

# Link\_Budget\_Modeling

March 3, 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.

### 1.1 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. Questions about this document? Please file an issue in the repository or write hello at operesearch dot institute.

## 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:  $555 \times 36$  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:  $1,824 \times 36$ Kb (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.

We set up dataclasses using the naming format SiteNameLinkParameters. We use numbers we have been given from sites such as Deep Space Exploration Society (DSES), who 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.

```
[133]: # Cell 1: Imports and class definitions
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from typing import Optional

@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_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 # Needs to be moved from worksheet to here
    tracking_error_deg: float = 0.01 # Needs to be moved from worksheet to here
    lna_noise_figure_db: float = 0.4 # from datasheet for the Kuhne MKU LNA 132
    ↪AH SMA
    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 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.

```
[134]: # 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 = * radius^2"""
        # 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)
    total_error_rad = np.sqrt(pointing_error_rad**2 + tracking_error_rad**2)
    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 (scroll
↳down)

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,
    '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
}

```

```
}
```

## 2.14 Link Budget Calculator Setup

We fetch all the LinkParameters 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 `cnr = calculator.calculate_link_budget(distance to Venus)`.

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

## 2.15 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.

```
[136]: # Cell 3: Calculate for Earth-Venus distances
min_distance_km = 38_000_000 # Minimum Earth-Venus distance
max_distance_km = 261_000_000 # Maximum Earth-Venus distance

# 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'])
    rx_powers.append(results['rx_power_dbw'])
```

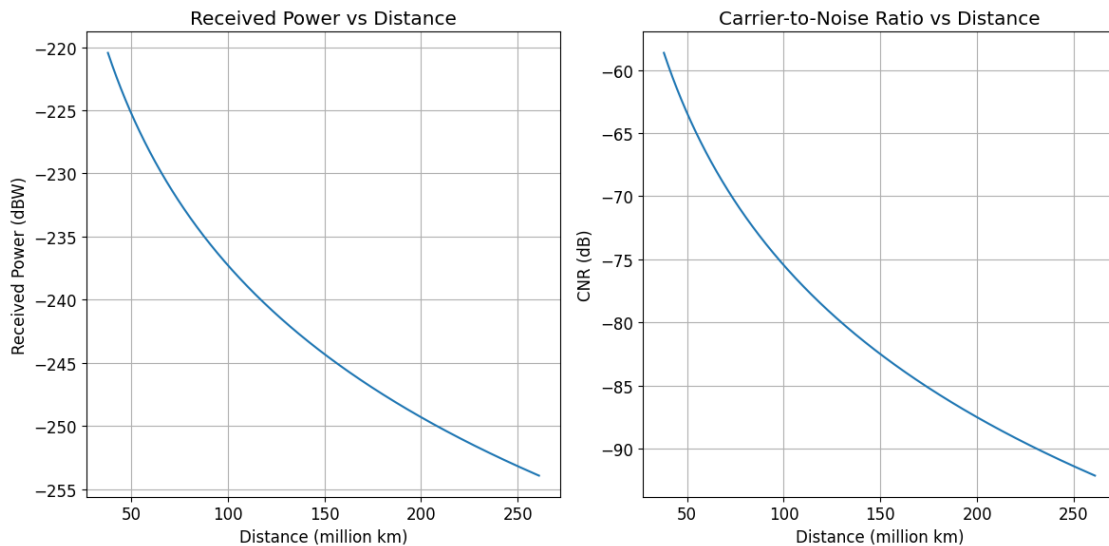
```
[137]: # Cell 4: Create plots
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(distances/1e6, rx_powers)
plt.grid(True)
plt.xlabel('Distance (million km)')
plt.ylabel('Received Power (dBW)')
plt.title('Received Power vs Distance')

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 vs Distance')
plt.tight_layout()
plt.show()

```



## 2.16 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.

```

[138]: # 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  $1.22 * \lambda / D$  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.

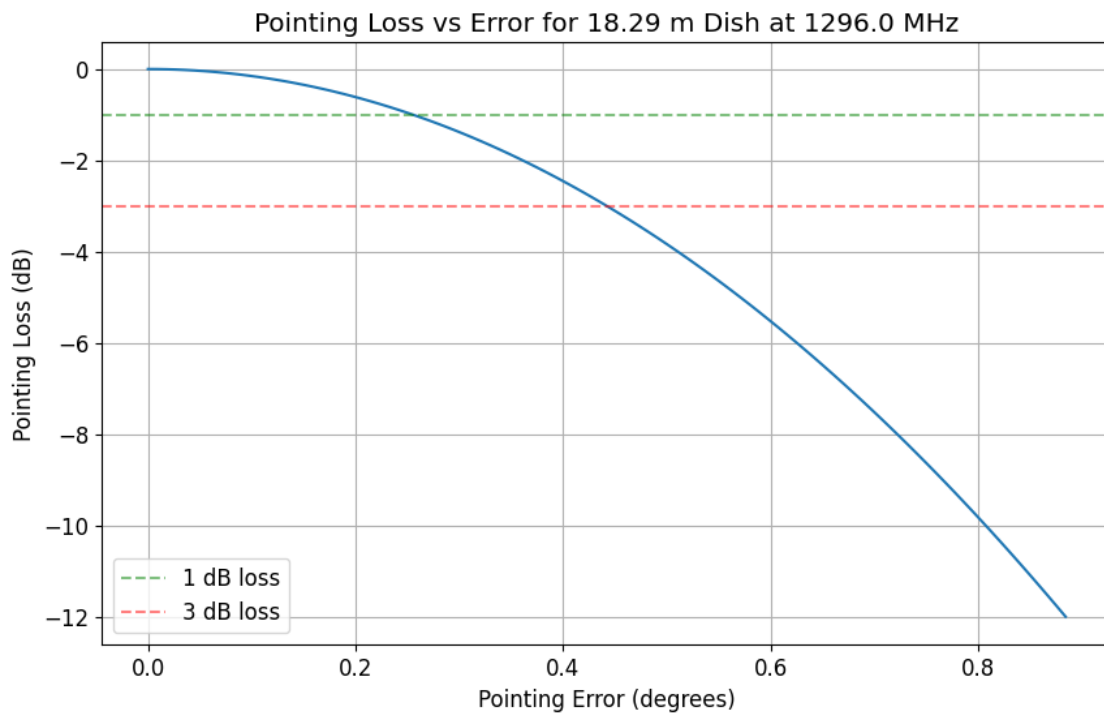
3. Working backwards from this formula:
  - For 1dB loss:  $-1 = -12 * (error/beamwidth)**2$
  - Solving for error:  $error = beamwidth/\sqrt{12}$   $beamwidth/5.66$
  - For 3dB loss:  $-3 = -12 * (error/beamwidth)**2$
  - Solving for error:  $error = beamwidth/2$

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

```
[139]: # 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()
```



## 2.17 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.

```
[140]: # 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}°")

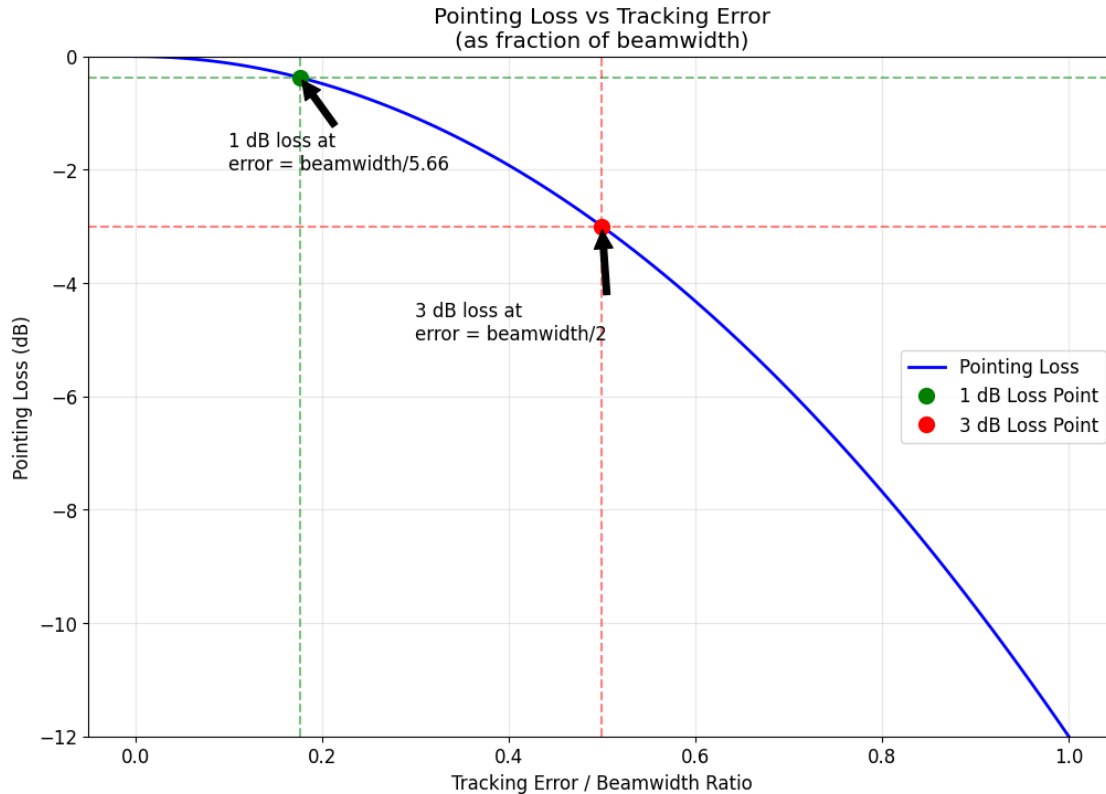
```

```

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°

## 2.18 Link Budget at Minimum and Maximum Distance

Here is the CNR at a particular receiver bandwidth for minimum and maximum distance from Earth to Venus.

```
[141]: # Cell 6: Print detailed results for minimum and maximum distances
min_results = calculator.calculate_link_budget(min_distance_km)
max_results = calculator.calculate_link_budget(max_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
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: 47.80
rx_power_dbw: -220.45
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
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: 47.80
rx_power_dbw: -253.93
noise_dbw: -161.80
cnr_db: -92.12
cnr_db_1hz: -42.12
```

[ ]:

## 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.

```
[142]: 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 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: 2025-03-04 04:57:26.660546+00:00

Current Earth-Venus:

Relative velocity: -7962.90 m/s

Doppler shift: 34423.54 Hz  
Received frequency: 1296.034424 MHz

Doppler shift rate of change (next 24 hours):  
Rate: -55.72 Hz/hour or -0.0155 Hz/second  
Starting Doppler: 34423.54 Hz  
Ending Doppler: 33086.14 Hz  
Total change: -1337.40 Hz

Calculating worst-case Doppler scenarios for a complete orbit cycle...

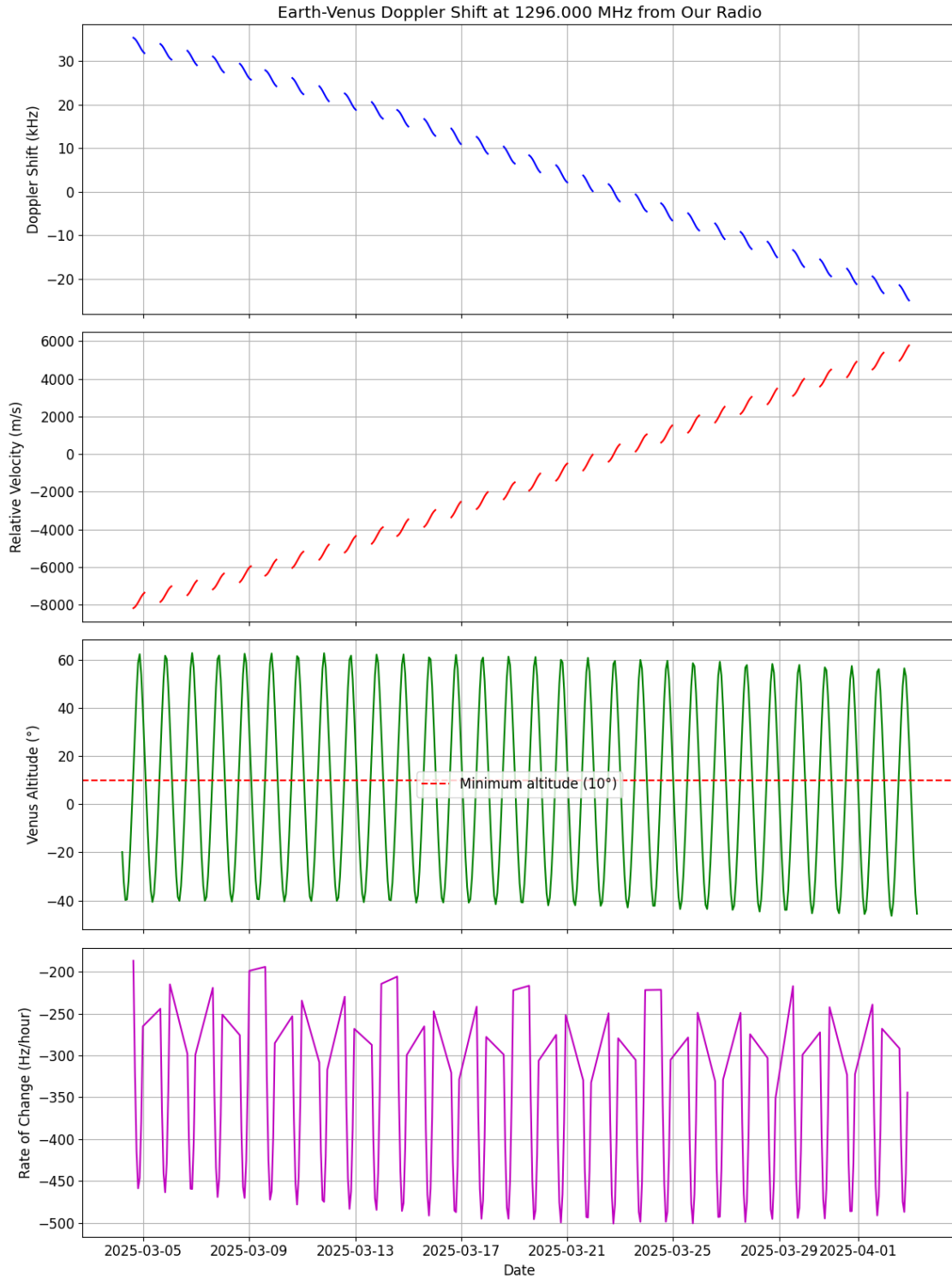
Worst-case Doppler shift:  
Maximum (positive) shift: 59965.13 Hz on 2026-08-04 10:57:26.668141+00:00  
Minimum (negative) shift: -60143.30 Hz on 2025-06-03 10:57:26.668141+00:00  
Total Doppler range: 120108.43 Hz

Worst-case Doppler rate of change:  
Maximum rate: -92.75 Hz/hour or -0.025763 Hz/second  
Occurs on: 2025-03-22 16:57:26.668141+00:00

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

Setting location to Our Radio (38.380833°, -103.156111°)  
Current Earth-Venus from Our Radio:  
Venus altitude: -19.87° / azimuth: 302.86°  
Relative velocity: -7674.32 m/s  
Doppler shift: 33176.00 Hz  
Received frequency: 1296.033176 MHz  
Venus is not visible (assuming min altitude of 10°)

Calculating worst-case scenarios for Our Radio over next 30 days...  
Maximum visible altitude: 62.72° on 2025-03-06 20:19:03.676237+00:00  
Venus visibility: 43.6% of time  
Maximum Doppler shift: 35404.77 Hz  
Minimum Doppler shift: -25171.89 Hz  
Maximum rate of change: -503.03 Hz/hour or -0.139732 Hz/second

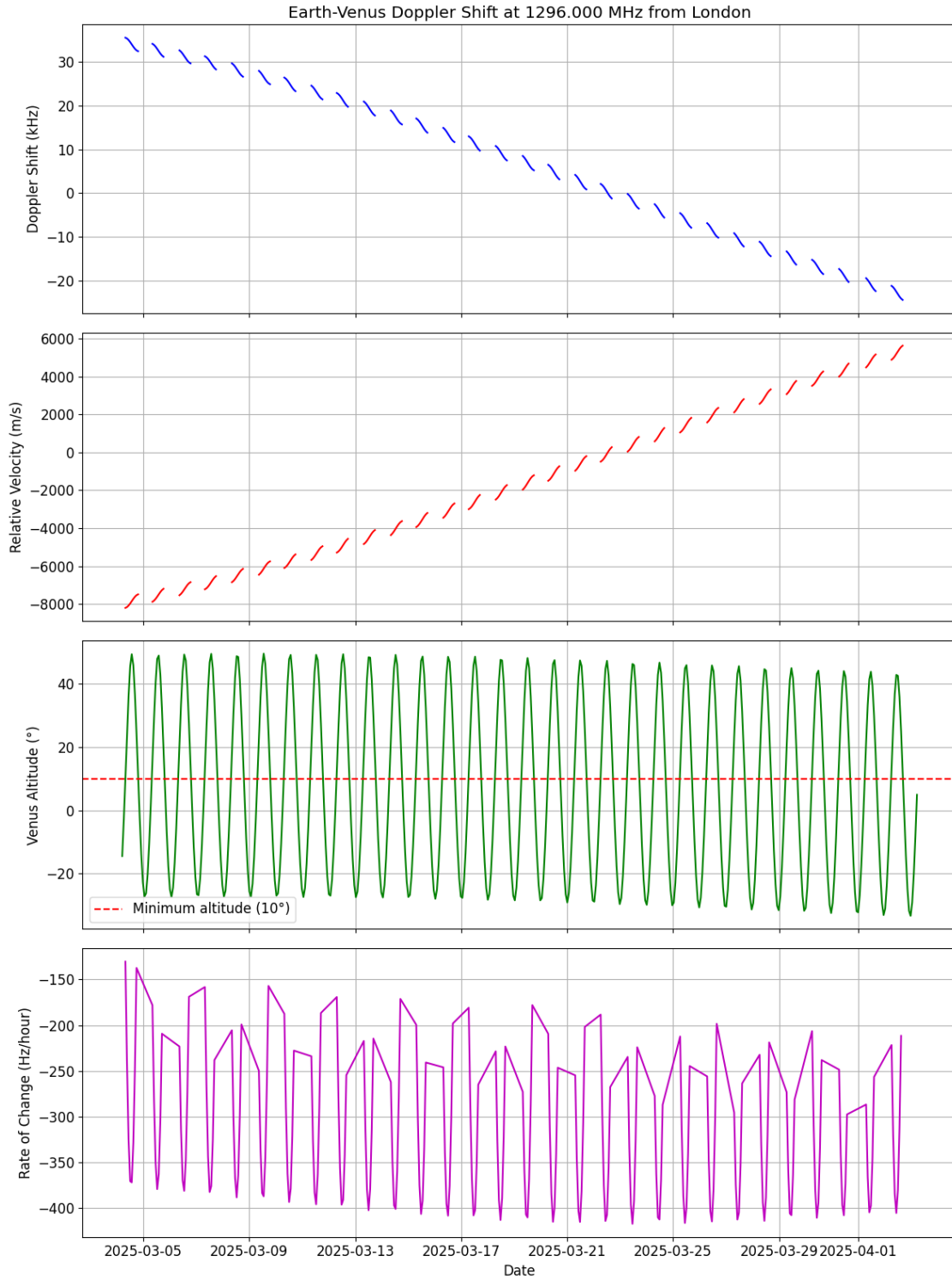


Setting location to London (51.5074°, -0.1278°)  
 Current Earth-Venus from London:

Venus altitude:  $-14.44^\circ$  / azimuth:  $50.40^\circ$   
Relative velocity:  $-8179.48$  m/s  
Doppler shift:  $35359.80$  Hz  
Received frequency:  $1296.035360$  MHz  
Venus is not visible (assuming min altitude of  $10^\circ$ )

Calculating worst-case scenarios for London over next 30 days...  
Maximum visible altitude:  $49.64^\circ$  on 2025-03-08 13:21:28.897669+00:00  
Venus visibility: 44.4% of time  
Maximum Doppler shift:  $35496.97$  Hz  
Minimum Doppler shift:  $-24392.16$  Hz  
Maximum rate of change:  $-418.21$  Hz/hour or  $-0.116169$  Hz/second

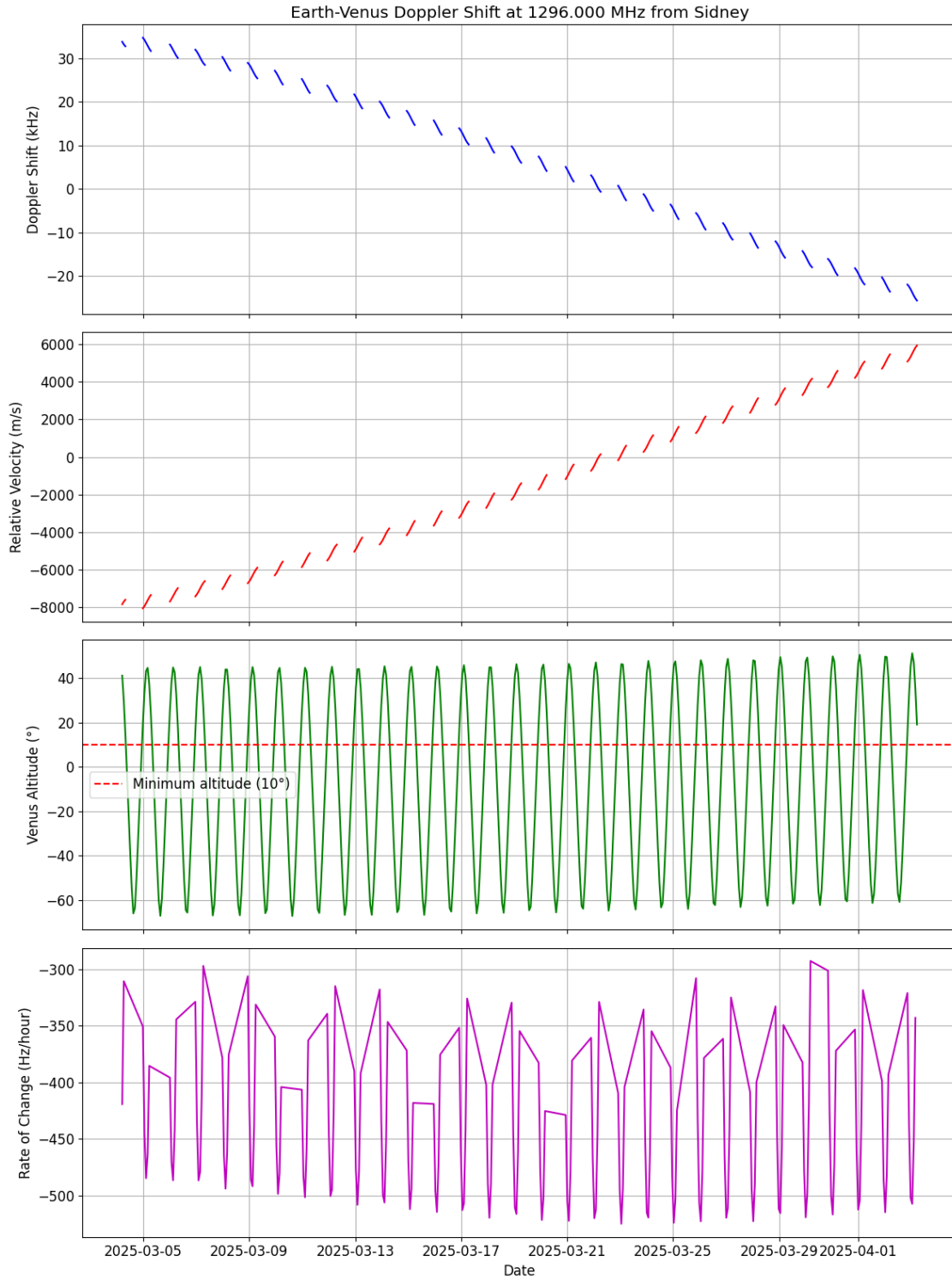




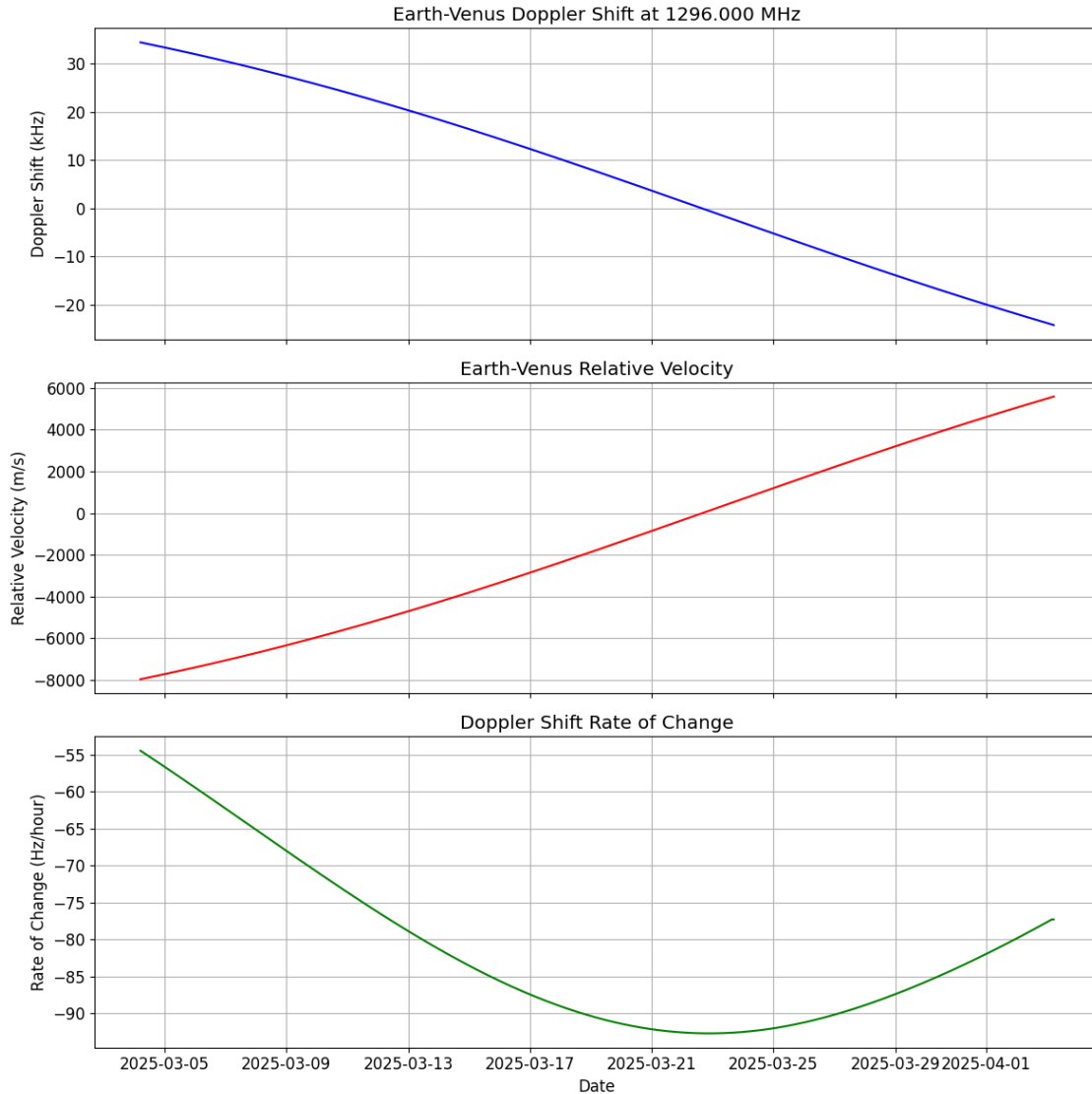
Setting location to Sidney (-33.8688°, 151.2093°)  
 Current Earth-Venus from Sidney:

Venus altitude: 41.06° / azimuth: 332.46°  
Relative velocity: -7827.88 m/s  
Doppler shift: 33839.84 Hz  
Received frequency: 1296.033840 MHz  
Venus is visible (assuming min altitude of 10°)

Calculating worst-case scenarios for Sidney over next 30 days...  
Maximum visible altitude: 51.14° on 2025-04-03 00:38:18.138986+00:00  
Venus visibility: 36.7% of time  
Maximum Doppler shift: 34951.31 Hz  
Minimum Doppler shift: -25669.07 Hz  
Maximum rate of change: -526.82 Hz/hour or -0.146339 Hz/second



Generating center-of-Earth Doppler curve from 2025-03-04 04:57:31.351869+00:00 to 2025-04-03 04:57:31.351869+00:00



## 2.20 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 a different bandwidth) is needed.

$$\text{SNR} = \text{CNR} \times (\text{BW\_CNR} / \text{BW\_SNR})$$

```
[143]: 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, cnr_db, receiver_noise_bandwidth,
↳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 = []

        # Scale our CNR result from link budget calculator to 1Hz
        bandwidth_factor_db = 10 * np.log10(receiver_noise_bandwidth / 1)
        cnr_db_1hz = cnr_db + bandwidth_factor_db

        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 it.
            # 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 (needs review because
↳Google says I can't do this)
            doppler_penalty_db = 0
            '''
            if doppler_spread_hz is not None and doppler_spread_hz > 0:
                # If Doppler spread exceeds mode bandwidth, apply penalty
                if doppler_spread_hz > mode["bandwidth_hz"]:
                    # Simple model: penalty proportional to how much Doppler
↳exceeds bandwidth.
                    # Claim:

```



```

# The Doppler spread penalty is being calculated as a ratio
↪between
# the actual Doppler spread (in Hz) and the mode's
↪bandwidth (also Hz).
# This ratio is dimensionless (Hz/Hz), so it doesn't need
↪scaling to 1Hz
# The 10*log10() turns this ratio into a straight up d
doppler_penalty_db = 10 * np.log10(doppler_spread_hz /
↪mode["bandwidth_hz"])
# Cap the penalty so things don't get out of hand here -
↪arbitrary 20 dB
doppler_penalty_db = min(doppler_penalty_db, 20)
#'''

# 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 = 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 - [ ] thank you Rose!
    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]

# candidate for some improvement but this is how we'll fetch CNR for now.
min_distance_km = 38_000_000 # Minimum Earth-Venus distance
max_distance_km = 261_000_000 # Maximum Earth-Venus distance
# Get our link budget CNR values from earlier calculations
min_results = calculator.calculate_link_budget(min_distance_km)
max_results = calculator.calculate_link_budget(max_distance_km)
min_cnr = min_results['cnr_db']
max_cnr = max_results['cnr_db']

```

```

# Example usage:
evaluator = AmateurRadioModeEvaluator(params)
results = evaluator.evaluate_modes(min_cnr, params.receiver_noise_bandwidth)

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

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

# # With Doppler:
print('Results Including Doppler Spread:\n')
results_with_doppler = evaluator.evaluate_modes(min_cnr, params.
↪ receiver_noise_bandwidth, doppler_spread_hz=30)
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())

```

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:

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

20	Q65-30B		90.0	-27	7.0
-15.6	Not Feasible	False		30	0.0
49	FST4W-1800		0.4	-45	7.7
-16.4	Not Feasible	False		30	18.8
19	Q65-15B		90.0	-26	8.0
-16.6	Not Feasible	False		30	0.0
34	Q65-15E		720.0	-26	8.0
-16.6	Not Feasible	False		30	0.0
24	Q65-15C		180.0	-26	8.0
-16.6	Not Feasible	False		30	0.0
14	Q65-15A		65.0	-26	8.0
-16.6	Not Feasible	False		30	0.0
29	Q65-15D		360.0	-26	8.0
-16.6	Not Feasible	False		30	0.0
48	FST4W-900		0.7	-42	8.3
-16.9	Not Feasible	False		30	16.3
47	FST4W-300		2.0	-37	8.7
-17.4	Not Feasible	False		30	11.8
46	FST4W-120		6.0	-32	9.0
-17.6	Not Feasible	False		30	7.0
0	CW		250.0	-15	9.0
-17.6	Not Feasible	False		30	0.0
9	WSPR-15		6.0	-32	9.0
-17.6	Not Feasible	False		30	7.0
45	FST4-1800		0.4	-43	9.7
-18.4	Not Feasible	False		30	18.8
41	FST4-60		12.0	-28	10.0
-18.6	Not Feasible	False		30	4.0
42	FST4-120		6.0	-31	10.0
-18.6	Not Feasible	False		30	7.0
40	FST4-30		29.0	-24	10.1
-18.8	Not Feasible	False		30	0.1
44	FST4-900		0.7	-40	10.3
-18.9	Not Feasible	False		30	16.3
43	FST4-300		2.0	-35	10.7
-19.4	Not Feasible	False		30	11.8
12	WSPR-LF		6.0	-30	11.0
-19.6	Not Feasible	False		30	7.0
13	WSPR-H		12.0	-26	12.0
-20.6	Not Feasible	False		30	4.0
10	WSPR-2		6.0	-28	13.0
-21.6	Not Feasible	False		30	7.0
39	FST4-15		67.0	-21	13.0
-21.6	Not Feasible	False		30	0.0
1	FT8		50.0	-20	14.0
-22.6	Not Feasible	False		30	0.0
8	JS8		30.0	-18	16.0
-24.6	Not Feasible	False		30	0.0

7	FT4		90.0	-17	17.0
-25.6	Not Feasible	False		30	0.0
6	PSK31		31.0	4	18.9
-27.6	Not Feasible	False		30	0.0
2	JT65		2.7	-25	19.4
-28.1	Not Feasible	False		30	10.5
5	RTTY		250.0	5	29.0
-37.6	Not Feasible	False		30	0.0
3	SSB		2500.0	8	42.0
-50.6	Not Feasible	False		30	0.0
4	FM		12500.0	12	53.0
-61.6	Not Feasible	False		30	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: []

## 2.21 Zadoff-Chu Transmission Proposal

1. Do coherent integration in Zadoff-Chu segments of ~1.3 seconds. For a 5 MHz chip rate and 1.3 seconds, the sequences have a length  $N$  of at least 6.5 million. We should probably 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 segments non-coherently until we know we can close the link.
5. Return result.

Number of chips in our coherent integration:  $5 \times 10^6 \times 1.34 = 6.7$  million

Processing gain:  $10 * \log_{10}(6.7 \times 10^6) = 68.3$  dB

How can this be so high? By coherently integrating 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.

The system is designed around the Doppler rate limitation of -0.14 Hz/second, ensuring optimal coherent processing. Let's see where we are with our CNR.

CNR in 1 Hz bandwidth = -8.65 dB  
CNR in 5 MHz bandwidth = -8.65 dB -  $10 * \log(5 \times 10^6) = -8.65$  dB - 67 dB = -75.65 dB

Our processing gain, 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 ~10 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 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, giving us a really nice test of our sites.

Starting with a 1 Hz CNR of -8.65 dB, the full-bandwidth CNR is -75.65 dB. Coherent processing provides 68.3 dB gain, yielding -7.35 dB. Non-coherent integration of 8 segments adds 9 dB ( $10 * \log(8) = 9$  dB), resulting in a final SNR of +1.65 dB.

So, we need to process at least 8 segments at inferior conjunction. I believe that this approach is mathematically sound and follows established radar and communications principles. Coherent processing maximizes gains within a Doppler rate of change limitation, and non-coherent integration extends processing beyond those constraints in a flexible way that can “follow” Venus for at least a little bit farther along the orbit.

At the receiver, we’d 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 8 (or more) sequence correlations, 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  $10 * \log(L)$  where L is the number of these sequences.

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. The detected peaks in the non-coherent sum indicate successful reception. That’s what we’re looking for.

We can extend out the non-coherent combination by quite a bit, but it does have a limit. It’s more resilient to phase and Doppler than coherent integration, but isn’t completely immune.

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.

Let’s run the numbers for 1 MHz as well. This lowers the chip rate down even further. Starting with a 1 Hz CNR of -8.65 dB, the full 1 MHz bandwidth CNR is  $-8.65 - 10 * \log(1 \times 10^6) = -68.65$  dB. Note this is less than the 5 MHz bandwidth CNR from above of -75.65 dB. After coherent integration:  $-68.65 +$  processing gain of 61.3 dB ( $10 \log(110^6) = -7.35$  dB (same as before!).

With 8 non-coherent integrations:  $-7.35 \text{ dB} + 9 \text{ dB} = +1.65 \text{ dB}$

Why is it the same as before? The 1 MHz bandwidth has 7 dB less noise power than 5 MHz. But it also provides 7 dB less coherent processing gain. These effects cancel out exactly, leaving the same post-processing SNR. So, either 1 MHz or 5 MHz should work.

### 2.21.1 At Maximum Range

Stretch goal: can we work Venus with this technique at superior conjunction?

New calculation. We have a -42 dB 1 Hz CNR (again, at DSES): Bandwidth expansion to 5 MHz:  $-42 - 10 * \log(5 \times 10^6) = -109$  dB. After coherent integration (1.34s):  $-109 + 68.3 = -40.7$  dB Required non-coherent integration gain:  $40.7 +$  margin (e.g., 3 dB) = 43.7 dB

For non-coherent integration gain of 43.7 dB, the number of sequences needed =  $10^{(43.7/10)}$  23,440 sequences. Whew, sounds like a lot.

At 1.34 seconds per sequence, this would require a total integration time of about 31,410 seconds or 8.7 hours.

### 2.21.2 What about at other sites?

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

### 2.21.3 Could this be wrong?

I've done my dead-level best and called in several friends to help, read several papers, and cracked several books. I've asked my new friend Claude.ai for help in searching for sources and confirming my understanding of spread spectrum. I asked Claude to make the SVG drawing for me, which he did in a lengthy conversation where I described what I wanted drawn. This design does need to be reviewed. Please let me know of any errors or oversights, and I'll fix them.

```
[144]: #[EVE Signal Processing Diagram](eve-signal-processing-diagram.svg) wouldn't
       ↳render to PDF, so stashing markdown command here for now.

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

    def get_bandwidth_expansion(self) -> float:
        expansion = self.cnr_db_1hz - 10 * np.log10(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((np.pi/4)/(np.pi * 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.chip_rate_hz
    #print(f"Number of chips is {num_chips:.2f}")
    return num_chips

def get_processing_gain(self) -> float:
    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 = 3.0 # 3 dB margin - might need to be larger?
    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 > (10 * 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 {10 * np.log10(count):.2f} dB more gain.")
        print("If we do that, we should now have at least a 3 dB margin.")
        return count

```

*# candidate for some improvement but this is how we'll fetch CNR for now.*

```

min_distance_km = 38_000_000 # Minimum Earth-Venus distance
max_distance_km = 261_000_000 # Maximum Earth-Venus distance
# Get our link budget CNR values from earlier calculations
min_results = calculator.calculate_link_budget(min_distance_km)
max_results = calculator.calculate_link_budget(max_distance_km)
min_cnr_db_1hz = min_results['cnr_db_1hz']
max_cnr_db_1hz = max_results['cnr_db_1hz']

# Create Zadoff-Chu Sequence Evaluator instance
params = DSESLinkParameters()
ZadoffChuSequenceEvaluator = ZadoffChuSequenceEvaluator(params, chip_rate_hz = 5e6, 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 {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 {10 * np.log10(num_noncoh_seq):.2f} dB")

```

The difference between Zadoff-Chu bandwidth expansion and the processing gain is -7.38 dB

We need this number to be at least 3.0 dB.

We didn't close the link.

We need to non-coherently integrate 11 sequences to produce 10.41 dB more gain.  
If we do that, we should now have at least a 3 dB margin.

## 2.22 System Noise Temperature Worksheet

Trying to recreate the 70 k Tsys from the DSES worksheet. Numbers don't match so we should figure out why.

Some notes on trying to move to a higher frequency, and whether the surface RMS estimation of 0.3mm could "hold":

At 1296 MHz:

Surface efficiency 98%

Very good performance, minimal scatter loss.

At 2450 MHz:

Surface efficiency 98%

Does not seem to present a problem.

Also,  $f_{\max} = c / (16 \times \text{RMS})$  as a guideline, which gives a number larger than 2304 MHz.

```
[145]: from dataclasses import dataclass
from typing import Optional
import numpy as np

@dataclass
class SkyNoiseParams:
    """Parameters affecting sky noise temperature"""
    tx_frequency_mhz: float
    elevation_angle_deg: float
    atmospheric_conditions: str = 'clear' # 'clear', 'cloudy', 'rain'

@dataclass
class AntennaParams:
    """Physical parameters of the antenna"""
    main_beam_efficiency: float
    spillover_efficiency: float
    surface_rms_mm: float
    diameter_m: float

class SystemNoiseCalculator:
    def __init__(self, antenna_params: AntennaParams):
        self.antenna = antenna_params

    def calculate_sky_noise(self, sky_params: SkyNoiseParams) -> float:
        """
        Calculate sky noise temperature based on frequency and elevation angle
```

```

Using simplified atmospheric model - [ ] Jan King used the ITU model
"""
# Convert elevation angle to radians
elev_rad = np.radians(sky_params.elevation_angle_deg)

# Basic atmospheric attenuation model (simplified)
# We might want to try to incorporate more complex models like the one
# Jan King uses in his excel spreadsheet (ITU).
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 (simplified model!)
freq_factor = 0.1 * (sky_params.tx_frequency_mhz / 1000) / 10.0 #
↳ Convert MHz to GHz for calculation

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

weather_multiplier = weather_factors.get(sky_params.
↳ atmospheric_conditions, 1.0)

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

def calculate_spillover_noise(self, ground_temp: float = 290.0) -> float:
    """Calculate spillover noise contribution"""
    return ground_temp * (1 - self.antenna.spillover_efficiency)

def calculate_scatter_noise(self, tx_frequency_mhz: float) -> float:
    """Calculate scattering noise due to surface imperfections"""
    wavelength_mm = 300000 / tx_frequency_mhz # Convert MHz to wavelength
↳ in mm
    surface_efficiency = np.exp(-(4 * np.pi * self.antenna.surface_rms_mm /
↳ wavelength_mm) ** 2) # surface efficiency, Ruze equation
    #print(f"{surface_efficiency}")
    return 290 * (1 - surface_efficiency)

def calculate_system_noise(self,
    sky_params: SkyNoiseParams,
    receiver_temp: float,
    ground_temp: float = 290.0) -> dict:

```



```

"""
Calculate total system noise temperature and its components

Args:
    sky_params: Sky noise parameters
    receiver_temp: Receiver noise temperature in Kelvin
    ground_temp: Ground temperature in Kelvin (default 290K)

Returns:
    Dictionary containing total and component temperatures
"""
# Calculate individual components
sky_noise = self.calculate_sky_noise(sky_params)
spillover_noise = self.calculate_spillover_noise(ground_temp)
scatter_noise = self.calculate_scatter_noise(sky_params,
↳tx_frequency_mhz)

# Calculate antenna temperature
t_ant = (self.antenna.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
}

# Example usage
if __name__ == "__main__":
    # Define antenna parameters
    antenna = AntennaParams(
        main_beam_efficiency=0.69, # Hoping this is the same as aperture
↳efficiency - [ ] if not we need a number
        spillover_efficiency=0.95, # 60-80 is typical for large dishes so I
↳picked 70% at first. This is the percent
        # of what is transmitted that is actually
↳reflected. Feed pattern doesn't perfectly
        # match the dish surface. Thomas
↳recommended 95% to reflect the better performance of
        # space oriented dishes like the ones we're
↳dealing with

```

```

    surface_rms_mm=0.3, # root mean square (RMS) deviation of the actual
↳dish surface from an ideal parabolic shape.
        # Professional radio telescopes 0.1-0.3mm "A
↳typical error budget [43] for the surface error
        # budget of a 34-m antenna is 0.3-mm
↳root-sum-square (rss)." -Large Antennas of the Deep Space Network
        # by William A. Imbriale
        # Cheaper mesh dishes are 2mm or more
    diameter_m=18.2 # Physical dimensions of the dish.
)

# Create calculator instance
NoiseCalculator = SystemNoiseCalculator(antenna)

# Define sky parameters
sky = SkyNoiseParams(
    tx_frequency_mhz=1296.0,
    #tx_frequency_mhz=2450.0,
    elevation_angle_deg=45.0, # venus can be up to 40-60 degrees elevation
↳- [ ] this can be a live number.
    atmospheric_conditions='clear'
)

# Calculate system noise
results = NoiseCalculator.calculate_system_noise(
    sky_params=sky,
    receiver_temp=28.0 # 0.4 dB NF Kuhne MKU LNA 132 AH SMA - [ ] fairly
↳confident on this temperature
)

# Print results
for key, value in results.items():
    print(f"{key}: {value:.1f} K")

```

```

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

```

[ ]:

[ ]: