Stochastic wave using Fractional Brownian Motion Noise

Lennart Heib

August 2022

1 Preamble

The Wave generation is based on the description of Giacomo Moretti and Gianluca Rizzello.

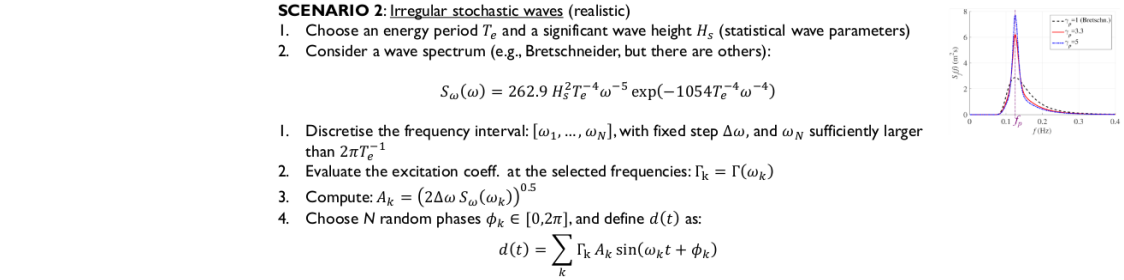

Figure 1: Algorithm for stochastic wave

2 Previous Version

Previously the algorithm was implemented exactly as described in Figure 1. The only source of randomness is the random shift of sine phases ϕ_k. The seed of this random shift was predefined so the wave would be the same each time the program executes. There were some minor problems with this version though.

1. Periodicity: Since the randomness only applies to the shift and is never changed the pattern will repeat.

2. The shape of the wave is dependent on the random shift. If two of the strong sine waves (waves near the maximum amplitude in Figure 1) have a similar random shift they will dominate the wave.

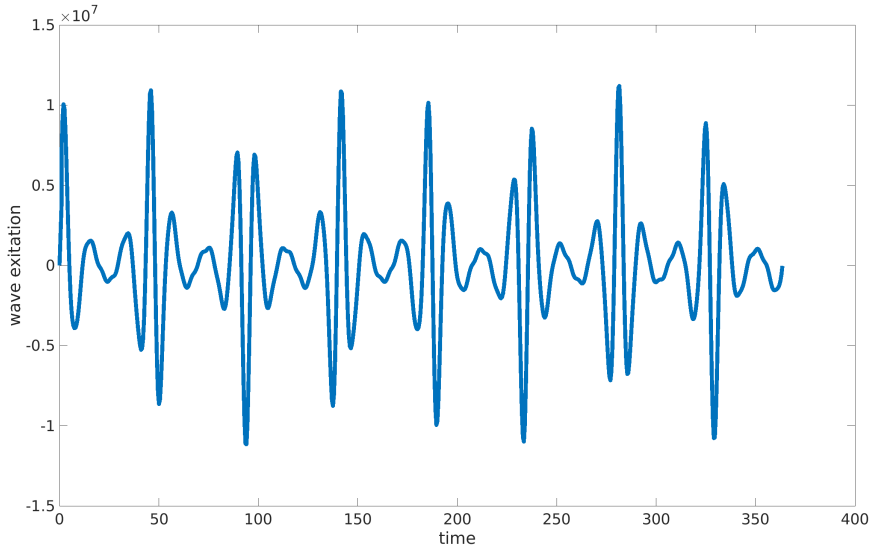

Figure 2: Example Wave generated using method described above

In Figure 2 the above described properties can be seen. The wave has a very similar pattern with a periodicity of about 50 seconds. Note, that the wave does not exactly repeat because of high frequency low amplitude waves contained in the signal. Still, there is enough similarity such that it is troublesome for our simulation.

3 Finding a suitable random number generator

The initial idea was to dynamically change the random phase shift ϕ_k so that the wave is less periodic. The problem is that standard random number generators are discontinuous and we can not simply apply a random phase shift to the signal and expect a continuous wave. We could, for example, change ϕ_k depending on the previous ϕ_k.

$$\phi_k^{t+1} = \phi_k^t + \phi_r \tag{1}$$

Where ϕ_r is a random increment. While this would result in a smooth random signal, the problem is, that we can only create the signal iteratively. If I wanted to evaluate the wave function $d(t)$ at $t = 100$, for example, the algorithm would have to calculate all 99 previous steps. The goal was therefore to find a smooth random number generator that could be evaluated at specific times. Luckily there are a class of noise generators that are designed exactly for this purpose. In our implementation we use the so called Fractional Brownian Motion Noise (FBM). The basic algorithm for this is to generate Gaussian distributed two-dimensional noise and then perform iterative spine interpolations to generate a smooth noise.

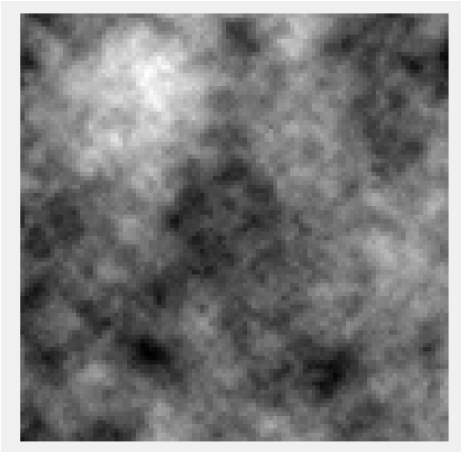

Figure 3: Two-dimensional FBM noise

4 FBM Noise

Using the FBM noise algorithm, we can create a string of numbers that are random but smooth. In two dimensions the result can be nicely visualized. In Figure 3 we can see the noise (scaled between 0 and 1 mapped to a grey-scale image). The random numbers are not distributed randomly but rather clustered so the total noise is smoothed over the two-dimensional plane. Since we only need to a one dimensional smooth noise (we want the phase shift ϕ_k to smoothly be changed each increment) we can use the 2 dimensional noise and create a cyclic path through it. We want the path to visit each pixel/value exactly once and we also want the path to loop. This kind of path is called a Hamiltonian Cycle.

5 Creating one dimensional cyclic noise from a two dimensional FBM map

Without further restrictions there are infinitely many paths that fulfill this criterion and describe a Hamiltonian Cycle.

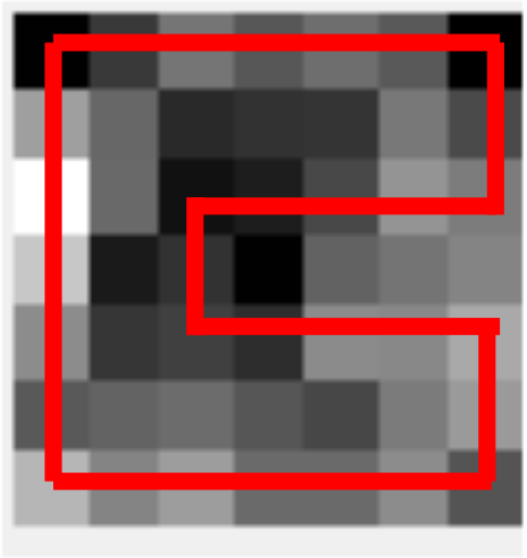

Figure 4: Hamiltonian Cycle on FBM Map

Figure 4 shows such a Cycle. This particular path works with square grids with vertices in the form $(N * 4) - 1$ but since the shape of the two dimensional input does not matter this condition can just be added as a restriction for the algorithm. The path shown skips one row each pass over so there is no systemic repetition in the path.

6 Randomness Evaluation

Firstly, the distribution of random numbers in the FBM map image has to be evaluated. The way the image is generated a Gaussian or Gauss-esq distribution would be expected but some test still have to be performed to check whether there is some form of bias and or unexpected distribution. Figure 5 shows the value distribution for different seeds. As expected the distribution is more or less Gaussian. Since there is no obvious systematic bias or other anomaly I stopped investigating here. Next, the random numbers generated with the path method described in the previous section is looked at. Figure 6 examines the distribution of values if the path method is used. Unlike the pure value histogram of the FBM map the path histogram shows significant biases. Seed 1 in Figure 6 has an expected value significantly larger then 0.5. While at first this seems counter-intuitive, if we look Figure 3 and imagine walking the previous described path we see that the path goes through non-homogenous sections. So if our path length is less then the total length of the image there is an expectation of a bias. The resulting noise then looks something like this. Figure

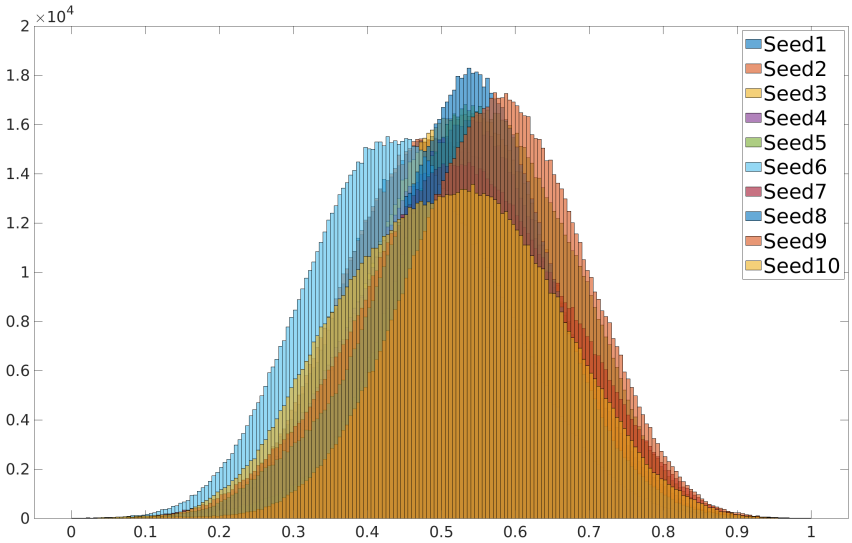

Figure 5: Histogram of value distribution of FBM Map N=1000^2

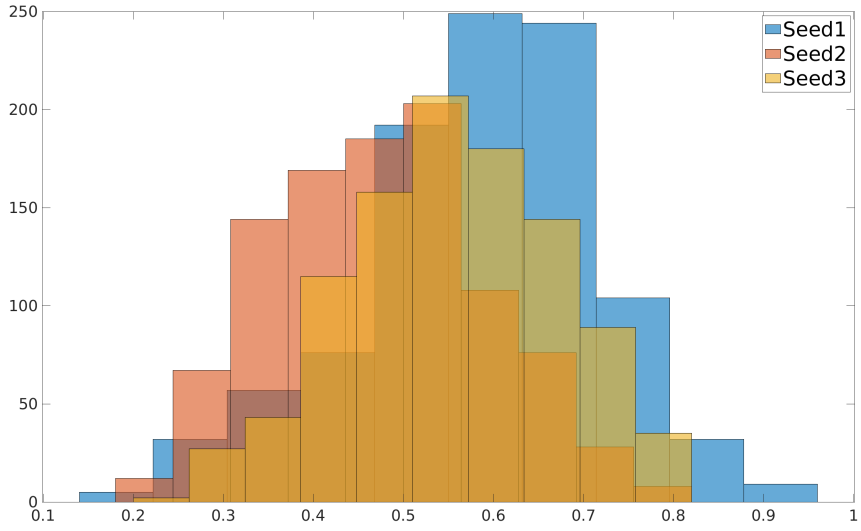

Figure 6: Histogram of value distribution of the path using different FBM maps

7 also shows the effect of the hyper-parameter "warp factor". A high warp factor translates to faster noise and a low warp factor to a slower, smoother noise. This implementation of the FBM noise fulfills both of our initial requirements. It is

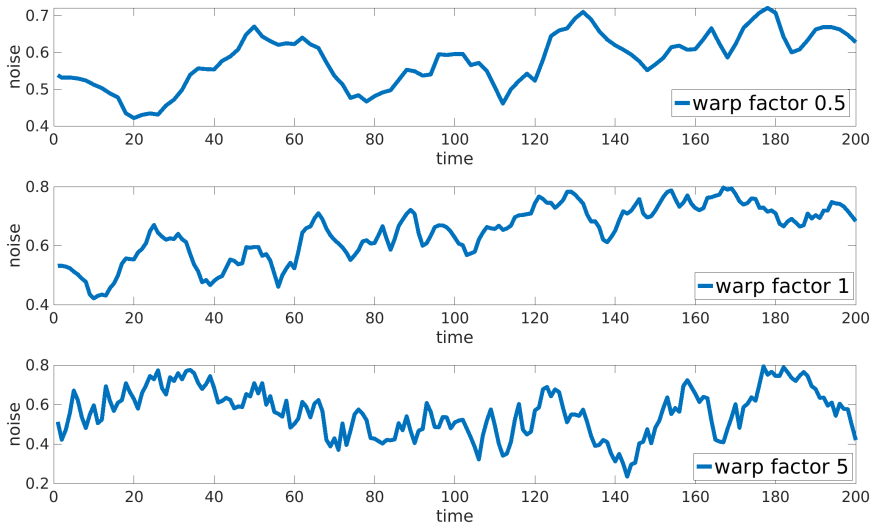

Figure 7: Output of random numbers for different warp factors

a smooth source of randomness and it can be accessed as an explicit function. Note, that this is only due to the cyclic nature of the path. There is however also a periodicity to this randomness, one of the problems we wanted to avoid using this implementation. This periodicity, however is arbitrarily large and not, like in the previous implementation, connected to the sinusoidal periodicity of the wave. It is for example easily feasible to create cycles with a periodicity of 10^4, i.e. a FBM Map with 100x100 Pixels. And in this example this would only mean the random pattern repeats after 10000 iterations and not that the wave pattern would. In reality much smaller maps are sufficient.

7 Some more algorithmic remarks:

1. Since the wave function also needs to be accessed at non-integer increments there is an additional linear interpolation between the discreet values.

2. To have an additional hyper parameter for the randomness I introduced a warp that stretches the input Map. For example a warp factor of 0.5 would create an additional increment between 2 values (linear interpolation) making the random noise "slower".

3. For the implementation I use a different map for each individual instance of randomness. Since the final wave is composed of a multitude if small waves, there are several separate instances. For example the random shift of wave 1 there is an individual map that corresponds that randomness. The maps are stored in a "persistent" variable so they are only created

once.

8 Wave generation

Section 1 already gives an algorithm for creating a stochastic wave. The only modification added are the random number generation. The frequency shift is then changed every increment with the smooth random noise. Section 6 showed the random numbers generated being in a Gaussian distribution with an expected value of about 0.5. This would vary a bit but generally we would expect average values ranging from 0.4 to 0.6. If we were to set the phase-shift ϕ_k equal to the random noise we would then expect most of the individual waves to have a phase-shift between 0.4 and 0.6. This, however would not be what was described in Section 1, the original algorithm. Therefore, we need one additional random factor to initialize the phases. The final algorithm then looks as follows.

1. Initialize Phase-shifts using Matlab Build in RNG. This RNG is uniformly distributed so we expect phase-shifts from 0 to 2π.
2. Save the initial phase-shift. Only change it if the seed changes.
3. Generate an FBM map for every individual wave.
4. Increment each wave every increment using the new smooth Gaussian noise

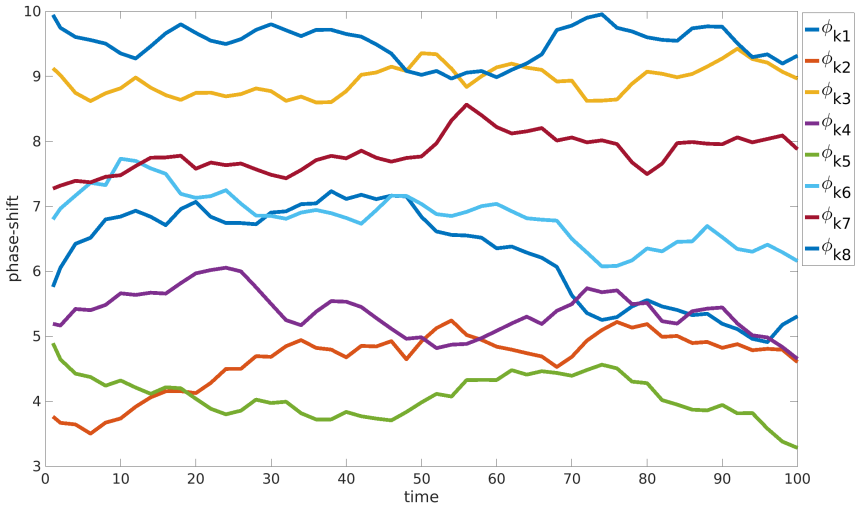

Figure 8: Phase-shifts of eight individual waves

Figure 8 shows what the resulting phase-shifts will look like. The phase shift at time $t = 0$ is the initial fixed random shift. On top of that the FBM noise is applied in such a way, that the change from one increment to the next is minimal. Figure 9 shows the then resulting final Wave function.

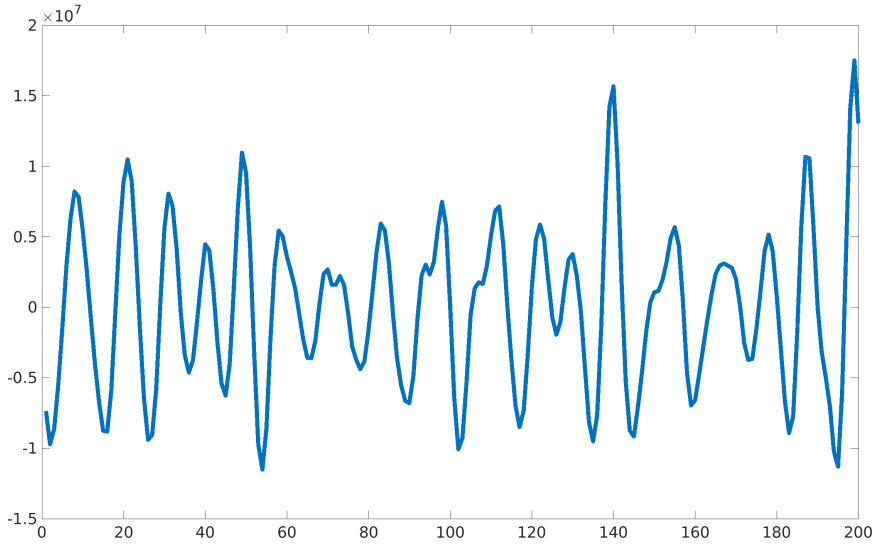

Figure 9: Stochastic Wave with FBM noise

9 Code structure

The code should have most sections explained. Still, here is a brief structure of the code:

1. FBMNoise(m,Seed): Generates the two dimensional FBM Maps. Needs dimension (m must be (N*4)-1 with $N \in \mathbf{N}$) and a Seed.This Code was not written by me but is available here

2. HamiltonianPathForSFBMNew(FBM,p): Takes an integer and a FBM map and returns the value at the position of the path.

3. FBM(time,[Seed,warpFactor]): Stores the Maps based on Seed (Map generated with Seed 1 is stored as Map.v1 in a persistent variable). Also handles interpolation between integer increments as well as the warping factor.

4. FBMStochasticWave(time,[warpFactor, Freqency Range, Amplitude Warp , PhaseWarp , Seed , TWave, HWave, NFreq, Debug]).Creates the wave function. Here all the hyper-parameters can be chosen in the arguments:

- WarpFactor: High warp factor means slower RNG. Different Factor for Amp and Phase.
- Frequency Range: The wave algorithm described in Section 1 requires the selection of frequencies. The Frequency range defines how large this range is.
- NFreq: similar to the Frequency range we need to specify how many individual frequencies we want
- Seed: This integer seed value influences all the individual seeds of the individual waves. New Seed –¿ completely new wave. Every aspect changes.
- TWave, HWave: Period and Hight of dominant Wave. Not really hyper-parameters but rather define the shape of the wave.
- Debug: If the debug flag is set the normal execution stops and the function enters into a loop where the wave is continuously drawn.

5. RunWaveExample.m Runs an example wave generation. Unlike the Debug function of the FBMStochasticWave(). Here, the waves is drawn at once and not continuously (much faster).

10 Wave Validation

The goal of this chapter is to find a heuristic approach to guess the average power of the wave. The wave is composed of N individual waves that are shifted by a random amount. If N, the number of waves/unique frequencies is increased, the average power of the wave increases. The first idea was to extensively test the correlation between these heuristic factors and the actual average power of the wave. At first the sum over all the amplitudes was a promising candidate. Figure 10 shows the prediction of the total power against the sum of amplitudes. Here a linear approximation seems to be working fine. I generated the Figure by iterating:

1. Frequencies from 7-50
2. Averaging power over 1000 seconds
3. 10 different Seeds per Frequency

The result was a polynomial of 1st order. By hard coding it into the wave generation and trying to predict and then adjust the wave power went just ok. The reason I don't explore it more here is that during testing it became clear, that the correlation of amplitudes and total power strongly depends on the Random phase shift ϕ_k. A uniform shift has a different correlation then a Gaussian distribution of phase shifts. This is not ideal since we would essentially hard code an amplitude adjustment and every time we change the wave generation we would have to change, and most importantly not forget to change the correlation function. All of these points led me to the conclusion that it will be much

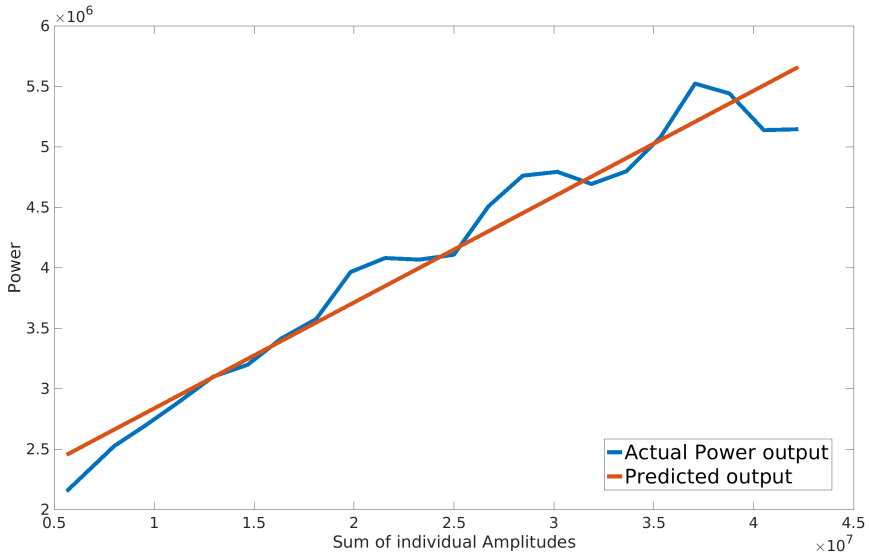

Figure 10: Prediction of power

easier to just test the average power in an initialization phase and save it for all future runs. (Similar to the saving of the input Maps with the FBM noise). The initialization takes about 0.34 seconds and only happens if the number of frequencies changes.

The Data displayed in Figure 11 was initialized with 100,300 and 1000 seconds and not reinitialized with different seeds. The testing hyper parameters were: 1000s, 25 Seeds per Frequency, 7-50 Frequency. The Figure shows, that for an accurate prediction we do not need to account for the seed at all. If the initial prediction is run with 1000 seconds the validation Saved as "Brute-Force".

11 Validation with evolutionary algorithm

Only bulletpoints here at the moment

- Paralellization difficult. Memory problem when used in current implementation. The Random Maps have to be stored somewhere. When using persistant every worker will generate their individual map. Large memory overhead.
- Idea 1.

12 TODO

1. Look for actual wave data to validate the model.

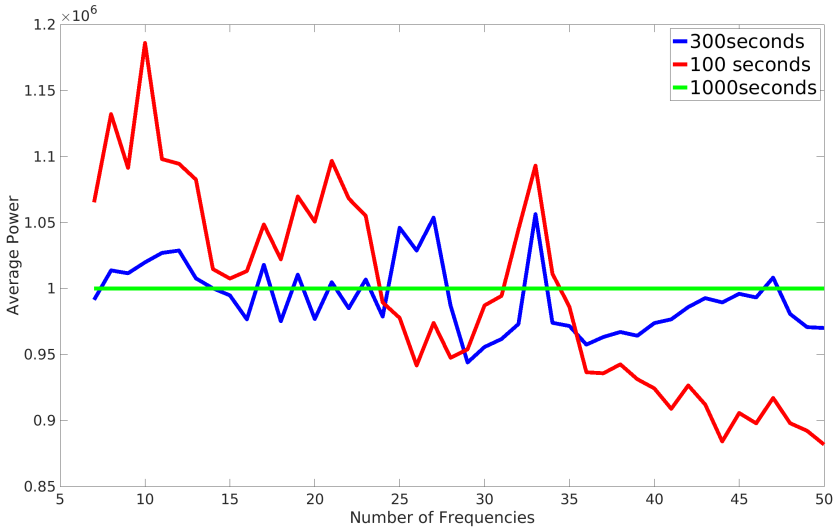

Figure 11: Validation of "Brute Force / Initialize and Save" Algorithm