Project 3: Simulating Radio Signal Modulation via Markov Chain Monte Carlo Sampling

Nicholas Mizero Rubayiza and Jasmine Watt

November 2024

1 Introduction

1.1 Description of the simulation

This project aims to model FM radio signals. An FM radio signal is generated with unknown modulation parameters and an attempt is made to model the signal. Using a Bayesian approach, we will model the probability distribution of the proposed parameters of our model, given the signal data. Using Markov chains, we can sample from this distribution and compare our parameters (and entire model) to their respective probability distributions - giving a basis for evaluating the effectiveness of our model in fitting the signal data.

1.2 Choice of parameters

The choice of parameters was based on visual inspection of the data. The parameters include a proposed amplitude, frequency and phase. The assumption made was that the transmitted signal experienced no significant decay within the period of transmission.

2 Code Validation

Each chain is to eventually be able to sample randomly from our posterior distribution. Once a chain has converged to the distribution, it is said to have been burnt it. There are ways to test whether this hasn't happened yet.

If some chain has indeed converged to our posterior distribution, then it should have statistical properties similar to the distribution and to any other chain that has burnt in. Therefore, comparing the mean value of the distribution of chains that have burnt in should show similar values within some statistical error margin.

Following this logic, if you plot any statistical property as a function of iterations of the chain, this property should eventually converge to some constant (within some statistical margin of error) that reflects the property of the posterior distribution as the chain burns in.

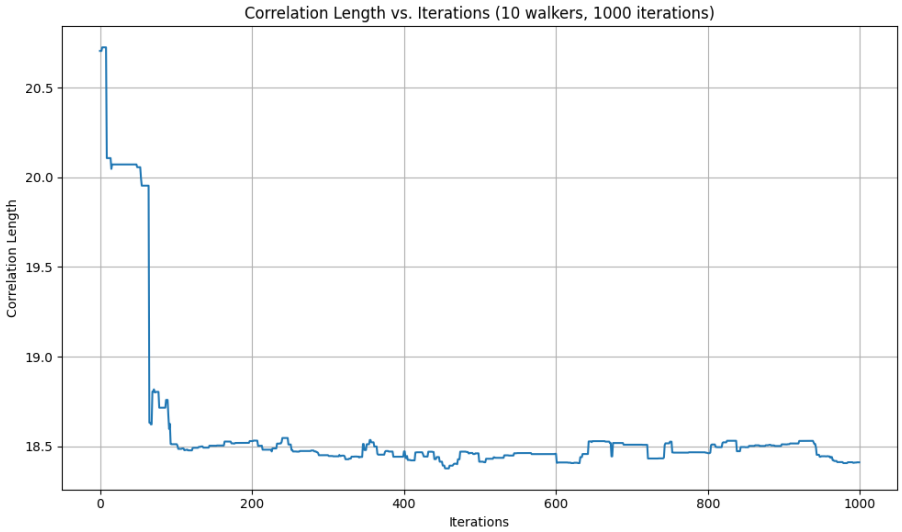

Figure 1: Autocorrelation Length for 1000 iterations. Burn in seems to have happened after 200 iterations

The ACL (autocorrelation length) is a measure of how independent (or related) each sample is from the previous one. Ideally, sampling from a distribution should be random and thus the correlation between any two samples should be zero. Practically, the walkers do retain some memory of its previous positions so the ACL is not zero. However, if a chain has burnt in, a plot of the ACL as a function of iterations should reveal a trend downwards that eventually converges to some constant (again within some statistical error margin). The downwards trend speaks to the randomness of the chains sampling and the constant value speaks to the convergence of the distribution to our posterior one.

Finally, it is correct to say that with enough burnt in chains, sampling once from each chain should yield the same result as sampling multiple times from a single distribution. Thus, sampling across burnt in chains should reveal the shape of our posterior distribution and this shape should be preserved for any number of iterations after burn in.

3 Error Analysis

To get a measure of our statistical error, we picked a chain that passed all validation tests and repeatedly run calculations for the ACL after a large number of iterations. This should yield a distribution reflective of our statistical error. It is expected to be Gaussian with some spread about the true ACL.

As was said before, a plot of a statistical property, like the mean, should

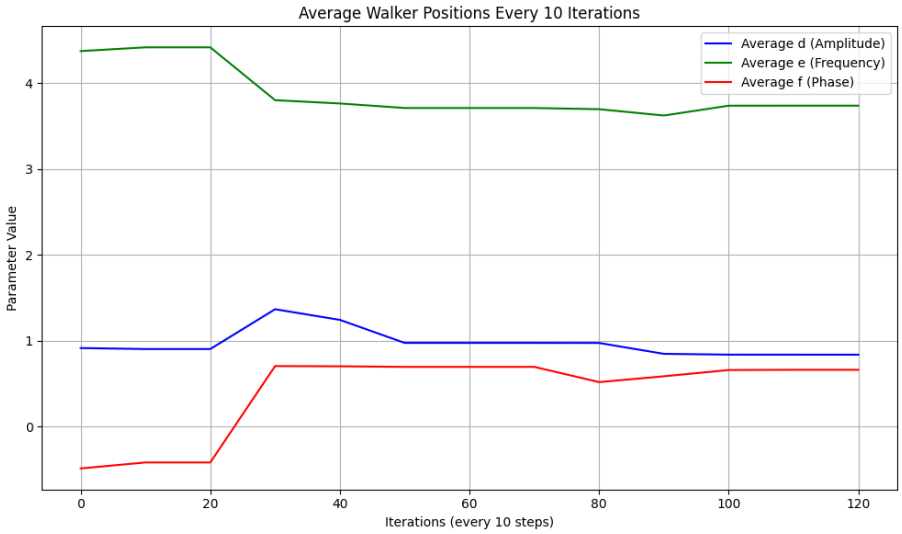

Figure 2: The plot of the averaged parameters every 10 iterations. This plot shows that the parameters approaching a certain value after large enough (meaning probably larger than burn-in number of iterations) iterations) to the true value of the parameters.

eventually yield some constant reflective of the distribution of the posterior. However, the above graph suggests that something may have gone wrong in the process. This may very well mean that one of the chains did not yet burn it, or the process did indeed suffer a larger than expected error margin, or equally likely, the parameter is yet still approaching its final value because we haven't a way to test if the parameter value is accurate. For instance, the frequency parameter is stuck in a multiple of the true fundamental frequency.

3.1 Package and its purpose

The purpose of the package is to model radio signals through Monte Carlo Markov Chain Modeling, generate plots that shows proper burn-in length (ACL), and show the correlation between our guess and the data input.

We built a package in the following structure.

```
fm-mcmc/
        main.py
        fm-mcmc
              __init__.py
              cor_len.py
              mcmc.py
              signal.py
              plotting.py
```

```
        README.md
        setup.py
        requirements.txt
```

```python
#!/usr/bin/env python3

import numpy as np
from fm_mcmc.signal import generate_fm_signal, generate_audio_signal
from fm_mcmc.mcmc import run_mcmc
from fm_mcmc.plotting import plot_correlation_length,
    plot_walkers_paths, plot_avg_positions
from fm_mcmc.cor_len import correlation_length

def main():
    # Constants for the FM signal
    fs = 10e6     # Sampling rate (Hz)
    fc = 100e6    # Carrier frequency (Hz)
    fm = 1e3      # Modulating frequency (Hz)
    duration = 1  # Duration of the signal in seconds

    # Time vector for the FM signal
    t = np.arange(0, duration, 1/fs)

    # Generate the true FM signal using the true parameters
    d_true = 1.0  # True scaling factor for the sine wave (Amplitude)
    e_true = 2 * np.pi * fm # True frequency of the sine wave
        (Modulating frequency)
    f_true = 0.0  # True phase offset

    audio_signal = generate_audio_signal(t)
    fm_signal = generate_fm_signal(d_true, e_true, f_true, t)

    # User prompt for number of walkers and iterations
    n_walkers = int(input("Enter the number of walkers: "))
    n_iterations = int(input("Enter the number of iterations: "))

    # Run MCMC simulation
    all_walkers_positions = run_mcmc(n_walkers, n_iterations, fm_signal,
        t)

    # Calculate correlation lengths
    correlation_lengths = []
    for i in range(n_iterations):
        walkers_pos_at_iter = all_walkers_positions[i, :, :].reshape(-1,
            3)
        correlation_lengths.append(correlation_length(walkers_pos_at_iter))

    # Plotting
```

```
    plot_correlation_length(correlation_lengths, n_walkers, n_iterations)
    plot_walkers_paths(all_walkers_positions, d_true, e_true, f_true,
        n_walkers, n_iterations)

    # Calculate and plot average walker positions every 10 iterations
    avg_positions = []
    for i in range(0, n_iterations, 10):
        walkers_at_iter = all_walkers_positions[i, :, :]
        avg_d = np.mean(walkers_at_iter[:, 0])
        avg_e = np.mean(walkers_at_iter[:, 1])
        avg_f = np.mean(walkers_at_iter[:, 2])
        avg_positions.append((avg_d, avg_e, avg_f))

    avg_positions = np.array(avg_positions)
    plot_avg_positions(avg_positions, n_walkers, n_iterations)

if __name__ == "__main__":
    main()
```

Under the package fm-mcmc, we have other files that define the main file. The first one is the signal file which defines the simulated FM radio function.

```
#!/usr/bin/env python3

import numpy as np

# Constants for the FM signal
fs = 10e6     # Sampling rate (Hz)
fc = 100e6    # Carrier frequency (Hz)
fm = 1e3      # Modulating frequency (Hz)
delta_f = 75e3 # Frequency deviation (Hz)
duration = 1  # Duration of the signal in seconds

def generate_fm_signal(d, e, f, t):
    """Generate an FM signal based on scaling factor d, modulating
        frequency e, and phase f."""
    return d * np.cos(2 * np.pi * fc * t + delta_f * np.sin(2 * np.pi *
        e * t + f))

def generate_audio_signal(t):
    """Generate the modulating sine wave signal."""
    return np.sin(2 * np.pi * fm * t)
```

The next one is the mcmc function file that defines the prior, likelihood, and probability function.

```
#!/usr/bin/env python3
```

```
import numpy as np
import emcee
from .signal import generate_fm_signal

def log_likelihood(theta, data, t):
    d, e, f = theta
    fm_signal_sim = generate_fm_signal(d, e, f, t)
    diff = data - fm_signal_sim
    return -0.5 * np.sum(diff**2)

def log_prior(theta):
    d, e, f = theta
    if 0 < e < 2 * np.pi and 0 <= f <= np.pi:
        prior_e = np.sin(e)
        prior_f = np.sin(f)
        prior_d = 1
        return np.log(prior_e * prior_f * prior_d)
    return -np.inf

def log_probability(theta, data, t):
    lp = log_prior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + log_likelihood(theta, data, t)

def run_mcmc(n_walkers, n_iterations, data, t):
    ndim = 3 # Number of parameters to estimate (d, e, f)
    pos = np.random.randn(n_walkers, ndim) * 10 # Initialize walkers
    sampler = emcee.EnsembleSampler(n_walkers, ndim, log_probability,
        args=[data, t])
    sampler.run_mcmc(pos, n_iterations, progress=True)
    all_walkers_positions = sampler.get_chain()
    return all_walkers_positions
```

Then, we find the correlation length.

```
#!/usr/bin/env python3

import numpy as np

def correlation_length(walkers_positions):
    """Calculate the correlation length by measuring the average
        distance between walkers."""
    distances = np.linalg.norm(walkers_positions[:, np.newaxis] -
        walkers_positions, axis=-1)
    return np.mean(distances)
```

Lastly, we plot the results.

```
!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np

def plot_correlation_length(correlation_lengths, n_walkers,
    n_iterations):
    plt.figure(figsize=(10, 6))
    plt.plot(range(n_iterations), correlation_lengths,
        label='Correlation Length')
    plt.xlabel('Iterations')
    plt.ylabel('Correlation Length')
    plt.title(f"Correlation Length vs. Iterations ({n_walkers} walkers,
        {n_iterations} iterations)")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f"{n_walkers}walker(s),
        {n_iterations}iterations_correlation.png")

def plot_walkers_paths(all_walkers_positions, d_true, e_true, f_true,
    n_walkers, n_iterations):
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')
    for i in range(n_walkers):
        path = all_walkers_positions[:, i, :]
        ax.plot(path[:, 1], path[:, 2], path[:, 0], color='grey',
            alpha=0.6) # e, f, d
    final_positions = all_walkers_positions[-1, :, :]
    ax.scatter(final_positions[:, 1], final_positions[:, 2],
        final_positions[:, 0],
              c='blue', marker='o', label="Final Position")
    ax.scatter(d_true, e_true, f_true, c='red', marker='x', label='True
        Value')
    ax.set_xlabel('Modulating Frequency (e)')
    ax.set_ylabel('Phase Shift (f)')
    ax.set_zlabel('Amplitude (d)')
    ax.set_title(f"Walkers' Positions Over Iterations (3D)")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{n_walkers}walker(s),
        {n_iterations}iterations_walkers_paths.png")

def plot_avg_positions(avg_positions, n_walkers, n_iterations):
    plt.figure(figsize=(10, 6))
    plt.plot(range(0, n_iterations, 10), avg_positions[:, 0],
        label="Average d (Amplitude)", color='blue')
    plt.plot(range(0, n_iterations, 10), avg_positions[:, 1],
```

```
        label="Average e (Frequency)", color='green')
    plt.plot(range(0, n_iterations, 10), avg_positions[:, 2],
        label="Average f (Phase)", color='red')
    plt.xlabel('Iterations (every 10 steps)')
    plt.ylabel('Parameter Value')
    plt.title(f"Average Walker Positions Every 10 Iterations")
    plt.legend(loc='best')
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f"{n_walkers}walker(s),
        {n_iterations}iterations_avg_positions.png")
```

Now we are ready to run the codes, with the proper setup file.

```
#!/usr/bin/env python3

from setuptools import setup, find_packages

setup(
    name="fm_mcmc", # Name of your package
    version="0.1.0", # Initial version
    description="Implementing Markov Chain Monte Carlo simulation to
        estimate parameters of FM signals", # Fix the description string
    long_description=open("README.md").read(), # Optional: You can write
        a README file
    long_description_content_type="text/markdown", # If your README is
        in Markdown
    author="Jasmine, Nich", # Replace with your name or the author of
        the package
    author_email="yhuan223@syr.edu, nmrubayi@syr.edu", # Replace with
        your email addresses
    url="https://github.com/Jazzman2100/project1.git", # Replace with
        the actual URL (if applicable)
    packages=find_packages(), # Automatically find all submodules (i.e.,
        fm_mcmc package)
    install_requires=[
        "numpy",     # Required dependency
        "matplotlib", # Required for plotting
        "emcee",     # Required for MCMC sampling
    ],
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License", # Replace with your
            license type
        "Operating System :: OS Independent",
    ],
    python_requires=">=3.6", # Minimum Python version supported
    entry_points={
        'console_scripts': [
```

```
            'fm-mcmc=main:main', # Ensure this is pointing to the main
                function in main.py (at root)
        ],
    },
)
```

3.2 Work distribution

3.2.1 Jasmine

1. Suggest choices of parameters.
2. Construct a Markov Chain Monte Carlo simulation.

3.2.2 Nich

1. Test the simulation codes.
2. Test the validity of the parameters.

3.3 Comparison on walker final position distribution

We compared the final walker position distribution for 20 walkers and 100 walkers, both with 300 iterations (over our burn-in iterations). They have the similar distribution, which means our code have this consistency in chains and between chains.

4 Conclusion

We were able to estimate the 'closeness of fit' of our model to the data. There are a few suggestions as to how we could make this model better. One way is to first optimize the parameters of our model by dividing up the data into subsets and updating the prior of one subset distribution with the posterior of the previous subset. Additionally, we can inform our proposal distribution using our new estimated priors. For instance, to ensure our chains are free to cover the entire probability space, our proposal function should include conditions that allow for walkers to start at frequencies that are not multiples of the prior frequency.

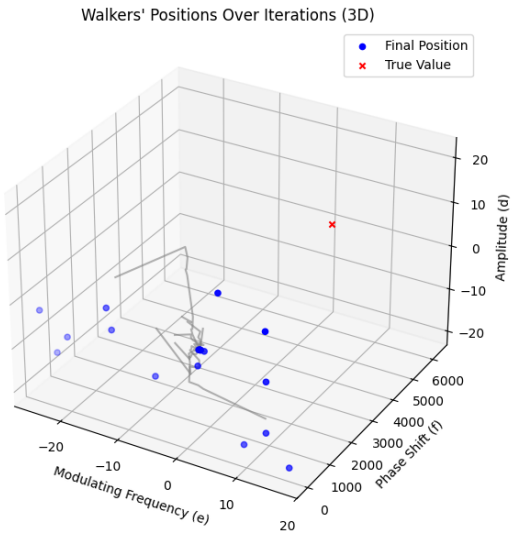

Figure 3: The walker final position plot with 20 walkers, 300 iterations.

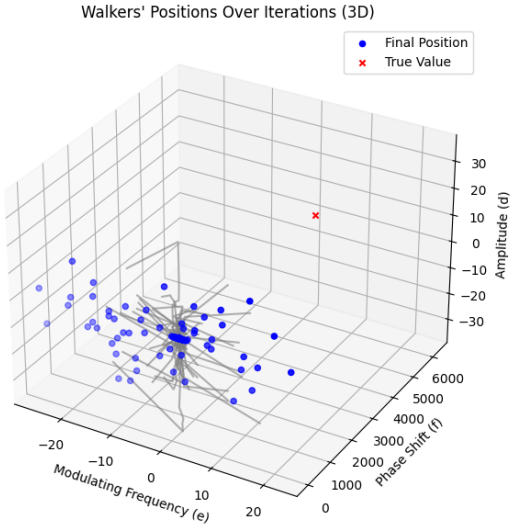

Figure 4: The walker final position plot with 20 walkers, 300 iterations.