Using MATLAB to Analyze Data Stored On NEEScentral

1. Overview

Whether conducting large-scale physical tests or numerical simulations, data resulting from these research endeavors often require extensive analysis using a multitude of external tools to post-process the results and ultimately make sense of the significance of the research. A common data mining tool used in the earthquake engineering community is MATLAB. This document demonstrates how a researcher can easily get data out of NEEScentral, bring it into MATLAB, perform calculations, and archive the processed data back into NEEScentral. The approach outlined in this document is only one method anticipated by NEESit for data usage. NEESit is currently working on methods to automatically extract data from NEEScentral using web services or other scripts in order to make the process more automated. This document provides background information about NEEScentral and in particular where in the NEES data model data files are stored, describes how users can upload and download data to and from NEEScentral, and provides sample scripts for MATLAB to do simple manipulations of the data.

2. The NEEScentral Data Repository

NEEScentral is the centralized data repository for the George E. Brown, Jr. Network for Earthquake Engineering Simulation. The repository provides a way to organize cutting edge earthquake engineering research data, as well as perishable reconnaissance data, and make both publicly available for dissemination to the broader earthquake engineering community.

NEEScentral users create and manage projects to store data and metadata collected over the course of their research whether it is physical experimentation, numerical simulations, or post-event reconnaissance. The current data model supports a hierarchical structure to help organize data and metadata. Each project is broken down into one or more experiments and/or simulations, which could represent a physical experiment, a numerical simulation, or a subset of a major reconnaissance investigation. Each experiment is further broken down into one or more trials to accommodate multiple data sets generated when slight modifications to the experiment are made. Simulations are broken down into runs instead of trials. Each trial has a folder associated with it for storing data, as does each simulation run. The trial *Data* folder contains four sub-folders to help organize data in a standard way across projects:

- The *Unprocessed Data* folder is for storing the most basic form of data available. For sensor data, measurements are usually in volts or some other non-meaningful unit. Sometimes equipment sites make use of sensors that automatically convert data into engineering units; this data is still considered to be unprocessed because it is the most basic form available.
- The *Converted Data* folder is for physical data that has been converted to more useful units by making simple conversions such as from voltages to strain, curvature, or displacement.
- The *Corrected Data* folder is for data that has been cleaned or revised to compensate for calibration problems, to eliminate noise, or to apply an overall correction factor.
- The *Derived Data* folder is for data generated by using information from one of the three other folders to plot, compare, or analyze results.

Users upload data to these folders, often as tab-delimited text files representing time-dependent sensor readings. See Figure 1. Instead of a *Data* folder, simulation runs have a*Files* folder that allows users to upload input and output files. Once uploaded, data is available for use to project members. Data files can be easily downloaded from their storage locations for analysis with external programs such as MATLAB. Once the data has been manipulated in MATLAB, a copy of the resulting data can be uploaded to NEEScentral for inclusion in the repository.

In addition to uploading data files to the *Data* and *Files* folders, users fill out web-based forms to input metadata that describes a variety of attributes at the project, experiment, and trial levels ranging from coordinate spaces to

input motions and loading histories. Consult the NEEScentral User's Guide for in-depth information about creating and managing NEEScentral projects.

```
Time                Sensor 1      Sensor 2 2003-12-05-07.59.27.323   -1.162800E-6   -5.814000E-7 2003-12-05-
07.59.27.328   -5.814000E-7   +0.000000E+0 2003-12-05-07.59.27.333   -5.814000E-7   -1.162700E-6 2003-12-05-
07.59.27.338   -1.162800E-6   -1.162700E-6 2003-12-05-07.59.27.343   -1.162800E-6   -5.814000E-7
```

Figure 1. Tab-delimited text file

3. MATLAB

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. It is assumed the reader is already familiar with MATLAB.

4. Accessing Data Stored On NEEScentral From Within MATLAB

To use data stored on NEEScentral within MATLAB, users must first download the data file(s) in which they are interested from NEEScentral. Data files are typically located in one of the four *Data* folders for experimental trials or in the *Files* folder for simulation runs. Once saved locally on the user's computer the data file may be opened in MATLAB using one of MATLAB's many functions for reading data in from a file.

An example of one such function isdlmread, which can be used to read data from a tab-delimited text file. The MATLAB script shown below reads in acceleration data from a tab-delimited text file, removes the linear trend, applies a low-pass filter to the data, resamples the data, performs a fast Fourier transform on the data, and saves the manipulated data as a tab-delimited text file called Spectra.txt. The new tab-delimited text file (Spectra.txt) can then be uploaded to NEEScentral for storage.

```
clear, close all

% Change path in quotes to where data is stored
cd('C:\Documents and Settings\mfraser\Desktop\2006-08-04-10.13.00_lvm')

%%%%%%%%%%% Load ASCII data
% Use dlmread to load an ASCII delimited file
Data = dlmread('test_06-08-04_0956.lvm', '\t', 22, 0);
% \t is for tab delimited data
% 22 Number of header lines to ignore
% 0 start with first point in row
% In this file, the 1st column is time and remaining 20 are acceleration

%%%%%%%%%%% Remove linear trend from acceleration data
Data(:,2:21) = detrend(Data(:,2:21));
% detrend removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

%%%%%%%%%%% Construct and apply a low-pass filter to acceleration data
% Construct a 5th order low-pass Butterworth digital filter with a 20 Hz cutoff frequency
% Calculate time step for file
dt = Data(2,1) - Data(1,1);
% Compute the Nyquist Frequency
F_Nyq = 1/2/dt;
% Set cutoff frequency to be 20 Hz
Cut_Freq = 20;
% Assign 5th order roll-off to filter
```

```
n = 5;
% Calculate the normalized cuttoff frequency
Wn = Cut_Freq/F_Nyq;
% Designs an order 5 lowpass digital Butterworth filter with normalized cutoff frequency Wn
[b,a] = butter(n, Wn, 'low');
Filtered_Data(:,1) = Data(:,1);
Filtered_Data(:,2:21) = filter(b, a, Data(:,2:21));

%%%%%%%%%%% Resample Data
Resampled_Data = downsample(Filtered_Data(:,2:21), 5);
Time = (1/200:1/200:length(Resampled_Data)/200)+Data(1,1);
Resampled_Data = [Time Resampled_Data];
% Decrease sampling rate of the sequence by a rate of 5 times the original sampling rate.
% This will change the sampling rate from 1000 Samples/sec to 200 Samples/sec

%%%%% Compute Fast Fourier Transform of Acceleration Data
% Construct a vector of time
Time = Resampled_Data(:,1);
% Determine size of data
[NTS N_CH] = size(Resampled_Data(:,2:21));
% Compute Fast Fourier Transform of Acceleration Data
Data_FFT = fft(Resampled_Data(:,2:21));
N = length(Data_FFT); % Determine length of FFT
% Compute the magnitude of the complex FFT data
Amplitude_Spectra = abs(Data_FFT);
% compute the phase angle of the FFT data
Phase_Angle = angle(Data_FFT);
% compute delta t again
deltat=Time(2)-Time(1);
% Compute length of resampled data
n=length(Resampled_Data(:,2:21));
% compute frequency increment
deltaf=1/n/deltat;
% compute Nyquist frequency
nyq=1/deltat/2;
% Construct frequency vector
f=(0:deltaf:nyq-deltaf);
% Compute single-sided amplitude spectra
Single_Sided_Amplitude_Spectra = Amplitude_Spectra(1:N/2,

Spectra = [f Single_Sided_Amplitude_Spectra];
save('Spectra.txt', 'Spectra', '-ascii', '-tabs');
```

Figure 2. Sample MATLAB script