasible     False
19     Q65-15B               90.0            -26                8.0
-16.6  Not Feasible     False
14     Q65-15A               65.0            -26                8.0
-16.6  Not Feasible     False
2         JT65                2.7            -25                9.0
-17.6  Not Feasible     False
0          CW               250.0            -15                9.0
-17.6  Not Feasible     False
40     FST4-30               29.0            -24               10.0
-18.6  Not Feasible     False
39     FST4-15               67.0            -21               13.0
-21.6  Not Feasible     False
1          FT8               50.0            -20               14.0
-22.6  Not Feasible     False
8          JS8               30.0            -18               16.0
-24.6  Not Feasible     False
7          FT4               90.0            -17               17.0
-25.6  Not Feasible     False
6        PSK31               31.0              4               18.9
-27.6  Not Feasible     False
5         RTTY              250.0              5               29.0
-37.6  Not Feasible     False
3          SSB             2500.0              8               42.0
-50.6  Not Feasible     False
```

```
4           FM              12500.0             12                53.0
-61.6  Not Feasible     False
```

Results Including Doppler Spread:

We calculated a Doppler spread of: 31.13 Hz
from the Link Budget calculator, our min cnr_db_1hz is -8.652244288215712

	Mode	Mode Bandwidth (Hz)	Mode SNR (dB)	Mode SNR 1hz (dB)	Margin (dB)	Reliability	Feasible	Doppler Spread (Hz)	Doppler Penalty (dB)
28	Q65-300C	180.0	-30	4.0	-12.6	Not Feasible	False	31.125533	0.0
33	Q65-300D	360.0	-30	4.0	-12.6	Not Feasible	False	31.125533	0.0
38	Q65-300E	720.0	-30	4.0	-12.6	Not Feasible	False	31.125533	0.0
23	Q65-300B	90.0	-30	4.0	-12.6	Not Feasible	False	31.125533	0.0
18	Q65-300A	65.0	-30	4.1	-12.8	Not Feasible	False	31.125533	0.2
22	Q65-120B	90.0	-29	5.0	-13.6	Not Feasible	False	31.125533	0.0
32	Q65-120D	360.0	-29	5.0	-13.6	Not Feasible	False	31.125533	0.0
27	Q65-120C	180.0	-29	5.0	-13.6	Not Feasible	False	31.125533	0.0
37	Q65-120E	720.0	-29	5.0	-13.6	Not Feasible	False	31.125533	0.0
17	Q65-120A	65.0	-29	5.1	-13.8	Not Feasible	False	31.125533	0.2
11	WSPR-120	6.0	-37	5.1	-13.8	Not Feasible	False	31.125533	8.2
36	Q65-60E	720.0	-28	6.0	-14.6	Not Feasible	False	31.125533	0.0
26	Q65-60C	180.0	-28	6.0	-14.6	Not Feasible	False	31.125533	0.0
21	Q65-60B	90.0	-28	6.0	-14.6	Not Feasible	False	31.125533	0.0
31	Q65-60D	360.0	-28	6.0	-14.6	Not Feasible	False	31.125533	0.0
16	Q65-60A	65.0	-28	6.1	-14.8	Not Feasible	False	31.125533	0.2
35	Q65-30E	720.0	-27	7.0	-15.6	Not Feasible	False	31.125533	0.0
20	Q65-30B	90.0	-27	7.0	-15.6	Not Feasible	False	31.125533	0.0

```
25     Q65-30C               180.0           -27               7.0
-15.6  Not Feasible     False           31.125533                 0.0
30     Q65-30D               360.0           -27               7.0
-15.6  Not Feasible     False           31.125533                 0.0
15     Q65-30A                65.0           -27               7.1
-15.8  Not Feasible     False           31.125533                 0.2
19     Q65-15B                90.0           -26               8.0
-16.6  Not Feasible     False           31.125533                 0.0
29     Q65-15D               360.0           -26               8.0
-16.6  Not Feasible     False           31.125533                 0.0
24     Q65-15C               180.0           -26               8.0
-16.6  Not Feasible     False           31.125533                 0.0
34     Q65-15E               720.0           -26               8.0
-16.6  Not Feasible     False           31.125533                 0.0
14     Q65-15A                65.0           -26               8.1
-16.8  Not Feasible     False           31.125533                 0.2
49  FST4W-1800                 0.4           -45               8.9
-17.5  Not Feasible     False           31.125533                19.9
0           CW               250.0           -15               9.0
-17.6  Not Feasible     False           31.125533                 0.0
48   FST4W-900                 0.7           -42               9.4
-18.1  Not Feasible     False           31.125533                17.5
47   FST4W-300                 2.0           -37               9.9
-18.5  Not Feasible     False           31.125533                12.9
9      WSPR-15                 6.0           -32              10.1
-18.8  Not Feasible     False           31.125533                 8.2
46   FST4W-120                 6.0           -32              10.1
-18.8  Not Feasible     False           31.125533                 8.2
45   FST4-1800                 0.4           -43              10.9
-19.5  Not Feasible     False           31.125533                19.9
42    FST4-120                 6.0           -31              11.1
-19.8  Not Feasible     False           31.125533                 8.2
41     FST4-60                12.0           -28              11.2
-19.9  Not Feasible     False           31.125533                 5.2
44    FST4-900                 0.7           -40              11.4
-20.1  Not Feasible     False           31.125533                17.5
43    FST4-300                 2.0           -35              11.9
-20.5  Not Feasible     False           31.125533                12.9
40     FST4-30                29.0           -24              11.9
-20.5  Not Feasible     False           31.125533                 1.9
12     WSPR-LF                 6.0           -30              12.1
-20.8  Not Feasible     False           31.125533                 8.2
39     FST4-15                67.0           -21              13.1
-21.8  Not Feasible     False           31.125533                 0.1
13      WSPR-H                12.0           -26              13.2
-21.9  Not Feasible     False           31.125533                 5.2
10      WSPR-2                 6.0           -28              14.1
-22.8  Not Feasible     False           31.125533                 8.2
```

```
1          FT8                50.0           -20               14.5
-23.1  Not Feasible     False           31.125533                 0.5
7          FT4                90.0           -17               17.0
-25.6  Not Feasible     False           31.125533                 0.0
8          JS8                30.0           -18               17.8
-26.4  Not Feasible     False           31.125533                 1.8
6        PSK31                31.0             4               20.6
-29.2  Not Feasible     False           31.125533                 1.7
2         JT65                 2.7           -25               20.6
-29.2  Not Feasible     False           31.125533                11.6
5         RTTY               250.0             5               29.0
-37.6  Not Feasible     False           31.125533                 0.0
3          SSB              2500.0             8               42.0
-50.6  Not Feasible     False           31.125533                 0.0
4           FM             12500.0            12               53.0
-61.6  Not Feasible     False           31.125533                 0.0
```

Results Filtered for Q65 Modes:

```
        Mode  Mode Bandwidth (Hz)  Mode SNR (dB)  Mode SNR 1hz (dB)  Margin (dB)
Reliability  Feasible
28  Q65-300C               180.0            -30                4.0        -12.6
Not Feasible     False
38  Q65-300E               720.0            -30                4.0        -12.6
Not Feasible     False
33  Q65-300D               360.0            -30                4.0        -12.6
Not Feasible     False
23  Q65-300B                90.0            -30                4.0        -12.6
Not Feasible     False
18  Q65-300A                65.0            -30                4.0        -12.6
Not Feasible     False
22  Q65-120B                90.0            -29                5.0        -13.6
Not Feasible     False
27  Q65-120C               180.0            -29                5.0        -13.6
Not Feasible     False
32  Q65-120D               360.0            -29                5.0        -13.6
Not Feasible     False
37  Q65-120E               720.0            -29                5.0        -13.6
Not Feasible     False
17  Q65-120A                65.0            -29                5.0        -13.6
Not Feasible     False
21   Q65-60B                90.0            -28                6.0        -14.6
Not Feasible     False
36   Q65-60E               720.0            -28                6.0        -14.6
Not Feasible     False
```

```
16   Q65-60A                65.0            -28                6.0        -14.6
Not Feasible     False
26   Q65-60C               180.0            -28                6.0        -14.6
Not Feasible     False
31   Q65-60D               360.0            -28                6.0        -14.6
Not Feasible     False
35   Q65-30E               720.0            -27                7.0        -15.6
Not Feasible     False
30   Q65-30D               360.0            -27                7.0        -15.6
Not Feasible     False
25   Q65-30C               180.0            -27                7.0        -15.6
Not Feasible     False
20   Q65-30B                90.0            -27                7.0        -15.6
Not Feasible     False
15   Q65-30A                65.0            -27                7.0        -15.6
Not Feasible     False
29   Q65-15D               360.0            -26                8.0        -16.6
Not Feasible     False
24   Q65-15C               180.0            -26                8.0        -16.6
Not Feasible     False
34   Q65-15E               720.0            -26                8.0        -16.6
Not Feasible     False
19   Q65-15B                90.0            -26                8.0        -16.6
Not Feasible     False
14   Q65-15A                65.0            -26                8.0        -16.6
Not Feasible     False
```

Results Filtered for Feasible Modes Without Considering Doppler Spread:

```
          Mode  Mode Bandwidth (Hz)  Mode SNR (dB)  Mode SNR 1hz (dB)  Margin
(dB) Reliability  Feasible
49  FST4W-1800                  0.4            -45              -11.0
2.4    Marginal      True
45   FST4-1800                  0.4            -43               -9.0
0.4    Marginal      True
```

Results Filtered for Feasible Modes Including Doppler Spread:

```
Empty DataFrame
Columns: [Mode, Mode Bandwidth (Hz), Mode SNR (dB), Mode SNR 1hz (dB), Margin
(dB), Reliability, Feasible, Doppler Spread (Hz), Doppler Penalty (dB)]
Index: []
```

2.22 Zadoff-Chu Transmission Proposal

1. Do coherent integration in Zadoff-Chu segments however long we can given the worst cast Doppler rate of change. We assume that we're going to have to do batch processing with overlapping segments to ensure no signal is missed.
2. Do a Doppler compensation between segments.
3. Apply a sliding Doppler compensation during correlation processing
4. Combine resulting segments non-coherently until we know we can close the link.
5. Return detection result.

2.22.1 Minimum Integration Time with get_integration_time()

What is the minimum integration time given our Doppler rate of change?

We calculate the maximum doppler rate of change to find this number, and then use that as a fixed value for our Zadoff-Chu seqeunces.

Coherent integration time is calculated by how long it takes to exceed a 45 degree phase shift. This number is equal to square root of (0.25/Doppler rate of change). This equation is from "Fundamentals of Radar Signal Processing" by Mark Andrew Richards, 2005.

2.22.2 Get Number of Chips with get_number_chips()

Number of chips in our coherent integration is (chip rate * coherent integration time).

2.22.3 Find Processing Gain with get_processing_gain()

Processing gain: is 10 * log10(number of chips in our coherent integration)

This is a large number. How can this be so high? By coherently integrating, for example, 5 MHz of bandwidth over 1.34 seconds, we're concentrating the energy of 6.7 million independent measurements into a single detection decision.

Zadoff-Chu sequences have ideal auto-correlation properties, meaning they achieve the theoretical maximum processing gain. This is unlike other modulation schemes which suffer various losses due to inefficiencies that Zadoff-Chu sequences simply do not have. This is not without precedent or wildly made-up. NASA's deep space network uses comparable processing gain to communicate with distant spacecraft at extremely low bit rates. Radio astronomers routinely detect signals far below the noise floor using long integration times and correlation techniques like this. It's very likely that the radio astronomy people at Dwingeloo, DSES, and other amateur sites are already familiar with this technique and structure. This part of the link budget worksheet is an attempt to tune the technique for EVE from amateur sites with achievable configurations and parameters.

This proposal is designed around a calculated worst case Doppler rate limitation for Venus of -0.14 Hz/second, ensuring optimal coherent processing during the worst case channel condition, which happens to occur at inferior conjunction.

2.22.4 Is Processing Gain Enough? Compare to Bandwidth Expansion with get_bandwidth_expansion()

Let's see where we are with our CNR, assuming DSES numbers.

For example, CNR in 1 Hz bandwidth = -8.65 dB CNR in 5 MHz bandwidth = -8.65 dB - 10 * log(5×10^6) = -8.65 dB - 67 dB = -75.65 dB Processing gain for 5 MHz bandwidth = +68.26 dB We have a shortfall of ~7.35 dB.

We do a bandwidth expansion for our chip rate bandwidth with get_bandwidth_expansion() and compare to our previously calculated processing gain.

Our processing gain, calculated with the limitation of the Doppler rate shift on the sequence length, doesn't quite get us there. What can we do? There's three things we can look at. First, we can do multiple non-coherent integrations (e.g., 10 non-coherent integrations would add ~5 dB). We can try a slightly longer coherent integration time if Doppler allows. We designed for the worst case, at inferior conjunction. This is the best place to try EVE, so we are probably stuck with this limitation. Or, we can use some error correction coding for actual data transmission.

Using JPL's work as a model, let's do multiple non-coherent integrations. This is under our control, doesn't add as much complexity as adding error correction, and doesn't run the risk of blowing past a physical Doppler limit when we have the lowest path loss.

We use a dual-stage processing strategy. First, we perform coherent integration within the Doppler-limited window of (for example) 1.34 seconds across a 5 MHz bandwidth. Then, we combine multiple such integrations non-coherently to achieve positive SNR. This part is flexible, and can be used as Venus approaches or recedes. We just combine more sequences non-coherently.

For example, starting with a 1 Hz CNR of -8.65 dB, the full-bandwidth 5 MHz CNR is -75.65 dB. Coherent processing provides 68.3 dB gain, yielding -7.35 dB.

So, we need to non-coherently integrate some number of segments at inferior conjunction. Coherent processing maximizes gains within a Doppler rate of change limitation, and non-coherent integration extends processing beyond those constraints in a flexible way, depending on what our shortfall really is. Our calculations have a parameter for margin. The default is 0 dB. This parameter changes the threshold of detecion.

Calculate the number of non-coherent integrations needed with get_number_noncoherent_sequences_required At the receiver, we do a matched filtering using the known Zadoff-Chu sequence. The matched filter correlates the received signal with the expected sequence. The correlation output is examined for peaks that exceed a detection threshold. The time offset of the peak indicates the precise round-trip delay. The peak amplitude provides information about signal strength.

Non-coherent combination works with the power (magnitude squared) of the correlation outputs rather than the complex values. For each of the non-coherently integrated sequence correlations, we calculate magnitude squared. This destroys the phase information but preserves signal energy, which is all we want at this point. Align the correlation outputs based on expected delay progression. Sum up all the magnitude squared results. This summation increases SNR by approximately 5 * log(L) where L is the number of these sequences.

The distinction between coherent versus non-coherent combining is important when calculating processing gain from multiple observations.

Coherent combining at 10 * log10(N) is used when you can preserve both amplitude and phase information. Signals add linearly before detection. This is a "voltage addition". Power grows as N^2 where N is the number of samples, which results in 10 * log10(N) dB gain. Non-coherent

combining is 5 * log10(N). This is used when only signal power or amplitude can be preserved. In other words, when phase information is lost. Signals add after detection. This is a "power addition". Power grows as N (and not N^2) where N is the number of samples. This results in 5 * log10(N) dB gain.

Apply detection threshold to the combined result. I'm not entirely sure how to set the threshold, but correlates usually have a false-alarm detection rate and assumptions about the noise. This is the reason for having a margin parameter in the code below. The detected peaks in the non-coherent sum indicate successful reception. A detectable peak is what we are looking for.

We can extend out the non-coherent combination until we close the link, but does this have a limit? Non-coherent is (allegedly) more resilient to phase and Doppler than coherent integration. Can we assume it's immune, or do we need a factor that scales with number of non-coherent integrations?

Each coherent segment can be processed independently, allowing for parallel implementation. This reduces the burden on the hardware compared to a huge sequence. We think that the combination of coherent processing (optimized to Doppler rate of change constraint) and non-coherent integration (for extending beyond the shortfall we still have) provides a practical approach to close the link.

2.22.5 Visualization

Below is a visualization of an example of this proposal. You can see the bandwidth expansion, followed by a processing gain for the coherent integration. This is followed by gain from non-coherent integrations of the coherent integrations.

[90]:
```
display_svg('eve-signal-processing-diagram.svg', width=1800)
```

[90]:
```
<IPython.core.display.HTML object>
```

2.22.6 What about at other sites?

A different 1 Hz CNR means more or less non-coherent combinations. That's the only change.

2.22.7 Could this be wrong?

Please let me know of any errors or oversights, and I'll fix them.

[91]:
```
class ZadoffChuSequenceEvaluator:
    """
    A class to evaluate the suitability of Zadoff-Chu sequences based on link␣
↪parameters.
    """
    def __init__(self, params: DSESLinkParameters, chip_rate_hz: float,␣
↪cnr_db_1hz: float):
        self.params = params
        self.chip_rate_hz = chip_rate_hz
        self.cnr_db_1hz = cnr_db_1hz
        self.bandwidth_factor = 1 # in case we need to move from Time-Rate (1)␣
↪to Time-Bandwidth (other than 1)
```

```
    def get_bandwidth_expansion(self) -> float:
        """
        For our Zadoff-Chu implementation, we're assuming rectangular pulse␣
↪shaping,
        so the bandwidth (W) equals the chip rate (R). Therefore, the␣
↪time-bandwidth
        product (TW) equals the number of chips (TR).

        Processing gain = 10*log10(TW) = 10*log10(TR) = 10*log10(number of␣
↪chips)
        Bandwidth expansion = 1 Hz CNR - 10*log10(R) = 1 Hz CNR - 10*log10(chip␣
↪rate)

        If we want to account for pulse shaping, then the number of chips needs␣
↪to
        be multiplied by a factor such as 1.2 in both get_processing_gain and
        get_bandwidth_expansion() methods.

        Change self.bandwidth_factor from 1 to the desired factor in order to
        include the TR to TW transition, due to effects from pulse shaping.
        """
        expansion = self.cnr_db_1hz - 10 * np.log10(self.bandwidth_factor *␣
↪self.chip_rate_hz)
        #print(f"Full bandwidth SNR is {expansion:.2f} dB")
        return expansion

    def get_integration_time(self) -> float:
        # first set_observer_location, in order to get accurate worst case␣
↪Doppler rate of change from calculate_location_worst_case_doppler()
        DopplerCalculator.set_observer_location(params.latitude, params.
↪longitude, params.elevation, location_name='Our Radio Site')
        my_worst_case = DopplerCalculator.
↪calculate_location_worst_case_doppler(duration_days=584)
        #print(f"The worst case Doppler is␣
↪{my_worst_case['max_rate_per_second']:.6f} Hz/second")
        Tcoh = np.sqrt((1/4)/(np.abs(my_worst_case['max_rate_per_second'])))
        #print(f"Tcoh is {Tcoh:.2f}")
        return float(Tcoh)

    def get_number_chips(self) -> float:
        num_chips = self.get_integration_time() * self.bandwidth_factor * self.
↪chip_rate_hz
        #print(f"Number of chips is {num_chips:.2f}")
        return num_chips

    def get_processing_gain(self) -> float:
```

```
        """
        For our Zadoff-Chu implementation, we're assuming rectangular pulse␣
↪shaping,
        so the bandwidth (W) equals the chip rate (R). Therefore, the␣
↪time-bandwidth
        product (TW) equals the number of chips (TR).

        Processing gain = 10*log10(TW) = 10*log10(TR) = 10*log10(number of␣
↪chips)
        Bandwidth expansion = 1 Hz CNR - 10*log10(R) = 1 Hz CNR - 10*log10(chip␣
↪rate)

        If we want to account for pulse shaping, then the number of chips needs␣
↪to
        be multiplied by a factor such as 1.2 in both get_processing_gain and
        get_bandwidth_expansion() methods.

        Change self.bandwidth_factor from 1 to the desired factor in order to
        include the TR to TW transition, due to effects from pulse shaping.
        """
        processing_gain = 10 * np.log10(self.get_number_chips())
        #print(f"Processing gain is {processing_gain:.2f} dB for {num_chips:.
↪0f} chips")
        return processing_gain

    def get_number_noncoherent_sequences_required(self) -> int:
        gain = self.get_processing_gain() + self.get_bandwidth_expansion()
        print(f"The difference between Zadoff-Chu bandwidth expansion and the␣
↪processing gain is {gain:.2f} dB")
        margin = 10.0       # in practice, a margin is usually assumed - we␣
↪default to 10 dB
        print(f"We need this number to be at least {margin} dB.")
        if gain > margin: # we closed the link
            print("We closed the link with coherent integration. No further␣
↪non-coherent integration needed.")
            num_noncoh_seq = 0
            return num_noncoh_seq
        else:
            # we didn't close the link.
            # we need to non-coherently combine some number of sequences.
            # how many sequences? This loop erodes our shortfall to find out.
            count = 1
            gain = gain - margin # move our gain to where it's purely negative
            #print(f"gain is shifted by margin to be {gain}")
            gain = np.abs(gain) # turn it into a positive number to make it␣
↪easier to think about
```

```
            #print(f"gain is made positive and is {gain}")
            while gain > (5 * np.log10(count)):
                #print(f"gain is {gain:.2f} dB")
                #print(f"number of non-coherent integrations so far:{count}")
                count = count + 1
            print("We didn't close the link.")
            print(f"We need to non-coherently integrate {count} sequences to␣
 ↪produce {5 * np.log10(count):.2f} dB more gain.")
            print(f"If we do that, we should now have at least a {margin} dB␣
 ↪margin.")
            print(f"This is {count * ZadoffChuSequenceEvaluator.
 ↪get_integration_time():.1f} seconds of sequences.")
            return count



# Get the CNR in 1 Hz result from link budget calculator and set the chip rate
min_cnr_db_1hz = min_results['cnr_db_1hz']
print(f"from the Link Budget calculator, our min cnr_db_1hz is {min_cnr_db_1hz:.
 ↪3f} dB")
chip_rate = 1e6
print(f"Our chip rate is {chip_rate} Hz")


# Create Zadoff-Chu Sequence Evaluator instance
params = DSESLinkParameters()
ZadoffChuSequenceEvaluator = ZadoffChuSequenceEvaluator(params, chip_rate_hz =␣
 ↪chip_rate, cnr_db_1hz = min_cnr_db_1hz)

# test get_bandwidth_expansion
expansion = ZadoffChuSequenceEvaluator.get_bandwidth_expansion()
print(f"Full bandwidth SNR for our Zadoff-Chu sequence goes from the 1 Hz␣
 ↪bandwidth value of {min_cnr_db_1hz:.2f} dB to the chip rate bandwidth␣
 ↪expansion of {expansion:.2f} dB")


# test get_integration_time
Tcoh = ZadoffChuSequenceEvaluator.get_integration_time()
print(f"Our maximum integration time based on the maximum Doppler rate of␣
 ↪change is {Tcoh:.2f} seconds")


# test get_number_chips
num_chips = ZadoffChuSequenceEvaluator.get_number_chips()
print(f"The number of chips in our integration time is {num_chips:.2f}")
```

```
# test get_processing_gain
processing_gain = ZadoffChuSequenceEvaluator.get_processing_gain()
print(f"Processing gain for our coherent integration time is {processing_gain:.
 ↪2f} dB for {num_chips:.2f} chips")

# test the number of non-coherent integrations needed
num_noncoh_seq = ZadoffChuSequenceEvaluator.
 ↪get_number_noncoherent_sequences_required()
print(f"Recommended number of coherently integrated sequences recommended for␣
 ↪further non-coherent integration is: {num_noncoh_seq}, for a gain of {5 * np.
 ↪log10(num_noncoh_seq):.2f} dB")
```

```
from the Link Budget calculator, our min cnr_db_1hz is -8.652 dB
Our chip rate is 1000000.0 Hz
Full bandwidth SNR for our Zadoff-Chu sequence goes from the 1 Hz bandwidth
value of -8.65 dB to the chip rate bandwidth expansion of -68.65 dB
Our maximum integration time based on the maximum Doppler rate of change is 1.34
seconds
The number of chips in our integration time is 1339411.22
Processing gain for our coherent integration time is 61.27 dB for 1339411.22
chips
The difference between Zadoff-Chu bandwidth expansion and the processing gain is
-7.38 dB
We need this number to be at least 10.0 dB.
We didn't close the link.
We need to non-coherently integrate 2997 sequences to produce 17.38 dB more
gain.
If we do that, we should now have at least a 10.0 dB margin.
This is 4014.4 seconds of sequences.
Recommended number of coherently integrated sequences recommended for further
non-coherent integration is: 2997, for a gain of 17.38 dB
```

2.23

2.24 Temporal Spread Worksheet

Gary K6MG writes "One other item I didn't see addressed is the temporal spreading of the energy. Given a 12000Km diameter for venus the returned energy gets spread over a 12e6/3e8 = 40ms window. This dual temporal/spectral spreading presents a challenge to coherently summing the returned energy to detect a weak signal. OTOH it provides astronomers the mechanism to do planetary surface mapping by pixellating the surface into delay-doppler regions.

With a chip period of 2e-7 a total of 40e-3/2e-7=2e5 PN sequences need to be summed to collect all of the returned energy."

[92]:
```
print(f"diameter of Venus is: {2*calculator.venus_radius_km * 1000} m")
print(f"chip_rate_hz from Zadoff-Chu calculator is {ZadoffChuSequenceEvaluator.
 ↪chip_rate_hz} Hz")
```

```
print(f"speed of light is {calculator.params.c} meters per second")
temporal_spread = (2 * calculator.venus_radius_km * 1000)/calculator.params.c
print(f"Temporal spread is {temporal_spread} seconds")
chip_period = 1/ZadoffChuSequenceEvaluator.chip_rate_hz
print(f"With a chip period of {chip_period} seconds a total of␣
 ↪{(temporal_spread/chip_period):.2f} chips need to be summed to collect all␣
 ↪of the returned energy.")
num_chips = ZadoffChuSequenceEvaluator.get_number_chips()
print(f"Number of chips in our Zadoff-Chu sequence is {num_chips:.2f}")
```

```
diameter of Venus is: 12103600.0 m
chip_rate_hz from Zadoff-Chu calculator is 1000000.0 Hz
speed of light is 299792458 meters per second
Temporal spread is 0.04037326382640353 seconds
With a chip period of 1e-06 seconds a total of 40373.26 chips need to be summed
to collect all of the returned energy.
Number of chips in our Zadoff-Chu sequence is 1339512.11
```

[ ]: