Integration Python scripts generated by AI with LabVIEW environment

Kacper Waśniewski

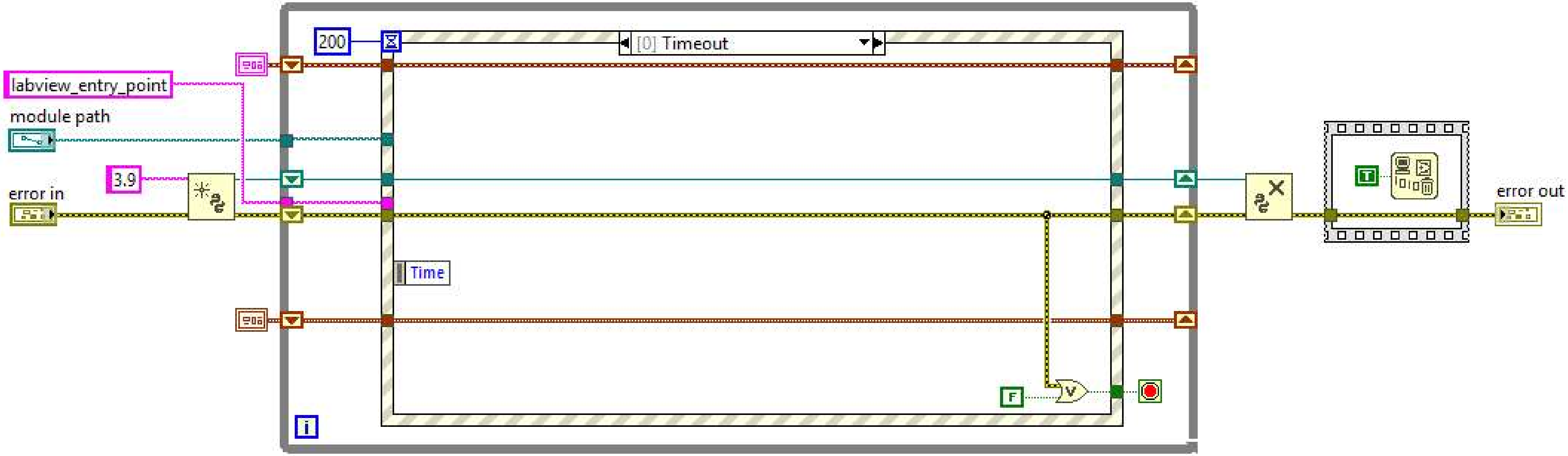
Example program

Program generate signal in LabVIEW with random parameters like offset, amplitude, frequency and phase. Before click “generate signal” the user should set number of generated samples and sampling rate. “Add noise” function added noise to the generated data. “Prediction params” handle execution of Python script and wait for a result as a table with predicted parameter of noise signal. The Python code is generated by Claude.ai.

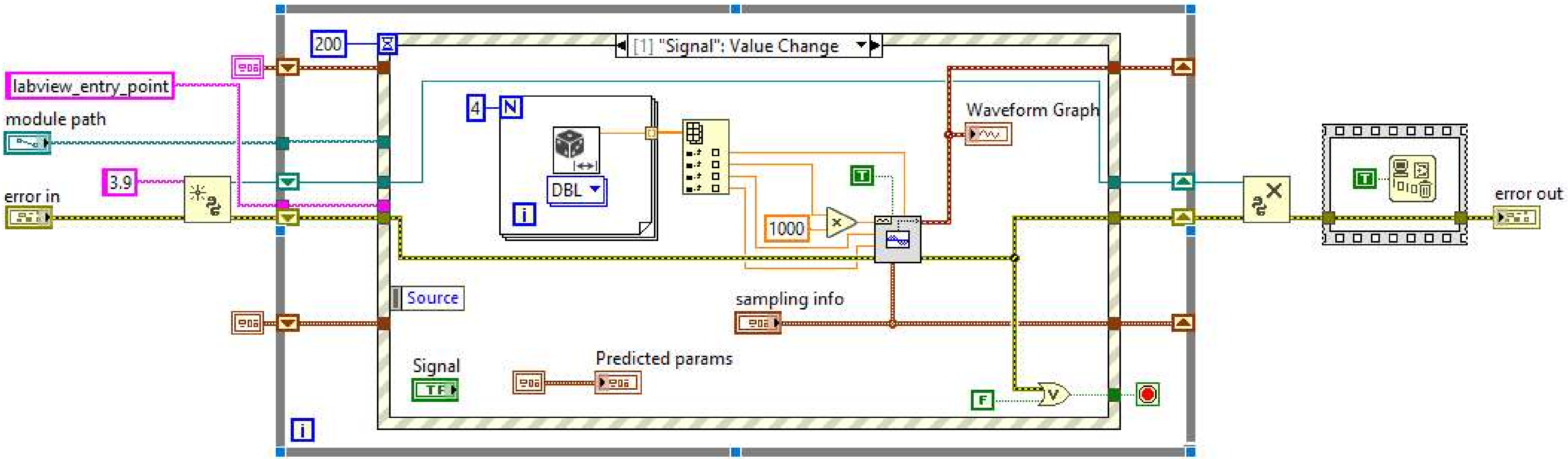
Integration LabVIEW with Python (NI documentation):

https://www.ni.com/en/support/documentation/supplemental/18/installing-python-for-calling-python-code.html

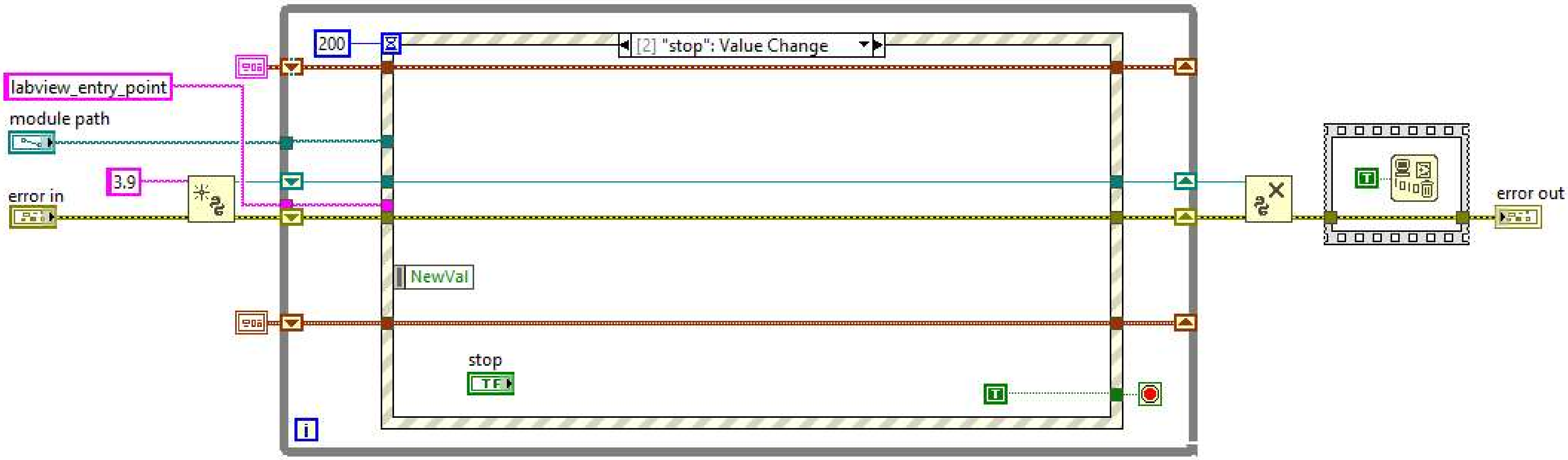
Timeout of Event Structure



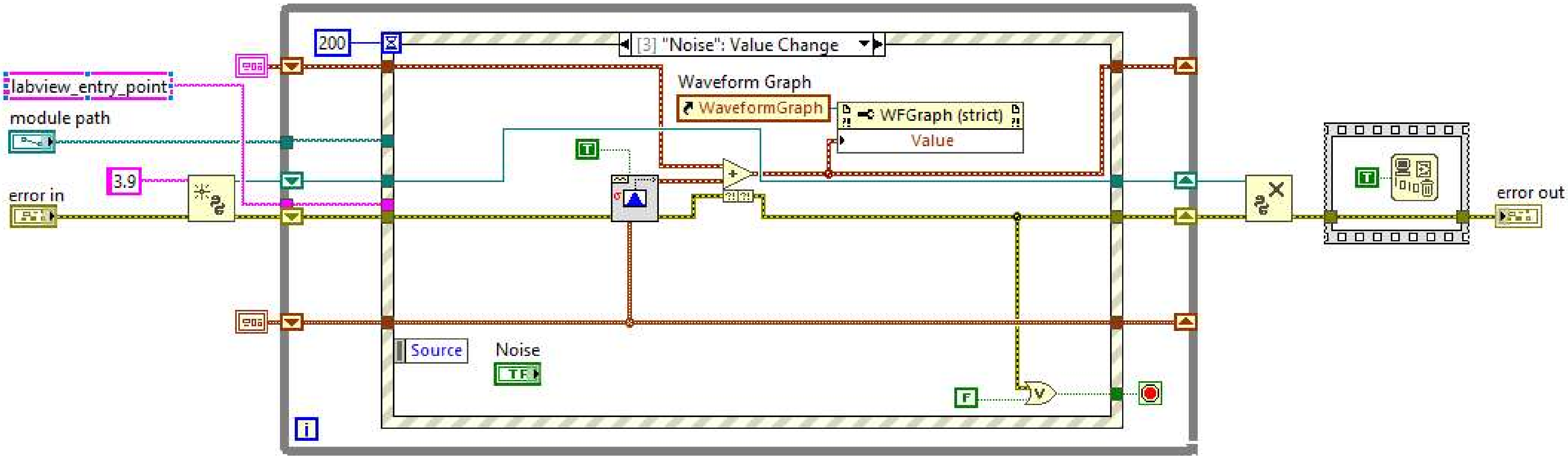
Generate signal – sampling rate and numbers of samples



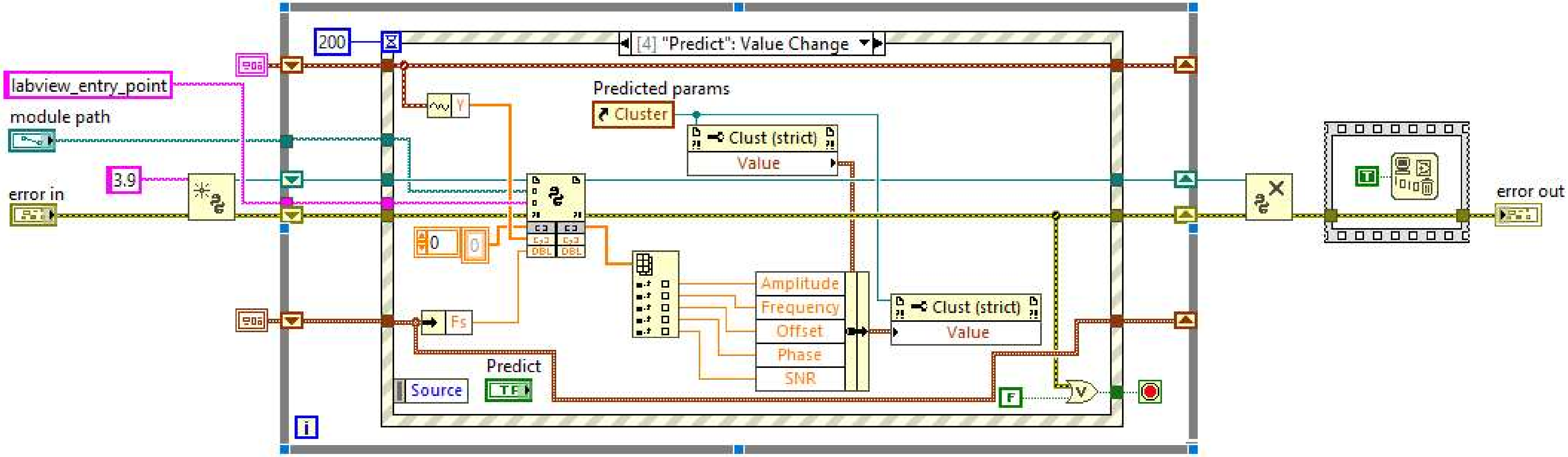
Stop program – button for close execution



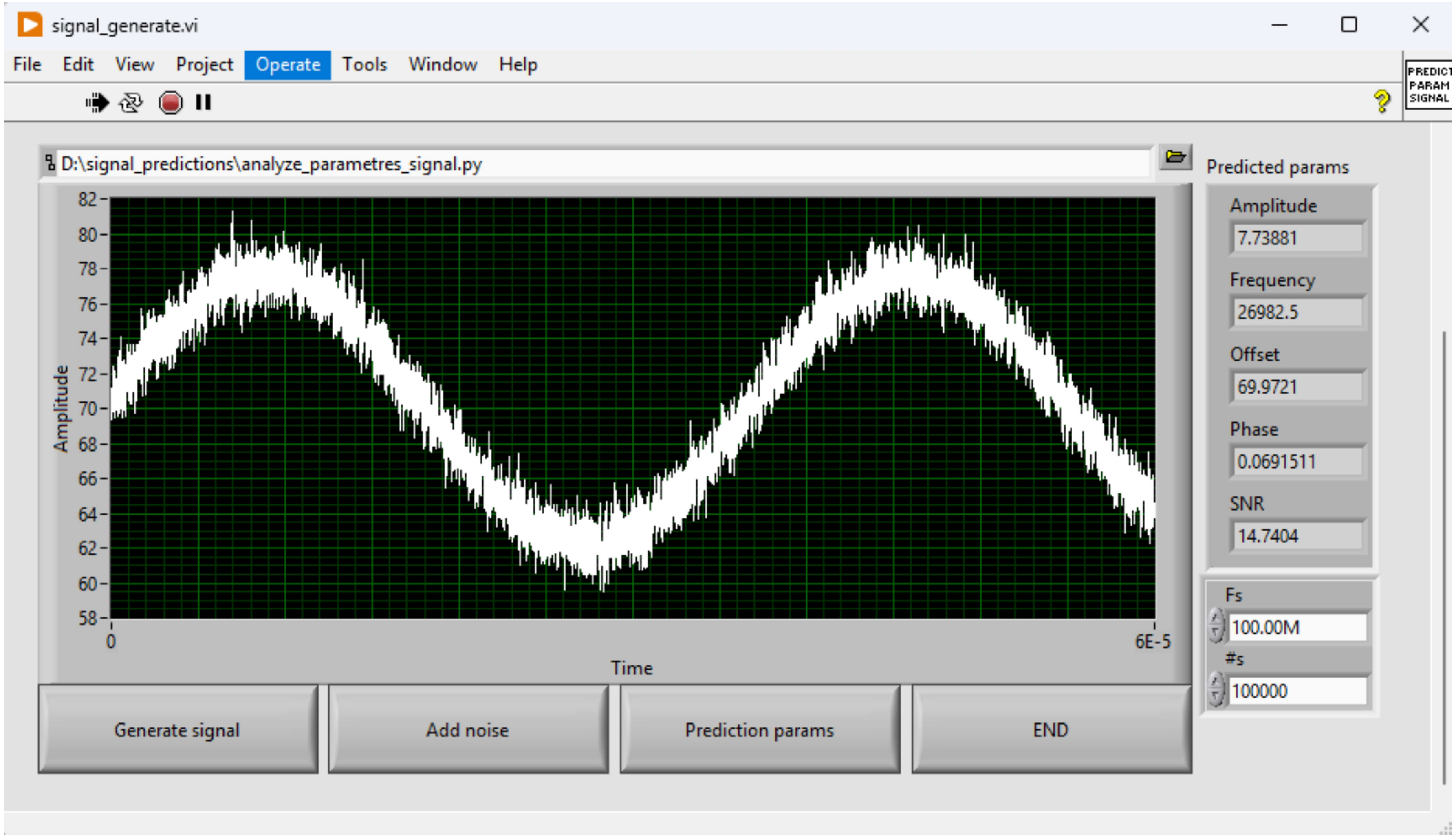
Add noise to generate signal



Handle external Python script



Front panel of application



D: > signal_predictions > analyze_parametres_signal.py > labview_entry_point

```python
import numpy as np
from scipy import optimize, signal

def analyze_signal_for_labview(data_array, sample_rate):···

def analyze_signal(data, sample_rate):···

def detect_signal_type(fft_values, fft_freqs, dominant_freq, time_domain_values):···

# Example of how the LabVIEW Python node would call our function
def labview_entry_point(signal_data, sample_rate):
    """
    Entry point for LabVIEW to call our function.

    Parameters:
    -----------
    signal_data : array-like
        1D array of signal data (amplitude values)
    sample_rate : float
        Sampling rate in Hz

    Returns:
    --------
    tuple
        signal_type, amplitude, frequency, offset, phase, snr
    """
    print("Signal Data:", signal_data)
    print("Sample Rate:", sample_rate)
    # Convert input data to numpy array if it's not already
    signal_array = np.array(signal_data)

    # Call our analysis function
    result = analyze_signal_for_labview(signal_array, sample_rate)

    # Return results as a tuple (easier for LabVIEW to handle)

    return [
        float(result['amplitude']),
        float(result['frequency']),
        float(result['offset']),
        float(result['phase']),
        float(result['snr'])
        ]

# signal_data = [0.1, 0.5, 0.3, -0.1, -0.5]
# sample_rate = 1000.0
# print(labview_entry_point(signal_data, sample_rate))
```

Code generated by Cloud.ai

GitHub:

https://github.com/Kacper2098dev/LabVIEW_Python_AI_example.git

Possible challenges during integration LabVIEW and Python

1. Data exchange between LabVIEW and Python

Large data handling: Exchanging significant volumes of data between the two can lead to performance bottlenecks

Type mismatch: Data types in LabVIEW don’t always map neatly onto Python’s data structures eg clusters, dictionaries

2. Error Handling and Debugging

Limited feedback: Debugging errors in Python scripts from LabVIEW can be challenging as the Python Node might not provide detailed feedback

Error propagation: Ensuring that errors occurring in Python are properly propagated back to LabVIEW for handling can require additional programming

3. Performance Concerns

Overhead of integration: Repeated calls between LabVIEW and Python can introduce latency, especially if data exchange is not optimized

Script execution time: Python scripts relying on external libraries might be slower due to computational complexity, which can affect real-time systems (machine learning)

4. Dependencies and library support

Python versions: LabVIEW supports specific versions of Python (Python 3.x) via its Python Node. If a script relies on incompatible Python versions, issues may arise

Library issues: Some Python libraries may require additional setup, which complicates the integration, particularly if the library includes compiled binaries that are system-dependent

5. Environment and deployment

Environment configuration: Setting up a compatible Python environment (path settings, required packages, etc.) can be error-prone

Cross-platform issues: Deployment to different operating systems might require different configurations for Python dependencies

Strategies to overcome these challenges

1. Use consistent data formats like JSON for smoother communication

2. Leverage LabVIEW's Python Node for direct integration and ensure Python environments are well-configured

3. Optimize data handling to minimize bottlenecks and ensure robust error handling across both systems

4. Document dependencies and use virtual environments in Python to make deployments predictable

Advantages of integration

1. Extended data analysis capabilities

2. Easy integration with external services and APIs, e.g. cloud applications, IoT

3. Wide range of data visualization applications, framweworks

4. Scalability

5. Combining these tools will facilitate cooperation between programming departments

6. Possibility to use AI tools to develop the LabVIEW

7. Possible acceleration of the engineer's work when creating programs that automate tests