View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarWorks at Central Washington University

Central Washington University
ScholarWorks@CWU

All Master's Theses Master's Theses

Spring 2018

Slip Estimation from Real-Time GPS in Cascadia

Jesse Senko
senko@geology.cwu.edu

Follow this and additional works at: https://digitalcommons.cwu.edu/etd

Recommended Citation
Senko, Jesse, "Slip Estimation from Real-Time GPS in Cascadia" (2018). *All Master's Theses*. 1014.
https://digitalcommons.cwu.edu/etd/1014

This Thesis is brought to you for free and open access by the Master's Theses at ScholarWorks@CWU. It has been accepted for inclusion in All Master's Theses by an authorized administrator of ScholarWorks@CWU. For more information, please contact pingfu@cwu.edu.

Slip Estimation from Real-Time GPS in Cascadia

A Thesis

Presented to

The Graduate Faculty

Central Washington University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Geology

---

by

Jesse Senko

April 2018

CENTRAL WASHINGTON UNIVERSITY

Graduate Studies

We hereby approve the thesis of

Jesse Senko

Candidate for the degree of Master of Science

APPROVED FOR THE GRADUATE FACULTY

______________ ____
Dr. Tim Melbourne, Committee Chair

______________ ____
Dr. Walter Szeliga

______________ ____
Dr. Breanyn MacInnes

______________ ____
Dean of Graduate Studies

Abstract

Slip Estimation from Real-Time GPS in Cascadia
by

Jesse Senko

April 2018

Current systems for rapidly characterizing earthquakes are based on seismic, teleseismic, and Deep-ocean Assessment and Reporting of Tsunami (DART) buoy data. These systems have significant limitations that hinder them from making rapid and accurate assessments of large earthquakes used for local tsunami warnings where run-up can occur minutes after the earthquake. Seismic and teleseismic networks saturate around M_w 7.0. Tsunami waves take tens of minutes to reach the buoys, so rapid assessment is impossible. GPS overcomes these limitations for large earthquakes. GPS does not saturate, and the offsets being detected occur very quickly after an earthquake. This thesis develops the algorithms necessary for detecting and characterizing large earthquakes from GPS measurements.

Point positioned GPS solutions are acquired from the CWU Geodesy Lab and filtered to detect offsets. Any detected offsets are then inverted to determine slip along the relevant faults. The moment and moment magnitude are calculated based on the estimated slip. The final solutions, detected offsets, calculated offsets and other relevant data are continuously pushed out to a database even when no earthquake is detected. The produced solutions can be used with existing methods to better inform tsunami estimates immediately following a large earthquake.

Acknowledgments

I would like to thank my committee, Tim Melbourne, Breanyn MacInnes, and Walter Szeliga. I would also like to thank the staff members of PANGA for their help in coding and technical issues throughout development, Craig Scrivner, Marcelo Santillan, and Rex Flake. Finally, I would like to thank my family for support and editing help.

Financial support was provided by National Aeronautics and Space Administration Research Opportunities in Solid Earth Science grant NNXlOAD15G. Operations of the Pacific Northwest Geodetic Array, including archiving and daily analysis of GNSS data, was supported by the USGS National earthquake Hazards Reduction Program Cooperative agreement G15AC00062.

Table of Contents

Chapter Page

I - Introduction ... 1

II - System Implementation ... 6

III - System Design Goals ... 7

IV - Kalman Filtering (General) ... 9

V - Inversion (General) ... 14

VI - Kalman Filtering ... 15

VII - Offset Detection ... 20

VIII - Site Selection ... 25

IX - Inversion ... 27

X - System Design ... 32

X1 - TULiveFilter ... 35

X2 - DataRouter ... 37

X3 - Kalman ... 38

X4 - DataWriter ... 40

X5 - TVLiveSlip ... 42

X6 - SlipWriter ... 48

X7 - RMQtoMDB ... 49

XI - Behavior ... 50

XII - Cascadia Implementation ... 54

XIII - South San Andreas Implementation ... 55

XIV - Future Work ... 56

XV - Conclusion ... 61

XVI - Works Cited ... 62

XVII - Appendixes ... 64

Appendix A – Config File Settings ... 64

Appendix B – Variables ... 67

Appendix C - Proof ... 68

Appendix D – Code ... 76

Figure list

Figure 1 - Estimated GPS-based magnitude estimates over time compared to seismic network estimates from the Tohoku-oki earthquake of 2011. Taken from Wright et al., 2012. 2
Figure 2 – Overall interagency system design and data sharing. READIMERGE and the offset detection and slip inversion are universal pieces of code being run at each agency independently to mitigate if an agency goes offline for any reason. 5
Figure 3 – An example of a simple Kalman filter. The x axis is the number of measurements. This is a single constant state example. The actual value is 20. The measurements, in red, have a random variable introduced. The predicted states of the system are the green x's. The covariance of the estimate is denoted by the green error bars on the predicted state. 12
Figure 4 – Example of GPS data that is being filtered. 13
Figure 5 - Offset logic between the various modes based on when a measurement comes in. 23
Figure 6 – Image shows how the state switching will behave with data. 24
Figure 7 - This is an example of why the system has to output 0's. This image shows an aftershock from the Tohoku-oki earthquake of March 11, 2011. If the system was still outputting the detected offsets from the initial earthquake, the aftershock would be completely overshadowed by the initial event and assessment of the aftershock would be impossible. 29
Figure 8 - This is a diagram of the data communication system in the program. The dashed boxes are instances of the same process function. The blue line denotes that the inversions are functions spawned out of TVLiveSlip. The black lines represent pipes. The green lines are queues. The red lines indicate that the inversions are simply spawned off and there is no communication back to TVLiveSlip once they start. RMQtoMDB is not a core process; only 1 instance of it needs to be running regardless of how many instances of the main program are running. 33
Figure 9 - This figure shows the control communication system in the program. When changes occur to the Config file, the changes are first interpreted by TULiveFilter and sent on to the appropriate processes. In this way, TULiveFilter has some knowledge of the state of the system. 34
Figure 10 - This is a diagram of how the smoothing matrix works. The boxes denote subfaults. The numbers inside show the value that the fault has in the smoothing matrix. Figure 13 shows how these subfault values translate into the smoothing matrix. 43
Figure 11 - This figure shows how the subfault values in figure 12 translate into a smoothing matrix. This only shows the top 2 subfault rows. The first 4

rows and columns, since every subfault those subfaults are adjacent to are also in the matrix, those rows and columns all add up to 0. In the full matrix, all rows and columns will add to 0. 44

Figure 12 - This figure demonstrates how the parallelization already implemented works. Essentially, since each inversion is an independent separate process, they can run concurrently. Inversion 1 is started and takes X seconds to complete. Before inversion 1 finishes, inversion 2, 3, and so on can all be started and run at the same time. Start time staggering, ideally ~1 second on a 1 Hz network, is necessary to not overload the system. Each inversion takes almost 100% of a computational unit, so there needs to be more than X cores for the system to work. 46

I - Introduction

Current earthquake rapid detection systems use seismic, teleseismic, and Deep-ocean Assessment and Reporting of Tsunamis (DART) buoys. These systems are currently not adequate to rapidly assess larger magnitude earthquakes (Ishii et al., 2005). Seismic networks tend to saturate around magnitude 7.0 (Melgar et al., 2013). Teleseismic networks require hours to produce estimates. DART buoys are not widespread, so accurate assessment of earthquakes takes longer (tens of minutes). GPS can augment both systems, providing faster estimates than buoy networks and more reliable estimates for large earthquakes than seismic networks.

Estimations of earthquake related hazards, especially tsunamis immediately after the earthquake, can also be improved using GPS. GPS provides faster and more accurate estimates of slip and slip distributions and resulting seafloor deformations for large earthquakes. These accurate estimates, in the minutes right after an earthquake, may be added to seismic and teleseismic networks to better inform tsunami estimations. Furthermore, GPS networks are rapidly expanding for various purposes and existing infrastructure can be used at little cost.

During the Tohoku-oki earthquake of 2011, GPS-based estimates were not in use. The seismic and teleseismic did not produce a formal estimate close to the final

magnitude, M_w 9.0, until 3 hours after the event (Wright et al., 2012). For the first 75 minutes, the estimate was from the saturated seismic network at M_w 8.1. During this earthquake, the tsunami started making landfall about 30 minutes after the event. The saturation led to severe underestimation of the tsunami hazard, particularly for coastal regions near the earthquake location. GPS-based estimates, shown in Figure 1, can produce estimates of M_w 8.8 about two minutes after the earthquake. This significantly more accurate and faster estimate of magnitude may be used to better inform tsunami hazard estimates in the minutes right after an earthquake, helping to reduce the underestimation of tsunami run-up and hazards.

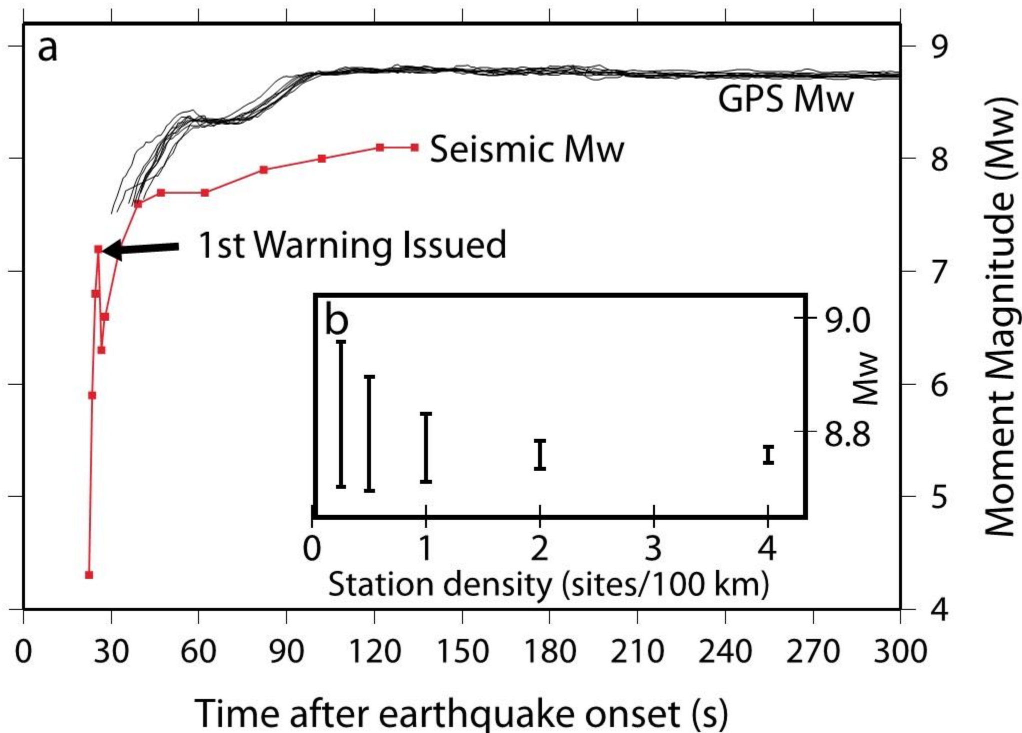

Figure 1 - Estimated GPS-based magnitude estimates over time compared to seismic network estimates from the Tohoku-oki earthquake of 2011. Taken from Wright et al., 2012.

Also, numerous people through various methods of shown that theoretically this is possible and potentially beneficial. Studies have shown that GPS-based offsets line up with similarly observed seismic records (Larson et al., 2003); GPS methods can provide faster magnitude estimates for large earthquakes (Wright et al., 2012); Filters can be designed that are capable of measuring offsets in data (Matthews & Segall, 1997). Finally, functional systems are being developed. Two current real-time GPS-based offset detection systems are set up at Berkeley Seismological Laboratory (Grapenthing et al., 2017) and in Japan (Kawamoto, et al., 2017). One system, though, uses a triggering mechanism from seismic networks, so issues occur if seismic network goes down. Japan's REGARD does not and is currently undergoing testing.

This project aims to take the current work already done and build a functional GPS-only real-time system. The system will take point positioned GPS data, filter it to detect offsets, then run a slip inversion all in real-time.

To achieve this, a Kalman filter (Zarchan & Mussoff, 2005) built in python specifically to detect offsets is used. Any detected offsets are inverted for a slip distribution and magnitude estimate. Afterwards, solutions are passed out to a MongoDB database and are viewable in the GPS cockpit.

The slip distributions can then be acquired by the tsunami warning centers and added to existing assessment methods to produce better estimates. Large events, which have the largest chance of producing damaging tsunamis, are difficult to accurately assess quickly using seismic and teleseismic networks. Augmenting current systems with GPS can address this issue. The drawback, though, is that GPS assessment systems are not as precise for small earthquakes or more prolonged, slower earthquakes.

Current partners in this work include NASA, NOAA, the Scripps Institute of Oceanography (SIO), the Jet Propulsion Laboratory (JPL), and the Pacific and North American Tsunami Warning Centers (PTWC and NTWC). Work is currently being done on the Real-time Earthquake Analysis and Disaster Mitigation (READI) system at SIO (Bock, 2013). JPL is working on the GPS-Aided and DART-Ensured Real-Time (GADER) Tsunami Early Detection System (Song, 2014). This thesis is part of the work being done at CWU.

The final goal is for SIO, JPL, and CWU to each independently produce point positioned solutions for GPS sites. The solutions will then be broadcast to the other groups, merged into a final solution and finally processed to detect offsets using a universal method. This thesis fits into the final portion, filtering for offsets and performing slip inversions. A visual representation of the overall system design can be found in Figure 2.

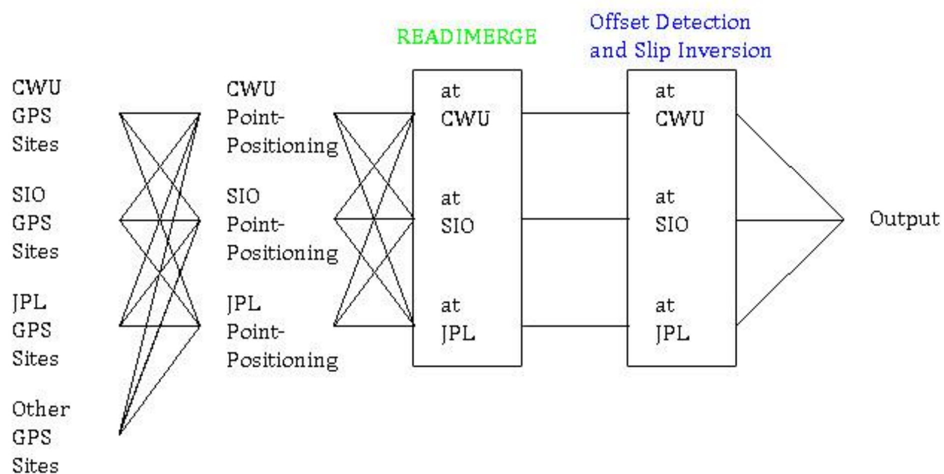

Figure 2 – Overall interagency system design and data sharing. READIMERGE and the offset detection and slip inversion are universal pieces of code being run at each agency independently to mitigate if an agency goes offline for any reason.

II - System Implementation

The current offset analysis system being discussed involves many GPS stations scattered throughout the Western US. From the stations, some GPS data is collected directly by CWU using radio arrays. Additional GPS data is collected at other centers and broadcast online, where CWU picks them up. Then, the GPS data is passed into Fastlane software (Santillan) for error corrections and processing. The resulting solutions are passed into an aggregator. From the aggregator, a RabbitMQ passes the data through to the offset analysis system, an offset detection and inverter program. The program carries out its processing and passes the results to another RabbitMQ. From there, a listener takes all the data and passes it through to a MongoDB. Then, the GPS cockpit, a viewer for GPS data developed at CWU, requests data from the MongoDB and displays the slip distribution and detected and calculated offsets in a window.

III - System Design Goals

The focus of this system is to rapidly assess large magnitude earthquakes. Standard seismologic networks can accurately and rapidly assess earthquakes with moment magnitude below 7.5. For this system, the goal is to assess earthquakes greater than moment magnitude 7 accurately and rapidly.

The system will be able to run many sites at the same time. It will be designed to allow some flexibility in the number of subfaults in the inversion. This allows the system to be adapted to what the computational system can handle. Both will be achieved through various versions of parallelization.

The system will be as accurate as reasonably possible. Random offsets, early earthquake pulls, and other factors will mean that this system will not produce slip distributions, offsets, and magnitudes that are publishable, but are accurate enough to act on shortly after an earthquake occurs. Data cleaning can be done afterwards to produce better results which may be used for future publications.

Speed is a focus. The system will produce results as quickly as possible, with the goal for a reasonable subfault model being about 30 seconds after first detection.

The system also needs to determine the offset and model it as a variable magnitude Heaviside step function. This is done to reduce the computational needs of this system.

Even with the parallelizations in place, the system should still be as lightweight overall as possible, both in terms of computation and memory usage by the whole system.

IV - Kalman Filtering (General)

Kalman filtering is a lightweight method for recursively determining the least squares best fit for a set of data (Zarchan & Mussoff, 2005). It uses a system based on the residuals between the measurement and the prediction of the measurement to determine Kalman filter gains. It uses the gains to adjust the system to the current least squares solution for the whole data set.

It achieves this by cycling through the three Riccati equations at each time-step to calculate the gains to adjust the data. The Riccati equations are

$$M_k = \Phi_k P_{k-1} \Phi_k^T + Q_k$$

$$K_k = M_k H^T (H M_k H^T + R_k)^{-1}$$

$$P_k = (I - K_k H) M_k$$

Where

M = previous covariance matrix for the filter

Φ = fundamental matrix describing how the system evolves

P = current covariance matrix for the filter

Q = process noise matrix

K = Kalman filter gains

H = measurement matrix

R = measurement covariance matrix

I = identity matrix

k = current time of the system

The filter then takes the Kalman filter gains and uses them to adjust the current predicted state to predict what the next measurement will be, i.e. the state of the system. To do this, it uses

$$Res_k = X_k - H\,\Phi_k\,S_{k-1}$$

$$S_k = \Phi_k\,S_{k-1} + K_k\,Res_k$$

Where Res is the residual matrix, X is the measurement, and S is the predicted state. This takes the error between the current state, S_{k-1}, and the measurement, X_k, and adjusts the next current state, S_k, based on how trustworthy the measurement is.

By doing this, the system evolves as each new measurement comes in. Ideally, with no process noise and a perfectly modeled system, the filter over time would become more and more accurate. When you are only measuring a constant value with noise, the filter will become a recursive least squares algorithm. But, Kalman filters can be used on more complex systems. With the addition of the process noise matrix, Q, the system does not have to be perfectly modeled in the filter. Kalman filters can reasonably track an object falling from a high altitude with wind resistance by only keeping track of position, velocity and acceleration by adding process noise to the filter to account for the wind resistance. But, better results would be found if wind resistance was accounted for in the filter. As the complexity of the model increases, the filter converges slower to the estimated states.

The filter covariance matrix, P, with no process noise, will converge towards 0 indefinitely. When process noise is added, P instead converges to a different value. This lets the filter adjust to newer measurements more than it normally would. If there are outside factors altering the value of the variables over time, the filter can account for them with process noise.

A simplistic example of a Kalman filter can be found in Figure 3. An example of data that is trying to be modelled is in Figure 4.

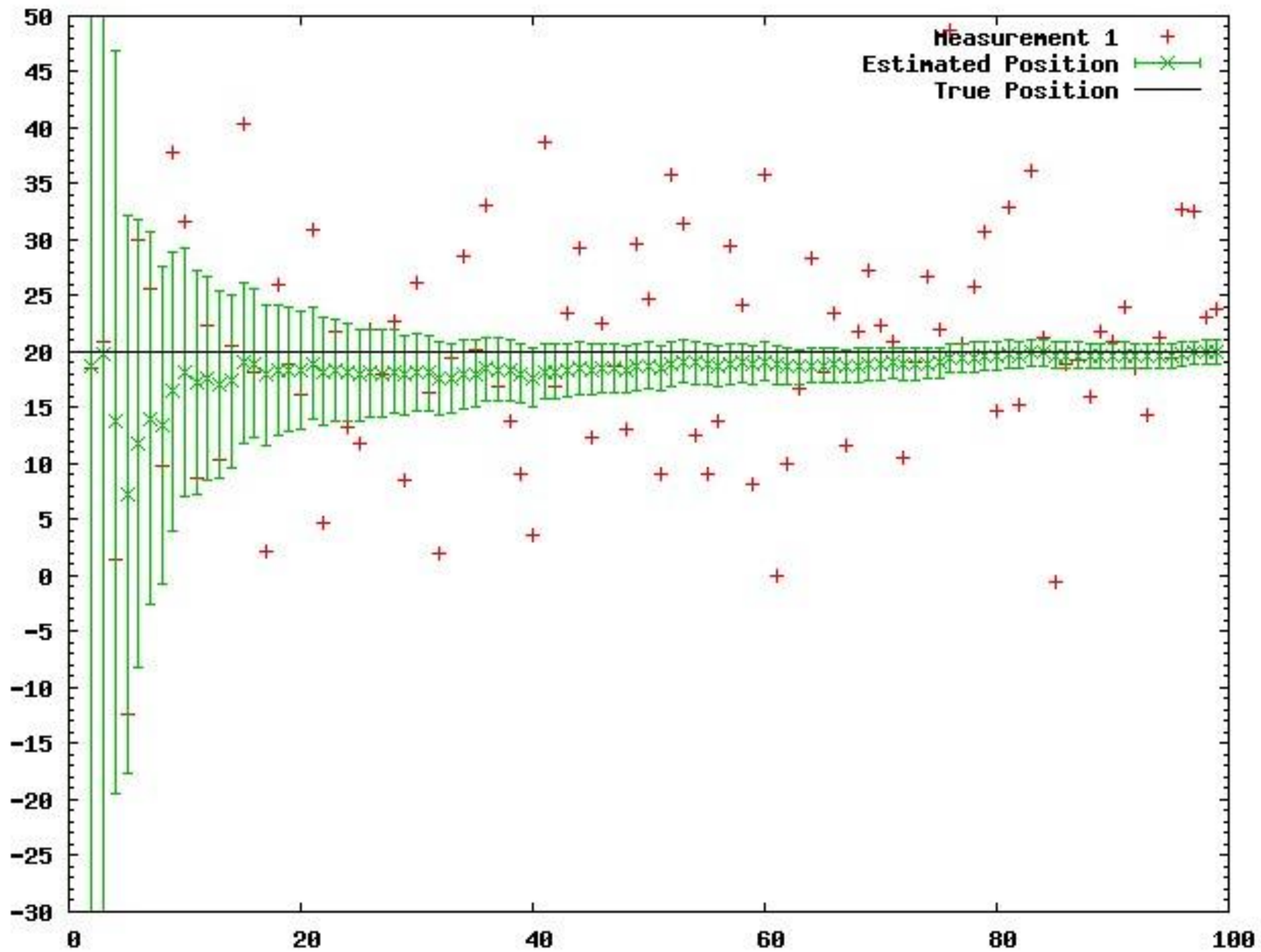

Figure 3 – An example of a simple Kalman filter. The x axis is the number of measurements. This is a single constant state example. The actual value is 20. The measurements, in red, have a random variable introduced. The predicted states of the system are the green x's. The covariance of the estimate is denoted by the green error bars on the predicted state.

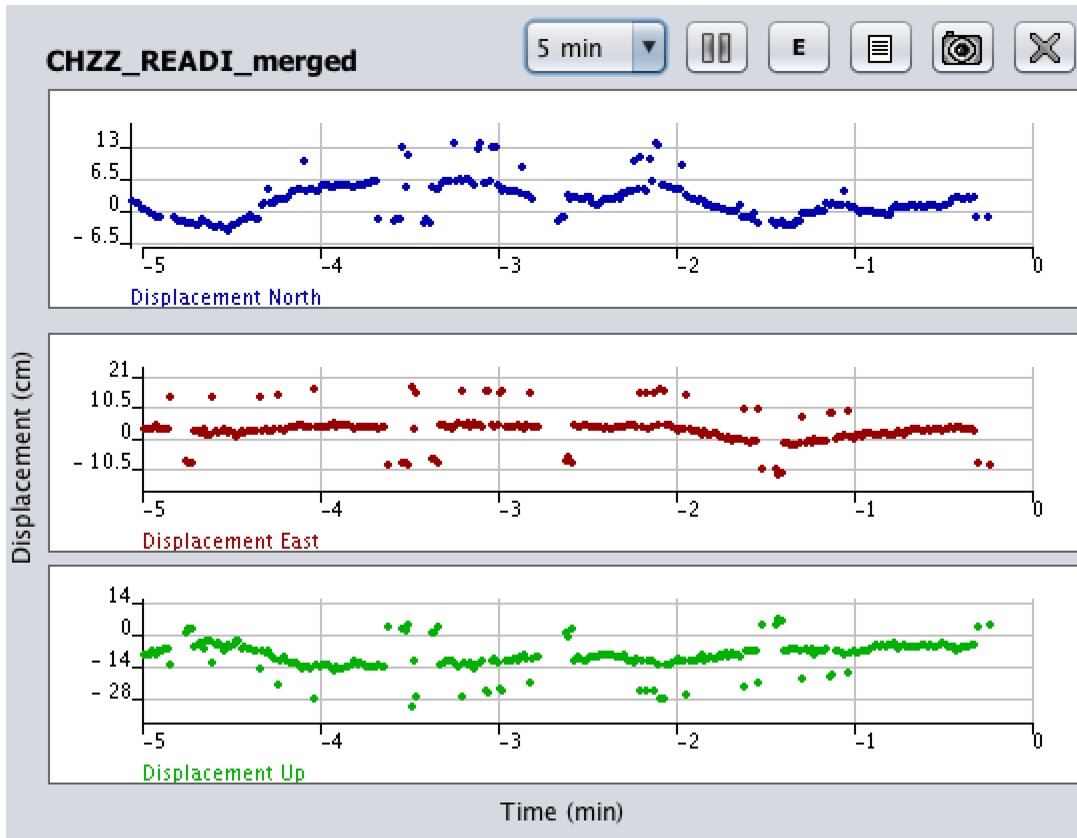

Figure 4 – Example of GPS data that is being filtered.

V - Inversion (General)

An inversion involves taking a known value multiplied by an unknown value with known outputs and solving for the unknown value. For example

A * unk = B

unk = B / A

or

unk = A^{-1} * B

In general algebra, this is easy to do, but with matrices, issues arise. There is no guarantee that B has an inverse if it is a matrix. Secondly, if B is not a square matrix, inverting it becomes much more difficult.

VI - Kalman Filtering

In this implementation, each Kalman filter is designed to process only one site. The north, east, and vertical are all processed in the same matrix with the offset detection being run on each site individually. This setup solves many potential issues.

This system assumes that the GPS stations stay in one place. This is not true; there are annual, semi-annual, linear and logarithmic signals. But, these signals occur over periods of months to years. For the purposes of a 1 Hz system over minutes to hours, the system does not show any of these signals and they can be ignored. Over the long term, the signals can be compensated for by introducing process noise into the filter.

This filtering method divides the total state and measurement into two different states. S_1 is the offset state. This state is the state that adjusts with every measurement as the filter runs. S_2 is a baseline state. S_2 represents the best known previous total state of the filter. Finally, S_T is the total state of the filter, $S_1 + S_2$. The residual, the difference between the predicted and actual measurement is compared to the total state, but only the offset state is allowed to adjust. This allows the earthquake to be modelled as a Heaviside step function

$$\text{Event} = a\,H(x_0)$$

Where $a = S_1$.

Φ in this situation becomes a 6 by 6 identity matrices. The other matrices become sparse matrices so the computation of the Kalman filter gains, K, is a little easier.

$$M_k = P_{k-1} + Q_k$$

$$K_k = M_k H^T (H M_k H^T + R_k)^{-1}$$

$$P_k = (I - K_k H) M_k$$

This implementation of the Kalman filter also changes the residual and state calculations.

$$Res_k = X_k - H S_T$$

$$S_{T(k)} = S_{T(k-1)} + K_k Res_k$$

And the system can stochastically reset.

$$S_{T(k)} = \Psi S_{T(k)}$$

$$P_k = \Xi$$

Where Ψ is the state reset matrix and Ξ is the covariance reset matrix.

This method of Kalman filtering, where each direction for a site is run simultaneously with the other two, makes it easier when a site detects an offset and must reset. A single offset is not expected to be detected in all directions, but it is assumed that when one is detected in one direction, all directions will move somewhat. So resetting them all at the same time is necessary. With all directions in the same process, they are easier to reset at the same time.

The Kalman filter also has an ongoing event mode. During normal operation, the filter resets the state at each timestep, making S_1 equal 0 and S_2 equal S_T. When an offset is detected, the system enters into the ongoing event mode. During this time, the filter does not reset the state. In this case, S_2 represents the best known previous state and S_1 is the difference between the current state and S_2. Therefore, S_1 being sent through to the inverter is not zero and inverted as if an event did occur. During this mode, the system is not allowed to reenter the mode until exiting it. This has two reasons; force the system to first identify where it is before trying to determine how much it has moved (when the filter is first turned on) and force the system to treat the offset as a single event, not multiple (when an event is detected). The event, if an earthquake, is not going to be a perfect Heaviside function. It is going to have an onset and duration during which the movement occurs. Therefore, it is expected that the measurements coming in are going to be bouncing around quite a bit. If left to its own desires, the filter will happily divide the offset into multiple smaller offsets. Forcing the system to not reenter the mode until it has exited is necessary so that the system captures the entire event in one mode session. The time spent is this mode can be adjusted to account for expected rupture duration and seismic wave travel time.

When initially started, the filter will enter a convergence period mode. This is the same as the ongoing event mode, except the system continues to reset the state so

that the detected offset, S_1, is 0. This is to allow the system to determine where it is before allowing it to 'detect' an offset.

One big issue that this Kalman filter design solves is time. Running all the sites in a single large matrix would require that all possible data be obtained before the Kalman filter moves forward a time-step. By running each site individually, it allows for asynchronous processing (i.e. site 1 could be at time-step X, site 2 at X+2, site 3 at X-2, etc.). This means that the most current data is being processed and if data is too old to be processed in the inversion, it can still be processed in the filter. This increases the accuracy of the filters overall by allowing as much data to be processed as possible. As an example, assuming the delay in DataWriter is 15 time-steps, a site could consistently be receiving data 90 seconds behind real-time. If the filters were running in one large matrix, then all that data would be ignored by the filter because it is too old. By running the filters individually and asynchronously, the data can still be processed into the filter, but it would be ignored in the inversion. This allows the filter to have a reasonably good idea of where that site is and, if the measurements start coming in closer to real-time, to more quickly converge onto the actual position of the site. This makes the overall system more robust because all the sites are positioned as best as they can be without any significant external time constraints.

One other big issue solved by this Kalman filter design is computational requirements. This method involves a bit more overhead since each process has

variables associated with it. But, these processes are very small and fast, and each filter requires the same amount of memory and computation requirements as all other filters. So, the filters will never bottleneck the system. By running all the sites in one large matrix, all possible variables would have to be stored in that one process. This includes all matrices, offset detection variables, gains, states, residuals, etc. Certain variables could be used universally, such as MesWait, but a large majority of the memory requirements would stay, just added together. Also, since all the sites need the offset detection to be run individually, as the number of sites increases the computational requirements would increase. Adding and deleting sites also becomes a bit more difficult since a correlation matrix would be needed to keep sites associated with their data, states, residuals, etc. So, while there would be some space and computational savings by running the filters in one large matrix, the linear stacking of the computational requirements from checks and individual site processing could potentially cause a bottleneck and drop the system out of real-time on networks with many sites. By running each site individually, there is no possible bottleneck in the filtering without choosing ridiculous settings for the filters. The computations can be vertically stacked across multiple cores and run at the same time to prevent interference with each other.

VII - Offset Detection

The offset detection is inside the Kalman filter itself. When a measurement comes in, the measurement is first checked to see if it makes sense. The measurement, X, is compared to the state, S_T. If the difference between the two exceeds a specified value, MaxOffset in the Config file chosen to be slightly unrealistic (i.e. 40m of movement in 1 measurement), then the measurement is ignored. Once this test has been passed the measurement begins processing.

The residual, Res, is calculated as the difference between the current measurement, X, and the predicted state, S_T. A threshold value, thres, is calculated by multiplying the standard deviation of the measurement, sqrt(R), times a constant, EQThres. The residual, Res, and threshold, EQThres*sqrt(R), values are then compared and the result of this is factored into the current mode of the filter to determine how the measurement is processed. If the residual is less than the threshold value, then the system proceeds as normal. If the residual is greater than the threshold value, then the measurement is considered anomalous.

There are 4 modes in the filter. The first is the detection enabled mode. This mode occurs when the residual is less than the threshold, Res<thres. During this mode, the incoming measurement is processed as normal, and the system progresses up to the current measurement.

The second mode is the possible event mode. This occurs when an anomalous measurement, Res>thres, has been detected. When this occurs, the measurement is stored while further measurements arrive. From this mode, the system will enter one of the last two modes once a determination has been made. The determination is made based on whether the number of consecutive anomalous measurements is greater than or less than a measurement wait value defined in the Config file, MesWait.

The third mode is the false event mode. This occurs when an anomalous measurement has been detected, but a normal measurement is detected before the measurement wait value has been reached. In this mode, all stored measurements are processed in order as if the state has stayed the same. This mode is to protect the system from declaring an offset every time one anomalous measurement has been detected. This mode only lasts until the stored measurements and current measurements are processed, then the mode switches back to the detection enabled mode.

The final mode is the ongoing event mode mode. This mode occurs when the number of consecutive anomalous measurements is equal to or greater than the measurement wait value. When this occurs, the system performs a reset. Then, the system processes all stored measurements as if there is a new state. The system is forced to stay in this mode for a defined duration as described in the Kalman Filtering section.

The logic behind how the modes switch between themselves can be found in Figure 5. An example of how the offset detection mode works can be found in Figure 6.

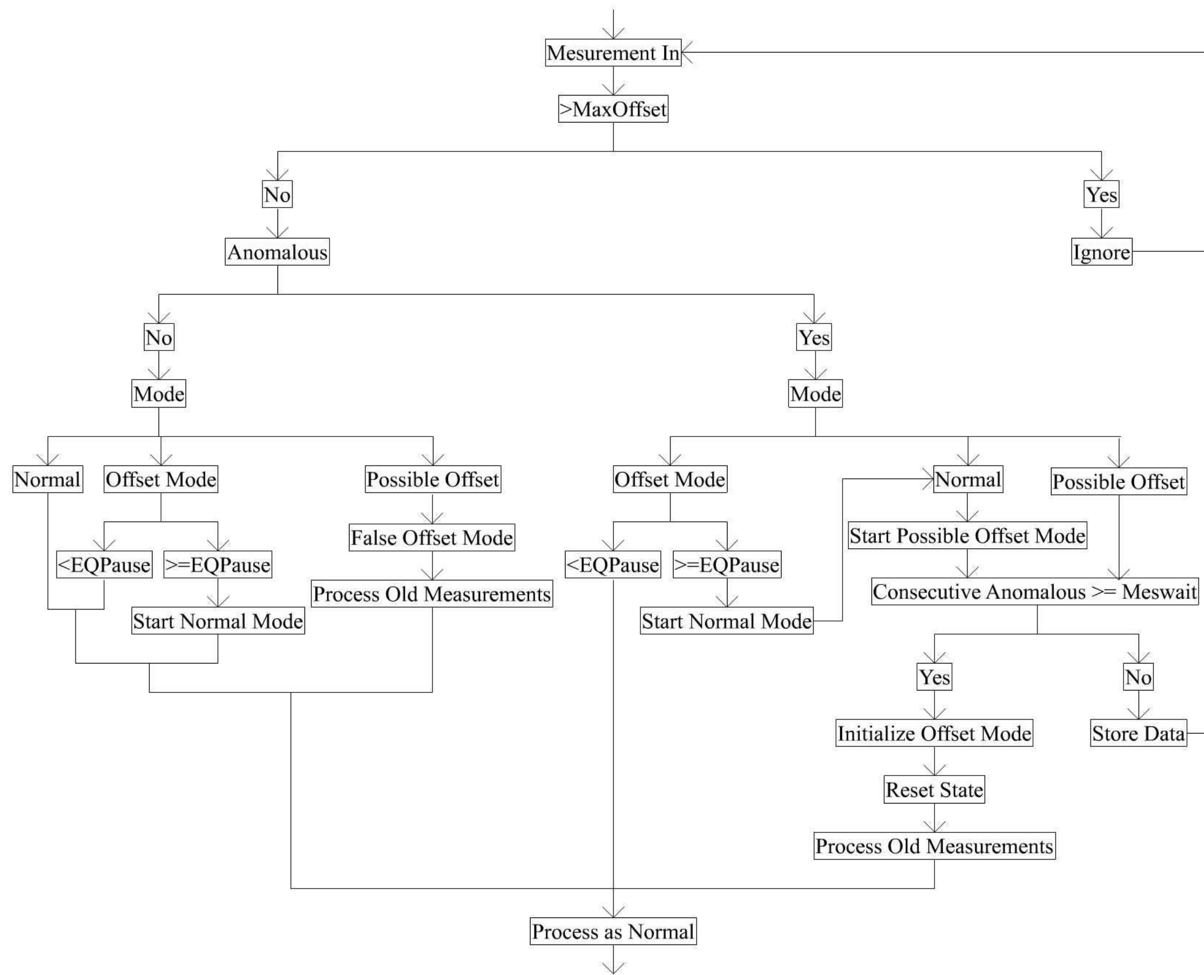

Figure 5 - Offset logic between the various modes based on when a measurement comes in.

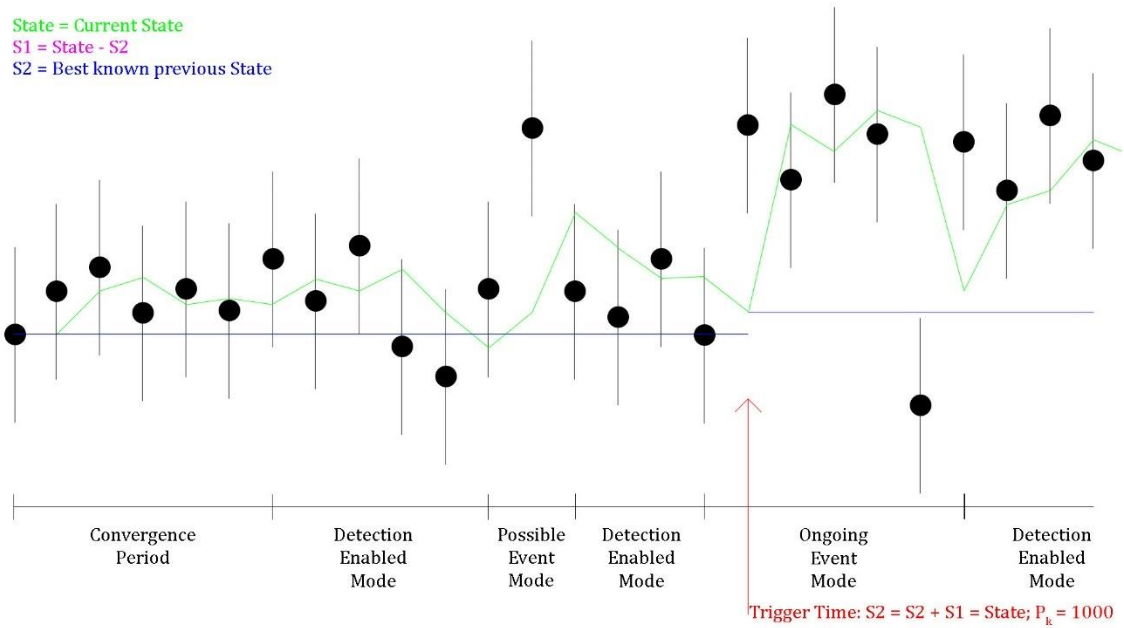

Figure 6 – Image shows how the state switching will behave with data.

One specific benefit resulting from the Kalman filtering method is that, since each site is run independently, each site does not need to know where it is in terms of latitude and longitude. As such, this data is ignored in the filter.

VIII - Site Selection

Site selection for this system is automated within the system. When the main inversion process receives a new site request, the sub-input matrix for the site is computed using the direction of convergence. The sub-input matrix is calculated using Okada (1992).

Then, the program looks at the resulting sub-input matrix to determine if the site is relevant to the current fault. The relevance is determined by looking at how many subfaults would cause an offset above a minimum offset, MinOffset, as defined in the Config file. If the ratio of irrelevant to total subfaults is below a value specified, RangeThres, in the Config file, then the site is added to the inversion and the relevance is relayed to the control program. Otherwise, the control program is told to ignore that site in the future. This is so that, for example, when looking at Cascadia, information coming from Japan is not included since any offset there from a Cascadia rupture would be undetectable by the system.

This method of site selection creates a more flexible system overall because data streams do not have to be sorted beforehand and a full system site list can be used, no need to cut out all irrelevant sites. To switch from one fault system to the next, replace the subfault file and everything else is handled, though adjustments to the Config file may be required for optimal performance on the new fault system.

When a new site is added to the network, all that needs to be done is to add the latitude and longitude to the full site list. Then, allow the data from that site to go through the system and any running systems will adjust and include the site if it is relevant.

IX - Inversion

The inversion requires some precalculated matrices, lists, and variables; the smoothing matrix, SM, to smooth the subfault solutions; the sub-input matrix, SIM, that describes how each station should move based on each subfault; and a correlation list, CL, that matches each station to the correct sub-input matrix lines. Also, it needs the output pipe, the lock for the pipe, the smoothing value, the subfault, and the station information. It also needs the data for the time-step that it is inverting for.

The calculation being represented in the smoothing matrix is described below

$$\text{alpha} \left(\Sigma \text{adjsubfaultslip} + Y \text{ subfaultslip} \right) = 0$$

where alpha is the smoothing constant, adjsubfaultslip is the slip on subfaults surrounding the subfault in question, Y is the number of adjacent subfaults, and subfaultslip is the slip occurring on the subfault in question. This equation smooths out the slip and keeps it from being unrealistic, i.e. one subfault having 60 meters of slip while the ones surrounding it have no slip.

The sub-input matrix is calculated based on the subfault strike and the direction of convergence based on the footwall of the fault. This restricts the fault movement to what has been historically observed. For example, the Cascadia subduction zone is not going to move in a strike-slip or normal faulting motion. The direction of convergence has been constant for millions of years and is not expected to change. Combined with

the use of a non-negative least squares (NNLS) inversion in the offset inversions, this restricts the movement to be in line with past observations (Lawson & Hanson, 1987).

The inversion takes these and constructs the appropriate sub-input (SIM) and detected offsets matrices (DOM) and the correlation list (CL), then passes it through to a sub-inversion. The sub-inversion iterates through CL and if it cannot find data for that specific site, it removes the site from CL along with the corresponding offsets in the SIM and the DOM. If it does find the site, it checks a tag in the data to determine if the filter is in an offset detected state. If the filter is not in an offset detected state, it sets the detected offsets to 0 and moves on.

Setting the value to 0 is necessary for the system to work properly. The system is not focused on inverting all the data; it is designed to only invert offsets. This helps to reduce the colored noise that would otherwise accumulate between events and skew the system. Also, during an event, if a site does not detect any offset from the event, it should be outputting 0's as necessary. An example of this is events such as aftershocks. Figure 7 shows an aftershock of the Tohoku-oki earthquake of 2011. If, in the inversion, the values had not been set to 0, then the lingering offsets from the main earthquake would overwhelm the inversion. This would work to mask the offsets from the aftershock and make the aftershock impossible to characterize. As such, setting the values to 0 unless an offset is detected is necessary for the functioning of this system.

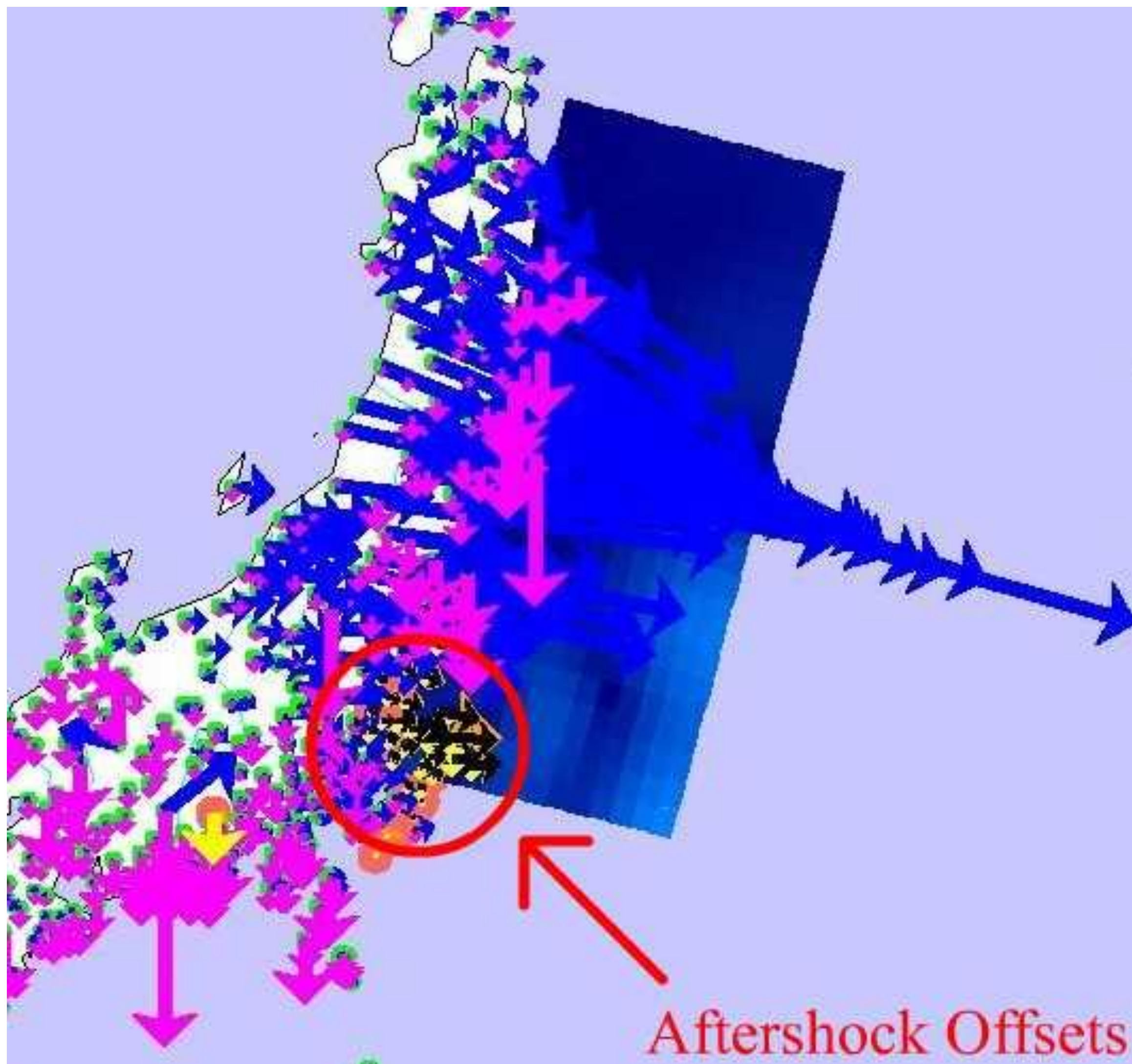

Figure 7 - This is an example of why the system has to output 0's. This image shows an aftershock from the Tohoku-oki earthquake of March 11, 2011. If the system was still outputting the detected offsets from the initial earthquake, the aftershock would be completely overshadowed by the initial event and assessment of the aftershock would be impossible.

After getting the DOM, the inversion adds the smoothing matrix multiplied by the smoothing constant alpha to the bottom of the sub-input matrix. It also adds a

matrix of 0's whose dimensions are the number of subfaults by 1 to the bottom of the offset matrix. It then uses the NNLS method to invert the matrix.

The equations below describe the inversion going on and how they relate to the various matrices.

SIM * Solution(SLIP) = DOM

SLIP = NNLS(SIM vstack SM, DOM)

The result of the inversion, SLIP, is then multiplied by the sub-input matrix, SIM, to get the calculated offsets, COM.

COM = SLIP · Subinput

Afterwards, the inversion uses the slip distribution to determine the moment and moment magnitude. The detected offsets, calculated offsets, slip distribution, moment, moment magnitude, and time are packed up and sent to SlipWriter.

Since each filter determines the offset independently, this system is essentially a bottom-up design in terms of its detection of earthquakes. It does not force the entire system into an earthquake-detected mode, a top-down method. It instead lets each offset be determined independently but processed as if true. The resulting inversion in false offset situations will be robust enough to not be completely fooled. And it also leaves the rest of the system ready to detect an actual earthquake with minimal error and no issues resulting from a forced mode. This method assumes that when an actual

earthquake occurs, multiple filters will detect offsets independently and the offsets will be consistent with a single event. Therefore, the resulting inversion will make sense. So, the slip distribution is the result of independent offsets and works in a bottom-up method.

X - System Design

Figure 8 shows the overall system and the communication network for data flow throughout. Figure 9 shows the communication network for controlling the system.

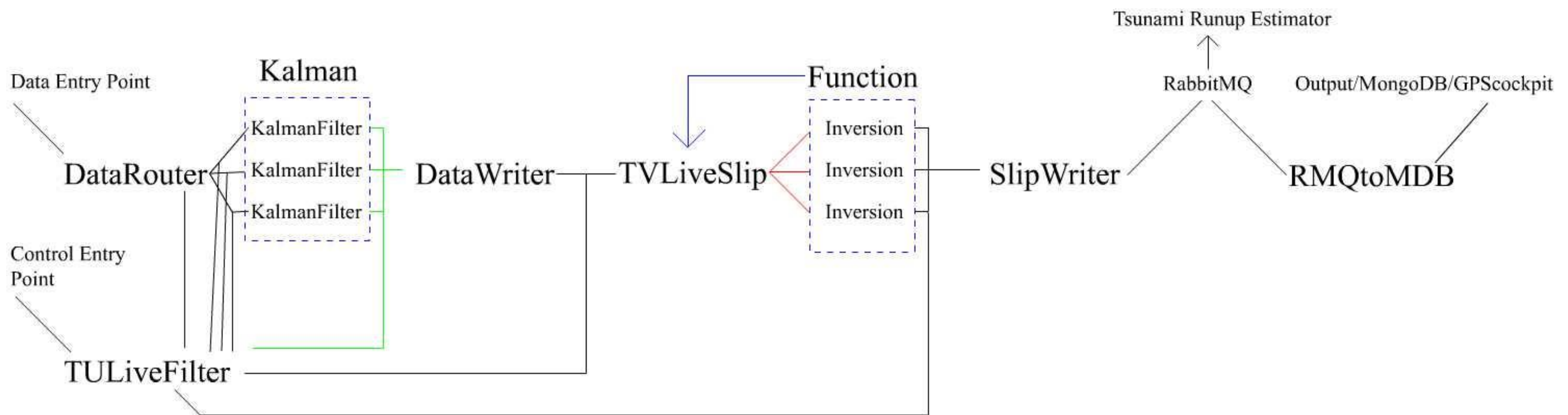

Figure 8 - This is a diagram of the data communication system in the program. The dashed boxes are instances of the same process function. The blue line denotes that the inversions are functions spawned out of TVLiveSlip. The black lines represent pipes. The green lines are queues. The red lines indicate that the inversions are simply spawned off and there is no communication back to TVLiveSlip once they start. RMQtoMDB is not a core process; only 1 instance of it needs to be running regardless of how many instances of the main program are running.

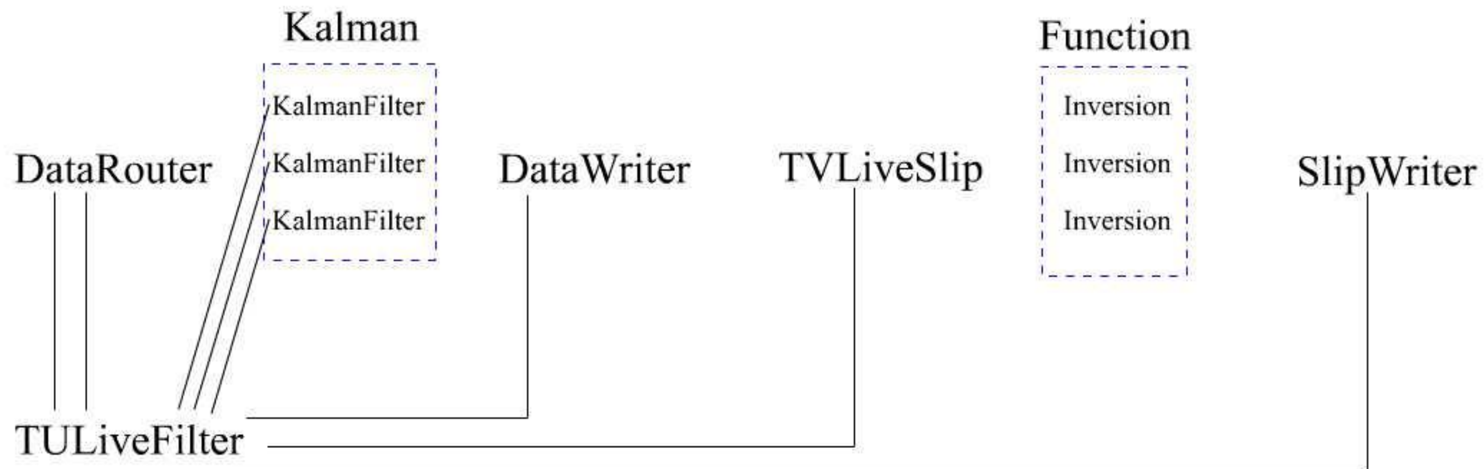

Figure 9 - This figure shows the control communication system in the program. When changes occur to the Config file, the changes are first interpreted by TULiveFilter and sent on to the appropriate processes. In this way, TULiveFilter has some knowledge of the state of the system.

X1 - TULiveFilter

This program is the entry point for the user. It starts up all other processes in the system except RMQtoMDB. Upon startup, it starts SlipWriter, TVLiveSlip, DataWriter, and DataRouter in that order. The reason for this startup order is so that the system is completely booted up before data starts being processed.

This process also starts up the Kalman filters as necessary. First, when data for a new site arrives at DataRouter, a request is sent to TULiveFilter. From there, a check request is sent to TVLiveSlip. The sub-input matrix, SIM, specific to that site is then computed for that site, and if that site meets the specified detection and relevancy requirements, the site is added into the sub-input matrix, other matrices are adjusted (DOM, COM and CL) and a response indicating to start the filter is sent back. From there, TULiveFilter starts a new Kalman filter with the necessary pipes. If the filter is restarting, TULiveFilter also sends the previous known state and variables. TULiveFilter then responds to DataRouter with the corresponding pipe. Then, DataRouter starts sending data for that site through that pipe.

If the SIM, does not meet the specified detection requirements, then TVLiveSlip responds indicating that the site should be ignored. Then, TULiveFilter sends a response to DataRouter indicating to ignore the site and any data for that site that is received is ignored.

TULiveFilter also controls when the Kalman filters are killed off. When a filter has not received any data for a specified time, the filter sends a request to kill itself off. Since TULiveFilter only checks that pipe occasionally, it sends a check back to the filter to make sure it still has not received any data. If it has received data, then the filter says to ignore the kill request. If it has not received new data, then the filter responds with its current state and shuts down. The current state is saved in TULiveFilter and then TULiveFilter sends a remove request to DataRouter, which removes the filter from the running list and responds. Then, TULiveFilter sends a request to TVLiveSlip to remove the site from the sub-input matrix. Finally, the site is then removed from the running list in TULiveFilter itself.

TULiveFilter also checks the Config file. First, it checks the last time the Config file has been changed relative to the last time it knows it has been changed. If they do not match up, then the current Config file is differenced against the previous Config file at .Config. Any differences are sent to the appropriate processes. Then, the Config file is copied to .Config and the last known modification time is updated.

X2 - DataRouter

The primary function of DataRouter is to route the data coming in on a single stream to the appropriate filter. Upon startup, DataRouter starts up two processes that watch the communication pipes to TULiveFilter. One watches for changes to the Config file. The other watches for when to remove a filter from the running list.

DataRouter then connects to the RabbitMQ and begins processing data. Normally, this involves decoding the data from json, checking if the site is in the pipe list, and sending it to the appropriate filter as necessary.

When data for a new unknown site is detected, a request is sent to TULiveFilter. While waiting for a response, any data for that site is stored. If the response is to ignore the site, the site is added to an ignore list and the stored data is deleted. If the response is to process the data, then the sent pipe is added to the pipe list dictionary and any stored data is sent to the filter.

If incoming data is for a site on the ignore list, then the data is ignored.

X3 - Kalman

The Kalman filter receives the decoded data from the DataRouter. If this is the very first measurement, it sets the baseline state, S2, to the current measurement. Otherwise, it sets up the measurement covariance (R), measurement (X), and residual (Res) matrices. Then, the residuals are compared to the maximum offset allowed, MaxOffset in the Config file, and if larger, the measurement is ignored. If that is not met, then the measurement and other matrices are passed through and processed in the filter. Then, R is compared to the minimum covariance value allowed, MinR in the Config file, and if the covariances are smaller, they are set to the minimum value. This is because the second Riccati equation will cause the filter to break if the covariances in R are zero.

The filter then does a quick check to see what mode it is in. If it is in a possible event mode, the filter skips updating the process noise matrix, Q, and the Riccati equations. Otherwise, it updates the process noise matrix by multipling an identity matrix by the amount of time since the last measurement in seconds and updates the Riccati equations.

Regardless, the matrix then calculates the residual matrix, Res, checks whether the filter is still starting up, and then determines the new mode for the system. To do this, the filter compares the residuals to the standard deviation, sqrt(R), of the measurement times a constant defined in the Config file, EQThres. The modes that the

filter can enter and how they impact processing are described in the Offset Detection section.

Normally, though, the filter will update the state, S_1, using the current gain matrix, K, pack it up, and send the data on to DataWriter.

X4 - DataWriter

DataWriter is created with a queue. The Kalman filters put their processed data into this queue. When the data is gathered from the queue, the times are read, and the data is put into a list for that specific time-step.

If the read time-step is greater than the current time-step in DataWriter, then the current time-step in DataWriter is updated. This process then sends the data to TVLiveSlip. This is done by going through the list of time-steps and sending the data for each time-step that is less than the current time-step in DataWriter minus a specified delay, DWDelay. The sending of all data for a time-step to TVLiveSlip is necessary for the inversion process.

If the data received is below the last sent time-step, then the data is ignored.

The first purpose of this process is to hold and sort data from the filters. Since all the data from the initial RabbitMQ does not arrive ordered, time-steps from some sites may be processed before the same time-steps from other sites. This process implements a pause to allow as many sites to report data as reasonably possible. This, though, must be balanced with how long the delay can last before impacting the overall goal of the whole system.

The second purpose is to delay the output to TVLiveSlip. TVLiveSlip starts a sub-inversion as soon as it receives data but sending 5-10 seconds of data in a very short time would impact the performance of the sub-inversions. To limit this, there is a pause between sending separate time-steps to temporally spread out the workload.

Lastly, this process also reduces the workload on TVLiveSlip. By organizing the data into separate time-steps before sending it to TVLiveSlip, TVLiveSlip can focus on sub-input matrix, SIM, calculations for the relevancy of new sites and spawning off sub-inversions.

X5 - TVLiveSlip

TVLiveSlip, upon startup, creates a smoothing matrix, SM, based on the subfaults. This is currently running with no corner fix. This means that for the corners and the sides, the number for each subfault corresponds to the number of adjacent subfaults. Figure 10 shows how this works out spatially, and Figure 11 shows how this works out into the smoothing matrix. When corner fix is turned on, the diagonal is set to 4.

2	3	3	2
3	4	4	3
3	4	4	3
2	3	3	2

Figure 10 - This is a diagram of how the smoothing matrix works. The boxes denote subfaults. The numbers inside show the value that the fault has in the smoothing matrix. Figure 13 shows how these subfault values translate into the smoothing matrix.

2	-1	0	0	-1	0	0	0
-1	3	-1	0	0	-1	0	0
0	-1	3	-1	0	0	-1	0
0	0	-1	2	0	0	0	-1
-1	0	0	0	3	-1	0	0
0	-1	0	0	-1	4	-1	0
0	0	-1	0	0	-1	4	-1
0	0	0	-1	0	0	-1	3

Figure 11 - This figure shows how the subfault values in figure 12 translate into a smoothing matrix. This only shows the top 2 subfault rows. The first 4 rows and columns, since every subfault those subfaults are adjacent to are also in the matrix, those rows and columns all add up to 0. In the full matrix, all rows and columns will add to 0.

The detected offset matrix, DOM, is also created. A dummy site, DUMMY, is added so that the matrix maintains the correct dimensions when empty. Several other matrices are also created with the correct dimensions during startup.

When the main process starts up, a control pipe watcher is started up. This allows TULiveFilter to communicate the settings and changes in settings to this process. Certain variables cannot be changed without restarting the whole program because they are immensely important and are incredibly difficult to change on the fly. Details of this can be found in Appendix A – Config File Settings. Mostly, though, the control pipe watcher is used for adding, removing, and checking for relevancy of sites as the filters are started and stopped.

An inversion watcher process is also started. This process occasionally kicks on and runs through the list of all running inversion processes. If an inversion process has been running for too long, it kills the inversion and logs that it was killed. This stops the whole system from clogging up when an inversion hangs for any reason.

The main process gathers the data from DataWriter and passes it to a spawned inversion process along with all other necessary variables. In this way the system parallelizes the inversions. Multiple inversions can be running at once without causing the system any significant issues. This allows for inversions that take longer than 1

second to process to be run. Figure 12 shows how the inversions would stack to allow for multiple at the same time.

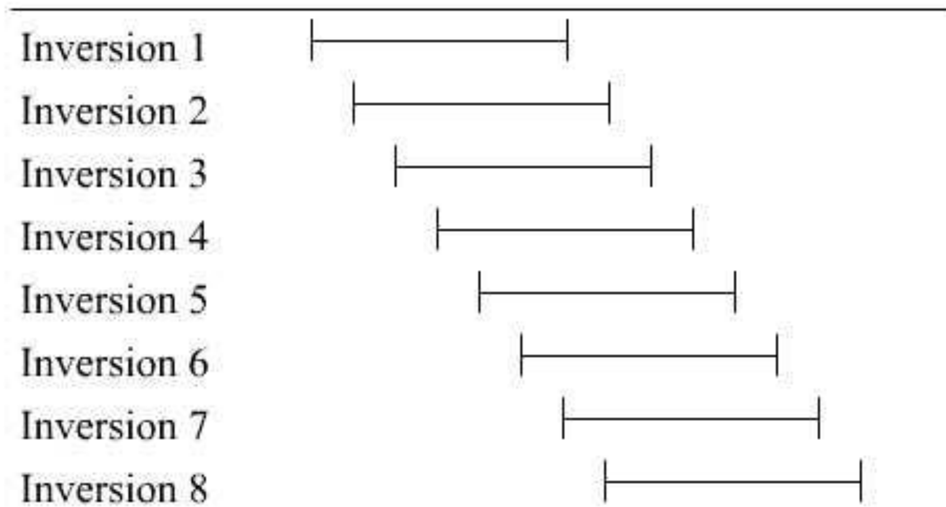

Figure 12 - This figure demonstrates how the parallelization already implemented works. Essentially, since each inversion is an independent separate process, they can run concurrently. Inversion 1 is started and takes X seconds to complete. Before inversion 1 finishes, inversion 2, 3, and so on can all be started and run at the same time. Start time staggering, ideally ~1 second on a 1 Hz network, is necessary to not overload the system. Each inversion takes almost 100% of a computational unit, so there needs to be more than X cores for the system to work.

The stacking depends both on this system and the computer system design. Each inversion tends to require almost all of a core to itself, so for each expected inversion on a 1 Hz network (how many seconds a single inversion takes), a dedicated core is required. So, running a 6 second inversion on 8-cores would be fine, on 4-cores with hyper-threading will start to cause issues, and on 4-cores the system will bottleneck. Another thing to consider is how long the inversion takes in the worst-case scenario. An inversion where all offsets are zero will run faster than an inversion where

the offsets are all non-zero. So even running an inversion that takes 6 seconds when there is the occasional random offset may still run into issues on an 8-core system when there is a large event. So, designing the computer system and picking the amount of subfaults to invert across needs to be done carefully, and should err very conservatively if there are any questions about reliability in typical large events.

In the inversion process, the data is organized correctly. Once organized, a NNLS inversion is run to get the slip distribution. The resulting best-fit model is multiplied by the sub-input matrix, SIM, to get the calculated offset matrix, COM. The moment and moment magnitude are calculated based on the best-fit slip distribution. Finally, all the data is packed up and passed along to SlipWriter.

X6 - SlipWriter

SlipWriter takes the solutions from TVLiveSlip and organizes the data.

First, it takes the data and repacks it into the format that the RabbitMQ needs for output. It checks the time and if the time is later than the current latest time received, it updates the current latest time. Then, it checks the magnitude of the event and will send an email to someone if a large enough magnitude offset is detected, greater than SWMagnitude in the Config file.

Secondly, it looks at the current latest time and, with a delay setting in the Config file, SWDelay, it determines if any data needs to be sent to the RabbitMQ. The delay is currently unnecessary, and it does not work now. Anytime the value is not 0, the whole process does not work. If any data does need to be sent to the RabbitMQ, then it sends the data as appropriate, which currently is as soon as it is received.

This process also had write-to-disk capabilities. They were removed but were useful in the past. This process also handles sending out emails during large events. Organizing it this way makes it easier since the inversions do not have to handle figuring out when the last email was sent and if they should send another. Also, each inversion does not have to open a separate connection to the RabbitMQ to pass its data out.

X7 - RMQtoMDB

This is a standalone process. It takes the data output from the RabbitMQ, repacks it, and sends it to the MongoDB. This two-step method between the end of SlipWriter and the database the aggregator pulls from before being displayed in the GPS cockpit is to allow other systems, such as the future tsunami estimation system, to hook into and acquire the slip distributions and other offset information. The MongoDB is a responsive system sending only the most recent information on request. This can cause systems down the pipeline to potentially miss data. The RabbitMQ is an active system, broadcasting all data as it comes in. This allows downstream processes to get all data instead of some.

This program uses a wildcard to capture all data coming through the RabbitMQ, so there only needs to be one instance running. Therefore, this system is not directly hooked into the TULiveFilter communication and control system.

XI - Behavior

Because of how complex the system and its goals are, it tends to exhibit some behaviors that are odd to those unfamiliar.

When the whole system loses data, it will pause. But, because of the delay in DataWriter, the last output from the system will be about 15 seconds before the system went down. When the system starts back up, the stored 15 seconds will be passed through and then a time jump will occur to the current time.

If an anomalous measurement is detected, the filter will pause for a short duration. This, on a 1 Hz network, this corresponds to the number of measurements to wait before declaring that an offset has been detected, MesWait. For example, if MesWait is set to 5, then once a seismic wave reaches and offsets the station, the station will wait about 5 seconds before beginning to process data. This only affects when the first waves reach the site. Once the filter switches into ongoing event mode then data will be processed as it comes in. If DWDelay is 15, the first site will take about 20 (realistically ~22.5 $\pm$ 2.5) seconds plus the time to invert it and other processing time before any indication that an event has happened appears.

When an event does occur, stations right at the cusp of detection may behave very oddly. They may appear to flicker in and out. This is because the offset is near the lower detection limit of the filters. This results in the filters receiving a few anomalous

offsets followed by a few normal. Therefore, all the measurements are processed as normal with the filter switching between a possible event mode and a detection enabled mode without flipping into an ongoing event mode. This causes the flicker and may last for a few minutes. This will not be observed in GPS Cockpit if DWDelay is significantly greater than MesWait (i.e. DWDelay = 15, MesWait = 5).

The system also gets more accurate as the magnitude of the event increases. This is due to the signal to noise ratio being better in large-scale events. Another factor is that for small events, the first few measurements of the event may pull the actual baseline into the offset a bit before the offset deviates enough to generate anomalous measurements. This results in reduced overall offset estimations. As the magnitude of the event increases, the overall offset increases and the small tugs into the offset before anomalous measurements are detected become less significant.

One last reason is because as the magnitude of the event increases, more sites from a wider area will detect offsets.

Another thing to consider is the random offsets detected by the system. These have a variety of sources. Since each filter is independent, these will not usually have a significant impact on large events, though noting them during an event is important. Regardless of the specific situation, a false offset will slightly skew a slip solution, but not significantly for large events.

Because of the initial convergence period, the system will not to be able to detect events that occur right after the system starts up. The duration of the convergence period depends on EQPause in the Config file.

Also, the system has a maximum offset possible between one time-step and the next or the measurement is ignored. This can cause issues at times. If an antenna gets replaced or the point positioning system gets restarted, the system may have trouble if the perceived offset is too great. Essentially, the data gets ignored and the filter will stagnate at the last measurement that it considered to be good. This can be overcome by temporarily adjusting the settings correctly but may mean having to completely restart the system so those filters calibrate to the new measurements.

There is no hard lower detection limit for the system. Since the measurements are peppered with white and colored noise, the lower detection limit is a probability distribution. This distribution depends on the covariances of both the filters and the measurements, P and R. Also, MesWait and EQThres affect the detection limits. Determining the detection probability distribution is complex, implementing it is difficult, and the actual distribution is of little use in real-time systems operating as intended. Furthermore, the lower magnitude events are more accurately measured using other seismic methods.

In the GPS cockpit, when a site detects an offset, a few things can happen. If the offset is in the opposite direction to the expected motion, the NNLS will keep the system from moving in the opposite direction and the system will show a blue arrow, observed offset, but no significant red arrows, calculated offset. When an offset is detected in the historically observed the direction, then a blue arrow and, depending on the NNLS slip distribution, a red arrow, calculated offset, may also show up. When the site is near a boundary between faults, then two red arrows may appear. This is because the site is being run independently by two different systems, so it is getting values from each system. For the observed offset, since the Kalman filters in both instances are receiving the same data, there will be some minor differences based on when each filter began and there are two blue arrows, but they normally will appear as one arrow. But, because the faults have different motions, that same offset may be interpreted into different calculated offsets based on the fault mechanics on both faults resulting in the two distinct red arrows. This only happens if the site detects an offset and both faults can generate realistic slip distributions based off it and surrounding data.

XII - Cascadia Implementation

The system has already been implemented in Cascadia. It uses a 20x10 subfault model. There are 20 subfaults along strike and 10 subfaults along dip creating 200 overall. This was done for performance reasons. The system that it currently runs on is not powerful enough to reliably run more subfaults.

As such, the longest time the system has been running continuously in a reasonably recent build is about 46 days. During that time, the system had two instances where it detected events larger than 9.0, both were false. Neither of these events were recorded but were likely due to cycle slips.

Cascadia was chosen as the main fault for development for a few reasons. First, CWU is in the region impacted by a large fault movement. Also, many of the sites being processed specifically by CWU are in the Pacific Northwest. Third, Cascadia represents the largest seismic hazard in the US. While the magnitude potentials are eclipsed by the Aleutian trench, the location relative to the major cities of Seattle, Portland, and Vancouver mean a large earthquake would cause significant damage.

XIII - South San Andreas Implementation

The system has been implemented for the south San Andreas Fault. This was done to test a few things about the expandability of the system. First, testing if the system can work on a second fault since most development and testing had been done solely on Cascadia. Secondly, it tests if the system can work on a strike-slip fault and exposes any potential problems with the general methods already implemented. Thirdly, this tests the ability to run multiple instances simultaneously.

Using this model has shown no significant issues running the program on different faults. But, it has highlighted a few issues concerning running multiple instances at the same time. These issues are talked about in the Future Work section.

XIV - Future Work

Slip-rate distribution, in the current system, cannot be calculated accurately. This system is designed to determine the slip as quickly as possible. Therefore, it does no travel time correction for the seismic waves. This results in each filter representing offsets from slip at different times. This affects the accuracy of the system, but not significantly enough to affect the ability to use this system for rapid earthquake assessment. The most accurate slip distributions are produced 5-15 minutes after a large event. This is because the system must wait for the total slip to finish, usually under 8 minutes (Ishii, Shearer, Houston, & Vidale, 2005), and then the seismic wave travel time, usually under 5 minutes. These issues make simply differencing the slip distributions to get the slip-rate distribution inaccurate. But, as designed, the system will produce estimates about 30 seconds after the event is first detected. Furthermore, it will produce estimates close to the final magnitude within a few minutes. The other issue is the influence of noise on the system altering the final results and making a perfect result impossible to obtain.

The seismic wave travel time can be automatically corrected in the code. The Kalman filter keeps track of how many measurements have passed since the offset was first detected. By passing this into the DataWriter, the data can be reorganized based on how many measurements have passed instead of based on time. A simple data cleaning can also be run; if a filter detects an offset much earlier or later than other filters, it can be removed. The reorganized data can then be passed to the inversion and

it will output a more accurate slip distribution. This slip distribution can then be differenced to get an approximate estimate of the slip-rate distribution through time.

There are two problems with this method which make it unsuitable based on the design goals. First, it requires all the filters that detect an offset to report the first detected offset before any data can be processed. This adds about a five-minute delay to the system. This completely defeats the goal of a rapid earthquake assessment system. Second, the system can be designed to do both, but the inversions would have to be doubled by running a second instance of DataWriter, TVLiveSlip, and SlipWriter. Because the inversions are the most intensive processes in the entire system, this effectively doubles the computational requirements of the system. The system would not need to run the second inversions when no offsets are detected. But this means the second inversions would kick on about five minutes after the event, when the system is most needed. Because the inversions process much slower during actual large events than during smaller or false events, the system would slow down significantly during events that it was designed to rapidly assess and potentially drop out of real-time. Therefore, the computational increase both compromises the ability of the system to function when needed and forces less subfaults or sites to be processed to reduce the computational load to something the system can handle.

The other main thing that still needs to be done is to extend the system to run multiple instances within the same folder. In the current version each instance must be

run in a separate folder. The system uses a hardcoded subfault file, Config file, and Run log file. To extend the program, the Config file needs to be defined on the command line, the Run log needs to either be command line or removed, and the subfault file should be command line.

The Run log file needs to be defined separately for each instance running to avoid overwriting the file and, in the case of 100's of instances running, to avoid output collisions from occurring. The easier option is to completely remove the Run logs.

Because of the creation of the DUMMY site early in the initialization of the TVLiveSlip file, the subfault file needs to be known early on. One method to solve this is to simply replace the subfault file outside the program for the newest instance. The other way is to pass the specified file into the program in the command line and pass that through to TVLiveSlip before the initialization of DUMMY site.

A better method to improve this is to completely remove the DUMMY site. This would require removing all code referencing it directly. Another thing that would have to change for this implementation is the code for adding sites to the subfault offset matrix. The DUMMY site maintains the dimensions so a standard vstack command can be used to add new subfault offset lines to the overall matrix. The system would have to switch to checking the dimensions of the subfault offset matrix and if they are zero by zero then just replace the matrix rather than using the vstack command.

There is another issue with this, regardless of whether a separate folder or single folder option is used. This revolves around when one of the systems goes haywire, which may happen from time to time. Determining specifically which process needs to be killed is nearly impossible to do. To get around this, a setting in the Config file to kill the entire process is necessary. The Config files are more easily identifiable than the specific processes. There is a setting in the Config file already to do this, but it is not completely implemented. To fully implement this setting smoothly into the system would require large portions of every piece of code to be rewritten.

The system should also be updated to use Euler poles instead of direction of convergence (Moores & Twiss, 1995).For smaller faults such as Cascadia, the Middle America trench, etc., the use of Euler poles would not create any significant changes to the results. But, for larger faults, such as South American trench, the Japanese trench and the Aleutian trench, the curvature of the Earth effects how the plates are coming together. So, the direction of convergence does not work well on those faults.

The direction of convergence and Euler pole methods also need to be altered to better account for strike-slip faults. Basically, instead of using a direction of convergence or Euler pole the system should be using a specific value denoting pure strike-slip, thereby keeping the system from trying to act as a dip-slip fault and generating unrealistic scenarios.

Another thing that may be worth doing in the future is to implement MAGMA (Tomov, 2010). MAGMA is the latest iteration of the linear algebra package LAPACK. It allows linear algebra problems to be parallelized across multiple cores, CPUs, and GPUs. By switching the NNLS inversion to MAGMA, it would allow more complex inversions to be run. This could include using more sites or more subfaults in the inversion. Though, this only makes sense for very large GPS networks and large computers. The current benefits from implementing this are minimal, with the current inversion parallelization method being able to handle all current data rapidly.

Further testing and fine-tuning of the system is necessary. The system has a lot of settings in the Config file. For stability and reliability testing, the system has mainly been running on unrealistic settings that would be impractical in the real world. The system, therefore, needs to be tested further on realistic synthetics to figure out specific settings for the system. This needs to be done for each specific instance of the program because subfault size will change and requirements for relevancy, offset detection, and various other settings need to be changed accordingly.

Work extending the expandability of the system also needs to be done. Pulling more hard-coded settings out of the code and adding them to Config files would help. Some of the settings that could still be pulled out are connection settings and some of the hard-coded delays.

XV - Conclusion

This project extends the monitoring and assessment of large earthquakes. By using smaller processes, it extends the capacity and ability of this system to rapidly assess large earthquakes while reducing the computation spikes resulting from them. By adding this system into the current seismic, teleseismic, and deep ocean buoy seismic and hazard estimation systems in place, the overall ability of various organizations to rapidly assess earthquakes and their hazards will be improved significantly.

Further, this system brings into being what individuals in the community have been saying for years, a GPS-based seismic system has advantages in speed and accuracy in the evaluation of large earthquakes. Further work needs to be done concerning extending the system around the Pacific, adding in tsunami run up estimations, and further testing and fine tuning the system on a fault by fault basis. At the current moment, the system acts as an applicable, functioning platform and bluepring for future work.

XVI - Works Cited

Bock, Y. (2013, March). *Feasibility Study using Real-time GPS/Seismic Displacements to Improve Disaster Management and Decisions Pertaining to Rapid Assessment of Structural Damage.* Retrieved from Scripps Instutute of Oceanography: https://scripps.ucsd.edu/research/proposals/feasibility-study-using-real-time-gps/seismic-displacements-improve-disaster

Grapenthin, R., Johanson , I. A., & Allen, R. M. (2014). Operational real-time GPS-enhanced earthquake early warning. *Journal of Geophysical Research: Solid Earth, 119*(10), 7944-7965.

Hayes, G. P., Earle, P. S., Benz, H. M., Wald, D. J., & Briggs, R. W. (2011). The USGS-NEIC response to the 2011/03/11 Mw. 9.0 Tohoku earthquake. *Seismological Society of America.* Seismological Society of America.

Ishii, M., Shearer, P. M., Houston, H., & Vidale, J. E. (2005). Extent, duration and speed of the 2004 Sumatra-Andaman earthquake imaged by the Hi-Net array. *Nature, 435*(16), 933-936.

Kawamoto, S., Ohta, Y., Hiyama, Y., Todoriki, M., Nishimura, T., Furuya, T., et al. (2017). REGARD: A new GNSS-based real-time finite fault modeling system for GEONET. *Journal of Geophysical Research: Solid Earth, 122*(2), 1324-1349.

Larson, K. M., Bodin, P., & Gomberg, J. (2003). Using 1-Hz GPS Data to Measure Deformations Caused by the Denali Fault Earthquake. *SCIENCE, 300*(5624), 1421-1424.

Lawson, C., & Hanson, R. J. (1987). Solving Least Squares. *SIAM*.

Matthews, M., & Segall, P. (1997). Time dependent inversion of geodetic data. *Geophysical Research Letters, 102*(B10), 22391-22409.

Melgar, D., Bock, Y., Cowell, B. W., & Haase, J. S. (2013). Rapid modeling of the 2011 Mw 9.0 Tohoku-oki earthquake with seismogeodesy. *Geophysical Research Letters, 40*(12), 2963-2968.

Moores, E., & Twiss, R. (1995). Tectonics. *W.H. Freeman and Company*, 50-69.

Okada, Y. (1992). Internal deformation due to shear and tensile faults in a half-space,. *Bulletin of the Seismological Society of America, 82*(2), 1018-1040.

Santillan, V. M. (n.d.). Personal Communication. (J. Senko, Interviewer)

Song, Y. T. (2014, October 1). *GPS-Aided and DART-Ensured Real-Time (GADER) Tsunami Early Detection System.* Retrieved from NASA Applied Sciences Program: https://appliedsciences.nasa.gov/content/11-disaster11d-0021

Tomov, S. D. (2010). Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing, 36*(5-6), 232-240.

Wright, T., Houlié, N., Hildyard, M., & Iwabuchi, T. (2012). Real-time, reliable magnitudes for large earthquakes from 1 Hz GPS precise point positioning: The 2011 Tohoku-Oki (Japan) earthquake. *Geophysical Research Letters, 39*(12).

Zarchan, P., & Mussoff, H. (2005). *Fundamentals Of Kalman Filtering: A Practical Approach.* American Institute of Aeronautics and Astronautics.

XVII - Appendixes

Appendix A – Config File Settings

Systemwide Settings	
Run	Currently not implemented. Ideally kills the system (Boolean).
Email	Who to send an email to when the system needs to (email string).

TULiveFilter settings	
ConfigCheck	How frequently to check the Config file for changes (seconds).

DataRouter settings	
SendData	Whether DataRouter passes data through to the system. Useful to test and debug issues concerning filters being turn on and off (Boolean).

Kalman settings	
EQPause	How long to freeze the offset detection ability after an offset is detected (measurements).
EQThres	How big an anomalous measurement must be before triggering the offset detection limit (multiple of the standard deviation of the measurement covariance) (float).
MesWait	Number of consecutive anomalous measurements before triggering the offset detection (measurements).
DieTime	How many seconds to wait since the last data was received before turning the filter off (seconds).
MinR	Default value for the covariance matrix of the measurements if the value comes through as 0. Needs to be greater than 0 or the system crashes (float).
Offset	Whether to add a synthetic offset into the system (Boolean).
MaxOffset	Maximum amount of offset between two measurements, reduces cycle slip impact (meters).

DataWriter settings

DWDelay	How many time-steps to wait before the DataWriter sends the data to the inverter (time-steps).
SendFreq	How many time-steps to skip between sending a time-step to the inverter (time-steps).
TVLiveSlip settings	
Alpha	Smoothing parameter for the inverter (float).
MaxChildren	How many different inversions can be spawned at any instant (value).
InvKillTime	How long to wait before the inversion gets terminated, prevents the system from stalling (seconds).
Label	String prefixed to the output label, viewable in the GPS cockpit as solution (string).
Model	What the subfault model is called in the GPS cockpit (string).
Tag	What the tag for the data in the MongoDB, typically current, but in the future may change as historical models are run (string).
TVLiveSlip settings only read once	
MinOffset	What the offset needs to be for the site is deemed relevant to the model (meters).
RangeThres	Maximum percentage of subfaults that the site is deemed irrelevant for before being deemed irrelevant to the whole system (percentage) (0-1).
Convergence	Direction on convergence between the plates based on the footwall, needs to be updated to account for Euler poles (degrees).
StrikeSlip	Whether the fault is strike slip or not (Boolean).
SlipWriter settings	
SWDelay	Delay between receiving a solution and outputting it to the MongoDB, can be left at 0 (measurements).
SWMagnitude	How large an event magnitude must be before sending an email (moment magnitude).

SWDuration	How long after a SWMagnitude email is sent before a second email can be sent (minutes).
Email	Who to email in the case of a large event
Cycler settings	
Cycle	Whether to run the program that turns the SendData value between True and False repeatedly to test stability (Boolean).

Appendix B – Variables

M_k	Covariance matrix from the previous Kalman filter state
Φ_k	Matrix describing how the Kalman filter system evolves from measurement to measurement
P_k	Covariance matrix for the current Kalman filter state
Q_k	Process noise matrix for the Kalman filter
K_k	Gain matrix for the Kalman filter
H	Measurement matrix describing how what is measured is represented in the Kalman filtering system
R_k	Covariance matrix for the measurements
I	Identity matrix
k	Current time-step of the filter system
Res_k	Residual matrix, the difference between the predicted measurement and the current measurement
X_k	Measurement matrix
S_T	Current predicted measurement ($S_1 + S_2$)
S_1	The state that the Kalman filter is filtering for
S_2	The baseline state for the Kalman filter system
H(x)	Heaviside function of x
Ψ	State reset matrix
Ξ	Filter covariance reset matrix
thres	Earthquake threshold value, EQThres(Config file) * standard deviation of the current measurement
SM	Smoothing matrix used during the inversion to constrain the inversion
SIM	Sub-input matrix used to describe how each site should move based on slip
CL	Correlation list to relate matrix rows to GPS sites in TVLiveSlip
alpha	Smoothing variable
adjsubfaultslip	Potentially remove
Y	Subfaults moving adjacent to the current subfault
subfaultslip	Matrix defining how slip is observed
DOM	Detected offset matrix
SLIP	Calculated slip distribution
COM	Calculated offset matrix

Appendix C - Proof

Both of these proofs are only for a 2 state system, i.e. N, E. The actual system used was 3 state, N,E,V. The first proof is the code as used in this thesis.

$$M_k = \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix}; H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \; H^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \; S_T = \begin{bmatrix} S_1 & 0 \\ 0 & S_3 \end{bmatrix}; \; \Phi_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \; R_k = \begin{bmatrix} R_y & 0 \\ 0 & R_z \end{bmatrix}; \; Q_k = \begin{bmatrix} Q_y & 0 \\ 0 & Q_z \end{bmatrix}; Mea_k = \begin{bmatrix} Mea_y & 0 \\ 0 & Mea_z \end{bmatrix}; S2 = \begin{bmatrix} S_2 & 0 \\ 0 & S_4 \end{bmatrix}$$

$$M_k = \Phi_k \, P_{k-1} \, \Phi_k^T + Q_k$$
$$K_k = M_k \, H^T \, (H \, M_k \, H^T + R_k)^{-1}$$
$$P_k = (I - K_k \, H) \, M_k$$

$$Res_k = Mea_k - H \, \Phi_k \, S_{Tk} - S2$$
$$S_{T(k+1)} = \Phi_k \, S_{Tk} + K_k \, Res_k$$

$$S2 = S2 + S_{Tk}$$
$$S_{Tk} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$
$$P_k = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}$$

Riccati EQ Proof

$$M_k = P_{k-1} + Q_k$$
$$K_k = M_k \, (M_k + R_k)^{-1}$$
$$P_k = (I - K_k) \, M_k$$

$$K_k = \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix} \left(\begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix} + \begin{bmatrix} R_y & 0 \\ 0 & R_z \end{bmatrix} \right)^{-1}$$
$$K_k = \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix} \left(\begin{bmatrix} M_y + R_y & 0 \\ 0 & M_z + R_z \end{bmatrix} \right)^{-1}$$
$$K_k = \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix} \begin{bmatrix} \frac{1}{(M_y + R_y)} & 0 \\ 0 & \frac{1}{(M_z + R_z)} \end{bmatrix}$$
$$K_k = \begin{bmatrix} \frac{M_y}{(M_y + R_y)} & 0 \\ 0 & \frac{M_z}{(M_z + R_z)} \end{bmatrix}$$

$$\delta_y = \frac{M_y}{\left(M_y + R_y\right)};\ \delta_z = \frac{M_z}{(\ M_z + R_z\ )}$$
$$K_k = \begin{bmatrix} \delta_y & 0 \\ 0 & \delta_z \end{bmatrix}$$

$$P_k = (\ I - \ K_k\ )\ M_k$$
$$P_k = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} \delta_y & 0 \\ 0 & \delta_z \end{bmatrix} \right) \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix}$$
$$P_k = \begin{bmatrix} 1 - \delta_y & 0 \\ 0 & 1 - \delta_z \end{bmatrix} \begin{bmatrix} M_y & 0 \\ 0 & M_z \end{bmatrix}$$
$$P_k = \begin{bmatrix} M_y\ (\ 1 - \delta_y\ ) & 0 \\ 0 & M_z\ (\ 1 - \delta_z\ ) \end{bmatrix}$$
$$\varepsilon_y = \ M_y\left(1 - \delta_y\right);\ \varepsilon_z = \ M_z(1 - \delta_z)$$
$$P_k = \begin{bmatrix} \varepsilon_y & 0 \\ 0 & \varepsilon_z \end{bmatrix}$$

$$M_k = \ P_{k-1} + \ Q_k$$
$$M_k = \begin{bmatrix} \varepsilon_y & 0 \\ 0 & \varepsilon_z \end{bmatrix} + \begin{bmatrix} Q_y & 0 \\ 0 & Q_z \end{bmatrix}$$
$$M_k = \begin{bmatrix} \varepsilon_y + Q_y & 0 \\ 0 & \varepsilon_z + Q_z \end{bmatrix}$$

State Proof

$$Res_k = \ Mea_k - H\ \Phi_k\ S_{Tk} - S2$$
$$S_{T(k+1)} = \ \Phi_k\ S_{Tk} + \ K_k\ Res_k$$

$$Res_k = \ Mea_k - \ S_{Tk} - S2$$
$$S_{T(k+1)} = \ S_{Tk} + \ K_k\ Res_k$$

$$Res_k = \ Mea_k - \ S_{Tk} - S2$$
$$Res_k = \begin{bmatrix} Mea_y & 0 \\ 0 & Mea_z \end{bmatrix} - \begin{bmatrix} S_1 & 0 \\ 0 & S_3 \end{bmatrix} - \begin{bmatrix} S_2 & 0 \\ 0 & S_4 \end{bmatrix}$$
$$Res_k = \begin{bmatrix} Mea_y - S_1 - S_2 & 0 \\ 0 & Mea_z - S_3 - S_4 \end{bmatrix}$$
$$Res_y = \ Mea_y - S_1 - S_2;\ Res_z = \ Mea_z - S_3 - S_4$$
$$Res_k = \begin{bmatrix} Res_y & 0 \\ 0 & Res_z \end{bmatrix}$$

$$S_{T(k+1)} = \ S_{Tk} + \ K_k\ Res_k$$
$$S_{T(k+1)} = \begin{bmatrix} S_1 & 0 \\ 0 & S_3 \end{bmatrix} + \begin{bmatrix} \delta_y & 0 \\ 0 & \delta_z \end{bmatrix} \begin{bmatrix} Res_y & 0 \\ 0 & Res_z \end{bmatrix}$$

$$S_{T(k+1)} = \begin{bmatrix} S_1 & 0 \\ 0 & S_3 \end{bmatrix} + \begin{bmatrix} \delta_y \ Res_y & 0 \\ 0 & \delta_z \ Res_z \end{bmatrix}$$
$$S_{T(k+1)} = \begin{bmatrix} S_1 + \delta_y \ Res_y & 0 \\ 0 & S_3 + \delta_z \ Res_z \end{bmatrix}$$

Reset Proof

$$S2 = S2 + S_{Tk}$$
$$S_{Tk} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$
$$P_k = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}$$

$$S2 = S2 + S_{Tk}$$
$$S2 = \begin{bmatrix} S_2 & 0 \\ 0 & S_4 \end{bmatrix} + \begin{bmatrix} S_1 & 0 \\ 0 & S_3 \end{bmatrix}$$
$$S2 = \begin{bmatrix} S_2 + S_1 & 0 \\ 0 & S_4 + S_3 \end{bmatrix}$$
$$S_{Tk} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$
$$P_k = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}$$

The second proof is an optimized version that incorporates everything into the Kalman filter.

$$M_k = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}; H = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}; \ H^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}; \ S_T = \begin{bmatrix} S_1 & 0 \\ S_2 & 0 \\ 0 & S_3 \\ 0 & S_4 \end{bmatrix};$$
$$\Phi_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \ R_k = \begin{bmatrix} R_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & R_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}; \ Q_k = \begin{bmatrix} Q_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & Q_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix};$$
$$Mea_k = \begin{bmatrix} Mea_y & 0 \\ 0 & 0 \\ 0 & Mea_z \\ 0 & 0 \end{bmatrix}; \ \Psi = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}; \ \Xi = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Riccati Equations

$$M_k = \Phi_k P_{k-1} \Phi_k^T + Q_k$$
$$K_k = M_k H^T (H M_k H^T + R_k)^{-1}$$
$$P_k = (I - K_k H) M_k$$

State Update Equations

$$Res_k = Mea_k - H \Phi_k S_{Tk}$$
$$S_{T(k+1)} = \Phi_k S_{Tk} + K_k Res_k$$

State Reset Equations

$$S_{Tk} = \Psi S_{Tk}$$
$$P_k = \Xi$$

Proof

Assume M, run through Riccati Equations and show M_{k+1} is of the same form.
Show $S_{T(k+1)}$ is the same form as S_{Tk}.
Show reset equations work

Show $M_{(k+1)}$ is same form as M_k.

$$M_k = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$K_k = M_k H^T (H M_k H^T + R_k)^{-1}$$

$$H M_k = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = M_k$$

$$M_k H^T = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = M_k$$

$$K_k = M_k (M_k + R_k)^{-1}$$

$$K_k = M_k \left(\begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} R_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & R_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)^{-1}$$

$$K_k = M_k \left(\begin{bmatrix} M_y + R_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z + R_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)^{-1}$$

$$K_k = M_k \begin{bmatrix} \frac{1}{(M_y + R_y)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{(M_z + R_z)} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$K_k = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{(M_y + R_y)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{(M_z + R_z)} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$K_k = \begin{bmatrix} \frac{M_y}{(M_y + R_y)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{M_z}{(M_z + R_z)} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\delta_y = \frac{M_y}{(M_y + R_y)}; \ \delta_z = \frac{M_z}{(M_z + R_z)}$$

$$K_k = \begin{bmatrix} \delta_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \delta_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$P_k = (I - K_k H) M_k$$

$$K_k H = \begin{bmatrix} \delta_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \delta_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \delta_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \delta_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = K_k$$

$$P_k = (I - K_k) M_k$$

$$P_k = \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} \delta_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \delta_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right) M_k$$

$$P_k = \begin{bmatrix} 1 - \delta_y & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 - \delta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$P_k = \begin{bmatrix} M_y(1-\delta_y) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z(1-\delta_z) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\varepsilon_y = M_y\left(1-\delta_y\right);\ \varepsilon_z = M_z(1-\delta_z)$$

$$P_k = \begin{bmatrix} \varepsilon_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M_{k+1} = \Phi_k\, P_k\, \Phi_k^T + Q_{k+1}$$

$$\Phi_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{k+1} = P_k + Q_{k+1}$$

$$M_{k+1} = \begin{bmatrix} \varepsilon_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} Q_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & Q_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M_{k+1} = \begin{bmatrix} \varepsilon_y + Q_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon_z + Q_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M_k = \begin{bmatrix} M_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & M_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

M_{k+1} is the same form as M_k.

Show $S_{T(k+1)}$ is of same form as S_{Tk}.

$$Res_k = Mea_k - H\, \Phi_k\, S_{Tk}$$

$$S_{T(k+1)} = \Phi_k\, S_{Tk} + K_k\, Res_k$$

$$Res_k = Mea_k - H\, \Phi_k\, S_{Tk}$$

$$\Phi_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Res_k = Mea_k - H\,S_{Tk}$$

$$H\,S_{Tk} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} S_1 & 0 \\ S_2 & 0 \\ 0 & S_3 \\ 0 & S_4 \end{bmatrix} = \begin{bmatrix} S_1 + S_2 & 0 \\ 0 & 0 \\ 0 & S_3 + S_4 \\ 0 & 0 \end{bmatrix}$$

$$Res_k = \begin{bmatrix} Mea_y & 0 \\ 0 & 0 \\ 0 & Mea_z \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} S_1 + S_2 & 0 \\ 0 & 0 \\ 0 & S_3 + S_4 \\ 0 & 0 \end{bmatrix}$$

$$Res_k = \begin{bmatrix} Mea_y - S_1 - S_2 & 0 \\ 0 & 0 \\ 0 & Mea_z - S_3 - S_4 \\ 0 & 0 \end{bmatrix}$$

$$Res_y = Mea_y - S_1 - S_2;\ Res_z = Mea_z - S_3 - S_4$$

$$Res_k = \begin{bmatrix} Res_y & 0 \\ 0 & 0 \\ 0 & Res_z \\ 0 & 0 \end{bmatrix}$$

$$S_{T(k+1)} = \Phi_k\,S_{Tk} + K_k\,Res_k$$

$$\Phi_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_{T(k+1)} = S_{Tk} + K_k\,Res_k$$

$$K_k\,Res_k = \begin{bmatrix} \delta_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \delta_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} Res_y & 0 \\ 0 & 0 \\ 0 & Res_z \\ 0 & 0 \end{bmatrix}$$

$$K_k\,Res_k = \begin{bmatrix} \delta_y\,Res_y & 0 \\ 0 & 0 \\ 0 & \delta_z\,Res_z \\ 0 & 0 \end{bmatrix}$$

$$S_{T(k+1)} = S_{Tk} + K_k\,Res_k$$

$$S_{T(k+1)} = \begin{bmatrix} S_1 & 0 \\ S_2 & 0 \\ 0 & S_3 \\ 0 & S_4 \end{bmatrix} + \begin{bmatrix} \delta_y\,Res_y & 0 \\ 0 & 0 \\ 0 & \delta_z\,Res_z \\ 0 & 0 \end{bmatrix}$$

$$S_{T(k+1)} = \begin{bmatrix} S_1 + \delta_y\ Res_y & 0 \\ S_2 & 0 \\ 0 & S_3 + \delta_z\ Res_z \\ 0 & S_4 \end{bmatrix}$$

$S_{T(k+1)}$ is the same form as S_{Tk}. Only S_1 and S_3 are updated, S_2 and S_4 stay constant.

Show reset equations work.

$$S_{Tk} = \Psi\ S_{Tk}$$
$$P_k = \Xi$$

$$S_{Tk} = \Psi\ S_{Tk}$$

$$\Psi = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\mathrm{S_{Tk}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{S_1} & 0 \\ \mathrm{S_2} & 0 \\ 0 & \mathrm{S_3} \\ 0 & \mathrm{S_4} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ \mathrm{S_1 + S_2} & 0 \\ 0 & 0 \\ 0 & \mathrm{S_3 + S_4} \end{bmatrix}$$

$$\mathrm{P_k} = \Xi$$

$$\Xi = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathrm{P_k} = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Equations set S_2 and S_4 to current total state, S_1 and S_3 to zero, and filter covariance matrix is reset. All matrices are still in correct form.

Appendix D – Code

Appendix D1 - TULiveFilter.py 77
Appendix D2 - DataRouter.py 86
Appendix D3 - Kalman.py... 91
Appendix D4 - DataWriter.py... 105
Appendix D5 - TVLiveSlip.py 108
Appendix D6 - SlipWriter.py 123
Appendix D7 - RMQtoMDB.py 128
Appendix D8 - Config 130
Appendix D9 - Cycle.py 132

Appendix D1 - TULiveFilter.py

```
#! /usr/bin/env python
# TULiveFilter, starts other processes, checks config file, stores variables


# imports
from Kalman import Kalman
from DataRouter import DataRouter
from DataWriter import DataWriter
from SlipWriter import SlipWriter
from TVLiveSlip import TVLiveSlip
import multiprocessing as mp
from multiprocessing import Pipe, Process, reduction, Lock, Queue
from multiprocessing.reduction import reduce_connection
import threading as thr
import pickle
import time
import json
import subprocess as sub
from subprocess import PIPE, Popen
from datetime import datetime as dt
from datetime import timedelta as td
import os
import difflib
import logging


# set up Run.log for monitoring code
p = sub.Popen( [ 'rm', 'Run.log' ] )
p.wait()

logging.basicConfig( filename='Run.log', level=logging.DEBUG, format='%(asctime)s -
%(levelname)s   %(message)s' )

# initialize some variables
Run = True
lock = thr.Lock()
que = Queue()
listLock = thr.Lock()
FilterSettings = {}
global email
email = ""
```

```
# pickle a connection for sending through a pipe
def _pickle_connection( connection ):
	return reduce_connection( connection )

# check the configuration file for changes
def _ConfigCheck():
	odate = 0
	ConfigCheck = 60.
	while( Run == True ):
		# print "Checking Config file"
		t = os.path.getmtime( "Config" )
		ndate = dt.fromtimestamp( t )

		if( ndate != odate ):
			with open( "Config", 'r' ) as f:
				flines = f.readlines()
			with open( ".Config", 'r' ) as g:
				glines = g.readlines()


			d = difflib.Differ()
			diff = d.compare( glines, flines )
			newst = [ line[1:].split() for line in diff if line[0] == '+' ]

			for new in newst:
				if not new:
					pass
				elif( new[0][0] == "#" ):
					pass
				elif( new[0] == "ConfigCheck" ):
					ConfigCheck = float( new[2] )
				elif( new[0] == "SendData" ):
					if( new[2] == "True" ):
						DataRouterPipe.send( [ new[0], True ] )
					elif( new[2] == "False" ):
						DataRouterPipe.send( [ new[0], False] )
				elif( new[0] == "EQPause" ):
					lock.acquire()
					for filt in RunningList:
						RunningList[filt][1].send( [ "EQPause", float(
new[2] ) ] )
					lock.release()
					FilterSettings['EQPause'] = float( new[2] )
				elif( new[0] == "EQThres" ):
```

```
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "EQThres", float(
new[2] ) ] )
                    lock.release()
                    FilterSettings['EQThres'] = float( new[2] )
                elif( new[0] == "MesWait" ):
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "MesWait", float(
new[2] ) ] )
                    lock.release()
                    FilterSettings['MesWait'] = float( new[2] )
                elif( new[0] == "DieTime" ):
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "DieTime", float(
new[2] ) ] )
                    lock.release()
                    FilterSettings['DieTime'] = float( new[2] )
                elif( new[0] == "MinR" ):
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "MinR", float(
new[2] ) ] )
                    lock.release()
                    FilterSettings['MinR'] = float( new[2] )
                elif( new[0] == "Offset" ):
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "Offset", new[2] ]
)
                    lock.release()
                    FilterSettings['Offset'] = new[2]
                elif( new[0] == "MaxOffset" ):
                    lock.acquire()
                    for filt in RunningList:
                        RunningList[filt][1].send( [ "MaxOffset",
new[2] ] )
                    lock.release()
                    FilterSettings['MaxOffset'] = new[2]
                elif( new[0] == "FKill" ):
                    lock.acquire()
                    i = 0
```

```
					fkill = str( new[1] )
					while( RunningList[i][0] != fkill ):
						i = i + 1
						if( i == len( RunningList ) ):break
					if( i != len( RunningList ) ):
						RunningList[i][1].send( [ "FKill", True ] )
					DataRouterPipe.send( [ "Ignore", fkill ] )
				elif( new[0] == "DWDelay" ):
					DataWriterPipe.send( [ "Delay", float( new[2] ) ] )
				elif( new[0] == "SendFreq" ):
					DataWriterPipe.send( [ "SendFreq", float( new[2] )
] )
				elif( new[0] == "Alpha" ):
					InverterPipe.send( [ "Alpha", float( new[2] ) ] )
				elif( new[0] == "MaxChildren" ):
					InverterPipe.send( [ "MaxChildren", float( new[2] )
] )
				elif( new[0] == "InvKillTime" ):
					InverterPipe.send( [ "InvKillTime", float( new[2] ) ] )
				elif( new[0] == "Label" ):
					tex = ""
					for x in range( len( new ) - 2 ):
						tex = tex + " " + new[x + 2]
					InverterPipe.send( [ "Label", tex ] )
				elif( new[0] == "Model" ):
					InverterPipe.send( [ "Model", new[2] ] )
					SlipWriterPipe.send( [ "Model", new[2] ] )
				elif( new[0] == "Tag" ):
					InverterPipe.send( [ "Tag", new[2] ] )
					SlipWriterPipe.send( [ "Tag", new[2] ] )
				elif( new[0] == "MinOffset" ):
					InverterPipe.send( [ "MinOffset", float( new[2] ) ] )
				elif( new[0] == "RangeThres" ):
					InverterPipe.send( [ "RangeThres", float( new[2] ) ]
)
				elif( new[0] == "Convergence" ):
					InverterPipe.send( [ "Convergence", float( new[2] )
] )
				elif( new[0] == "StrikeSlip" ):
					InverterPipe.send( [ "StrikeSlip", new[2] ] )
					print "StrikeSlip = " + new[2]
				elif( new[0] == "SWDelay" ):
					SlipWriterPipe.send( [ "Delay", float( new[2] ) ] )
				elif( new[0] == "SWMagnitude" ):
```

```
                    SlipWriterPipe.send( [ "Magn", float( new[2] ) ] )
                elif( new[0] == "SWDuration" ):
                    SlipWriterPipe.send( [ "Dur", float( new[2] ) ] )
                elif( new[0] == "Email" ):
                    global email
                    email = new[2]
                    SlipWriterPipe.send( [ "Email", email ] )
                    DataRouterPipe.send( [ "Email", email ] )

            p = sub.Popen( [ 'cp', 'Config', '.Config' ] )
            p.wait()
            odate = ndate
            print "Config File Modified"
            logging.info( "Config File Modified" )
            time.sleep( ConfigCheck )

        else:
            logging.info( "Config File Not Modified" )
            time.sleep( ConfigCheck )


# store filter variables in case filter restarts later
def UpdateFilter( l ):
    KalmanList[l[0]].K = l[1]
    KalmanList[l[0]].M = l[2]
    KalmanList[l[0]].P = l[3]
    KalmanList[l[0]].ResetP = l[4]
    KalmanList[l[0]].State = l[5]
    KalmanList[l[0]].State2 = l[6]
    KalmanList[l[0]].IState = l[7]
    KalmanList[l[0]].IState2 = l[8]
    KalmanList[l[0]].SMea = l[9]
    KalmanList[l[0]].offset = l[10]
    KalmanList[l[0]].Rcount = l[11]
    KalmanList[l[0]].InitP = l[12]
    KalmanList[l[0]].PCount = l[13]
    KalmanList[l[0]].SMCount = l[14]
    KalmanList[l[0]].EQCount = l[15]
    KalmanList[l[0]].prevTime = l[16]
    KalmanList[l[0]].Tag = l[17]
    KalmanList[l[0]].StartUp = l[18]

# turn off filters as necessary
def _FilterWatcher():
```

```
	sent = dt.now()
	defdate = sent
	while( Run == True ):
		to_delete = []
		listLock.acquire()
		for filt in RunningList:
			if( RunningList[filt][1].poll() == True ):
				print "TULiveFilter is starting to kill filter" + str( filt )
				t = RunningList[filt][1].recv()
				if( t[0] == "Kill" ):
					print "Starting to Kill filter " + str( filt )
					RunningList[filt][1].send( True )
					l = RunningList[filt][1].recv()
					if( l == False ):
						RunningList[filt][1].send( True )
						RunningList[filt][1].recv()
					else:
						UpdateFilter( l )
						RunningList[filt][1].send( True )
						DataRouterKillPipe.send( t )
						DataRouterKillPipe.recv()
						to_delete.append( filt )
						InverterPipe.send( [ "Remove", t[1] ] )
						print "Killed Filter " + str( t[1] )
				if( t[0] == "Resend" ):
					for sett in FilterSettings:
						RunningList[filt][1].send( [ sett,
FilterSettings[sett] ] )

		for x in to_delete:
			del RunningList[x]
		listLock.release()
		if( len( RunningList ) < 1 ):
			print "Running List length < 0 "
			logging.info( "THERE ARE CURRENTLY NO RUNNING FILTERS" )
		cur = dt.now()
		if( ( sent <> defdate ) and ( cur - sent > td( minutes = 15. ) ) ):
			defdate = sent
		time.sleep( 15 )


# initialize more variables
with open( ".Config", "w" ) as file:
	file.write( '\n' )
```

```
t = os.path.getmtime( "Config" )
odate = dt.fromtimestamp( t )

print odate

lock = Lock()


mp.allow_connection_pickling()

KalmanList = {}

RunningList = {}
settings = []


FrInvertPipe, ToSWriterPipe = Pipe()
ControlPipe, SlipWriterPipe = Pipe()

SlipWriter = SlipWriter( FrInvertPipe, ControlPipe )

SlipWriterProc = mp.Process( target = SlipWriter.Run )

SlipWriterProc.start()

ToInvertPipe, FrDWriterPipe = Pipe()

ControlPipe, InverterPipe = Pipe()

Inverter = TVLiveSlip( FrDWriterPipe, ToSWriterPipe, ControlPipe )

InverterProc = mp.Process( target = Inverter.Run )

InverterProc.start()


ControlPipe, DataWriterPipe = Pipe()

ToWriterPipe, FromFilterPipe = Pipe()

FromFilterPipe = que

Writer = DataWriter( FromFilterPipe, ToInvertPipe, ControlPipe )
```

```
WriterProc = mp.Process( target = Writer.Run )

WriterProc.start()


ControlPipe, DataRouterPipe = Pipe()

ControlPipe2, DataRouterKillPipe = Pipe()

Router = DataRouter( ControlPipe, ControlPipe2 )

RouterProc = mp.Process( target = Router.Run )

RouterProc.start()

ControlPipe, OrgPipe = Pipe()

num = 0
count = 0

k = thr.Thread( target = _ConfigCheck )
k.start()

time.sleep(5)

m = thr.Thread( target = _FilterWatcher )
m.start()

ToWriterPipe = que

# main code, start filters as necessary
while True:

	t = DataRouterPipe.recv()

	listLock.acquire()
	try:
		l = KalmanList[t]
		InverterPipe.send( [ "Add", t ] )
		m = InverterPipe.recv()
		print m
		if( m[0] == "Add" ):
			FromRouter, ToFilter = Pipe()
```

```
                ToFilterPipe, ControlPipe = Pipe()
                ToFilter = _pickle_connection( ToFilter )
                l.Init_Filter( FromRouter, ToWriterPipe, ControlPipe )
                z = mp.Process( target = l.FilterOn )
                z.start()
                RunningList[t] =  [ FromRouter, ToFilterPipe, ToFilter ]
                for sett in FilterSettings:
                        RunningList[t][1].send( [ sett, FilterSettings[sett] ] )
                logging.info( "Restarting filter " + str( t ) )
                DataRouterPipe.send( [ "Add", t, ToFilter ] )
        elif( m[0] == "Ignore" ):
                DataRouterPipe.send( [ "Ignore", t ] )
except:
        InverterPipe.send( [ "Add", t ] )
        m = InverterPipe.recv()
        print m
        if( m[0] == "Add" ):
                FromRouter, ToFilter = Pipe()
                ToFilterPipe, ControlPipe = Pipe()
                ToFilter = _pickle_connection( ToFilter )
                l = Kalman()
                l.setName( t )
                l.Init_Filter( FromRouter, ToWriterPipe, ControlPipe )
                KalmanList[t] = l
                z = mp.Process( target = l.FilterOn )
                z.start()
                RunningList[t] = [ FromRouter, ToFilterPipe, ToFilter ]
                for sett in FilterSettings:
                        RunningList[t][1].send( [ sett, FilterSettings[sett] ] )
                logging.info( "Beginning filter " + str( t ) )
                num = num + 1
                send = [ "Add", t, ToFilter]
                DataRouterPipe.send( send )
        elif( m[0] == "Ignore" ):
                DataRouterPipe.send( [ "Ignore", t ] )
listLock.release()
```

Appendix D2 - DataRouter.py

```
#! /usr/bin/env python
# DataRouter, connects to an outside rabbitMQ, recieves data and passes it to the correct Kalman filter

#imports
import amqp
from multiprocessing import Pipe, reduction, Lock
from multiprocessing.reduction import reduce_connection
import pickle
import json
import time
from datetime import datetime as dt
from datetime import timedelta as td
import threading as thr
import subprocess as sub
from subprocess import PIPE, Popen
import logging
import sys
import traceback


class DataRouter:

    # get pipe back from pickled pipe sent through pipe
        def _unpickle_connection( self, reduced):
                return reduced[0](*reduced[1])

    # initialize variables
        def __init__(self, CPipe, KPipe ):
                self.ConPipe = CPipe
                self.KillPipe = KPipe
                print self.ConPipe
                self.PipeList = {}
                self.exchange_name = ''
                self.host = ""
                self.userid = ""
                self.password = ""
                self.virtual_host = ""
                self.curtime = dt.now()
                self.prevtime = dt.now()
                self.WaitMsg = {}
                self.CatcherStarted = False
```

```
            self.count = 0
            self.First = True
            self.run = True
            self.lock = Lock()
            self.sendData = True
            self.ignoreList = []
            self.nextiter = 0.
            self.email = ""

    # check for changes to the configuration file
    def __CPipeWatcher( self ):
        while( self.run == True ):
            t = self.ConPipe.recv()
            if( t[0] == "Add" ):
                site = t[1]
                with self.lock:
                    self.PipeList[t[1]] = self._unpickle_connection( t[2] )
                try:
                    while( len( self.WaitMsg[site] ) > 0 ):
                        self.PipeList[t[1]].send( self.WaitMsg[site][0] )
                        del self.WaitMsg[site][0]
                    del self.WaitMsg[site]
                except:
                    pass
                logging.info( "Adding Site " + str( t[1] ) )
            elif( t[0] == "Ignore" ):
                self.ignoreList.append( str( t[1] ) )
                del self.WaitMsg[ t[1] ]
                logging.info( "Ignoring site " + str( t[1] ) )
            elif( t[0] == True ):
                pass
            elif( t[0] == "SendData" ):
                self.sendData = t[1]
            elif( t[0] == "Email" ):
                self.email = t[1]

    # set up connection variables for data input
        def _Connection( self, _host, _userid, _password, _virtual_host,
_exchange_name ):
            _connection = amqp.Connection( host = _host, userid = _userid,
password = _password, virtual_host = _virtual_host, exchange = _exchange_name )
            _channel = _connection.channel()
            _channel.exchange_declare( _exchange_name, 'fanout', passive = True )
            _queue_name = _channel.queue_declare( exclusive = True )[0]
```

```
                    _channel.queue_bind( _queue_name, exchange = _exchange_name )
                    p = sub.Popen( [ 'mail', '-s', 'DataRouter', self.email ], stdin = PIPE )
                    p.communicate( 'The DataRouter has established a connection with the
RabbitMQ server at time ' + str( dt.now() ) + '.' )
                    p.wait()
                    return _connection, _channel, _queue_name


    # main code, check data and send it through to correct filter
        def Run(self):
                    connection, channel, queue_name = self._Connection( self.host,
self.userid, self.password, self.virtual_host, self.exchange_name )

                    t = thr.Thread( target = self.__CPipeWatcher )
                    t.start()

                    k = thr.Thread( target = self.__KPipeWatcher )
                    k.start()

                    def callback( msg ):
                            jmsg = json.loads( msg.body )
                            if( str( jmsg['site'] ) not in self.ignoreList ):
                                    if( self.nextiter < float( jmsg['t'] ) ):
                                            self.nextiter = float( jmsg['t'] )
                                            logging.info( "DataRouter got data for time {} for
site {}".format( self.nextiter, jmsg['site'] ) )
                                    try:
                                            if( self.sendData == True ):
                                                    self.PipeList[str( jmsg['site'] )].send( jmsg )
                                    except:
                                            try:
                                                    self.WaitMsg[str( jmsg['site'] )].append(
jmsg )
                                            except:
                                                    if( self.First == True ):
                                                            self.First = True
                                                            l = thr.Thread( target = self.initFilter,
args = ( msg.body, str( jmsg['site'] ) ) )
                                                            l.start()

                    channel.basic_consume( queue = queue_name, callback = callback,
no_ack = True )
```

```
			while True:
				if( connection.is_alive() == False ):
					p = sub.Popen( [ 'mail', '-s', 'Inverter', self.email ], stdin =
PIPE )
					p.communicate( 'The DataRouter has lost the connection
with the RabbitMQ server at time ' + str( dt.now() ) + '.' )
					p.wait()
					connection = self._Connection( self.host, self.userid,
self.password, self.virtual_host, self.exchange_name )
				connection.drain_events()

	# set initial pipe for communication with TULiveFilter
		def setPipe( self, CPipe ):
			self.ConPipe = CPipe

	# initialize new site, store messages and wait for response
		def initFilter( self, msg, site ):
			try:
				jmsg = json.loads( msg )
				self.WaitMsg[site].append( jmsg )
			except:
				if site not in self.ignoreList:
					jmsg = json.loads( msg )
					self.WaitMsg[site] = []
					self.WaitMsg[str( site )].append( jmsg )
					self.ConPipe.send( site )
				else:
					logging.info( "Ignoring data for site " + str( site ) )

	# watch for pipes that need to turn off
		def __KPipeWatcher( self ):
			while( self.run == True ):
				t = self.KillPipe.recv()
				if( t[0] == "Kill" ):
					try:
						with self.lock:
							del self.PipeList[t[1]]
						self.KillPipe.send( [ "True" ] )
					except:
						cause = sys.exc_info()[1]
						for frame in traceback.extract_tb( sys.exc_info()[2]
):
							fname, lineno, fn, text = frame
```

```
logging.error( "Error - {} {} {} {} {}".format(
cause, fname, lineno, fn, text ) )

if t[1] in self.PipeList:
        logging.error( str( t[1] ) + " is in
Pipelist" )

else:
        logging.error( str( t[1] ) + " is not in
Pipelist" )
```

Appendix D3 - Kalman.py

```
#! /usr/bin/env python
# take data from DataRouter and process it, searching for offsets

# imports
import numpy as np
from numpy import matlib
import urllib2
import json
from datetime import datetime as dt
from datetime import timedelta as td
import logging
import time
from multiprocessing import Lock, Queue
import threading as thr
import logging
import sys
import traceback

class Kalman:

    # initialize variables
        def __init__(self):
                self.NAME = ''
                self.LAT = 0.
                self.LON = 0.
                self.HEI = 0.
                self.delta_T = 1
                self.H = np.matrix( [ [ 1., 0., 0. ], [ 0., 1., 0. ], [ 0., 0., 1. ] ] )
                self.iden = np.matrix( [ [ 1., 0., 0. ], [ 0., 1., 0. ], [ 0., 0., 1. ] ] )
                self.K = np.matrix( [ [ 0., 0., 0. ], [ 0., 0., 0. ], [ 0., 0., 0. ] ] )
                self.M = np.matrix( [ [ 0., 0., 0. ], [ 0., 0., 0. ], [ 0., 0., 0. ] ] )
                self.Mea = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
                self.P = np.matrix( [ [ 1000., 0., 0. ], [ 0., 1000., 0. ], [ 0., 0., 1000. ] ] )
                self.ResetP = self.P* 1.
                self.Phi = np.matrix( [ [ 1., 0., 0. ], [ 0., 1., 0. ], [ 0., 0., 1. ] ] )
                self.Q = np.matrix( [ [ self.delta_T,  0., 0. ], [ 0., self.delta_T, 0. ], [ 0., 0.,
self.delta_T ] ] )
                self.R = np.matrix( [ [ 0., 0., 0. ], [ 0., 0., 0. ], [ 0., 0., 0. ] ] )
                self.Res = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
                self.State = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
                self.State2 = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
                self.IState = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
```

```
self.IState2 = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
self.SMea = []
self.DispWork = False
self.DispInit = False
self.DispNum = False
self.EQPrint = False
self.offset = False
self.Rcount = 0
self.InitP = 0
self.Pcount = 0.
self.smoothing = 60.
self.SMCount = 0.
self.Wait = 2
self.EQFlag = np.matrix( [ [ False ], [ False ], [ False ] ] )
self.EQDState = 0
self.EQCount = np.matrix( [ [ 0 ], [ 0 ], [ 0 ] ] )
self.EQThres = 0.001
self.StateData = []
self.Time = 0
self.OverrideFlag = False
self.Ready = True
self.send = {}
self.prevTime = 0.
self.write = True
self.Tag = False
self.StartUp = True
self.defR = 0.0001
self.Running = False
self.streams = []
self.urlst = "http://www.panga.org/realtime/data/api/"
self.urlen = "?q=5min&l="
self.lasttime = 0
self.clusters = []
self.Live = False
self.First = False
self.ptime = 0.
self.curtime = 0.
self.streamtime = dt( year = 1970, month = 1, day = 1, hour = 0, minute =
0, second = 0 )
self.child_conn = ""
self.First_mea = True
self.lock = ''
self.run = True
self.laMea = 0
```

```
                self.DieTime = 300.
                self.PCount = 0.
                self.KillMe = False
                self.Synth = [ 0., 0., 0. ]
                self.MaxOffset = 25.0

    # set filter name
        def setName( self, n ):
                self.NAME = str( n )

    # initialize filter
        def InitFilter( self, sttime ):
                self.lasttime = sttime
                self.Live = True
                self.First = True

    # watch for changes in the config file
        def __CPipeWatcher( self ):
                while( self.run == True ):
                        if( self.KillMe == False ):
                                if( self.ConPipe.poll() == True ):
                                        t = self.ConPipe.recv()
                                        if( t == True ):
                                                self.ConPipe.send( False )
                                        elif( t != None ):
                                                if( t[0] == "EQPause" ):
                                                        self.smoothing = float( t[1] )
                                                if( t[0] == "EQThres" ):
                                                        self.EQThres = float( t[1] )
                                                if( t[0] == "MesWait" ):
                                                        self.Wait = float( t[1] ) + 1.
                                                if( t[0] == "DieTime" ):
                                                        self.DieTime = float( t[1] )
                                                if( t[0] == "MinR" ):
                                                        self.defR = float( t[1] )
                                                if( t[0] == "Offset" ):
                                                        if( t[1] == "True" ):
                                                                with open( './Offsets.d', 'r' )
as file:
                                                                        end = False
                                                                        while( end == False ):
                                                                                line =
file.readline().split()
                                                                                if not line:
```

```
logging.warning( "Could not find Synthetic offset for site " + str( self.NAME ) + "
in offset file" )
                                                                    break
                                                               if( str(
self.NAME ) == line[0] ):

    self.Synth[0] = float( line[1] )

    self.Synth[1] = float( line[2] )

    self.Synth[2] = float( line[3] )
                                                                    end =
True

    logging.info( "Found Synthetics for " + str( self.NAME ) + " of " + str( self.Synth ) )
                                        elif( t[1] == "False" ):
                                            self.Synth[0] = 0.
                                            self.Synth[1] = 0.
                                            self.Synth[2] = 0.
                                    if( t[0] == "MaxOffset" ):
                                        self.MaxOffset = float( t[1] )
                        else:
                            time.sleep( 1 )
                    else:
                        time.sleep( 1 )
                logging.warning( self.NAME + " CPipe Ending" )

  # keep code alive as long as data is coming in
    def FilterOn( self ):
        conn = self.WConn
        lo = 0.
        t = thr.Thread( target = self.__CPipeWatcher )
        t.start()
        self.lock = lo
        self.child_conn = conn
        self.Running = True
        self.ptime = dt.now()
        self.curtime = dt.now()
        while( self.Running == True ):
            self.getData()
            self.Pause()
            now = dt.now()
            if( ( now - self.laMea ) > td( seconds = self.DieTime ) ):
```

```
                self.KillFilter()
            t.join()
            logging.info( self.NAME + " Filter Ending" )
            print "Exiting Filter " + str( self.NAME )

    # get data and process it
        def getData( self ):
            update = False
            num = 0
            measurementlist = []
            try:
                while( self.RConn.poll() == True ):
                    l = self.RConn.recv()
                    measurementlist.append( l )
                    self.laMea = dt.now()
            except:
                logging.info( self.NAME + " recieved data that could not be
processed." )
                cause = sys.exc_info()[1]
                for frame in traceback.extract_tb( sys.exc_info()[2] ):
                    fname, lineno, fn, text = frame
                    logging.error( "ERROR - {} {} {} {} {}".format( cause, fname,
lineno, fn, text ) )


            measurementlist = sorted( measurementlist, key = lambda x: x['t'] )
            while( len( measurementlist ) > 0 ):
                try:
                    timest = measurementlist[0]['t']
                    if( float( timest ) > self.prevTime ):
                        if( self.testZero( measurementlist[0] ) ):
                            self.prevTime = float( timest )
                            cn = measurementlist[0]['cn']
                            cv = measurementlist[0]['cv']
                            ce = measurementlist[0]['ce']
                            n = measurementlist[0]['n'] + self.Synth[0]
                            e = measurementlist[0]['e'] + self.Synth[1]
                            v = measurementlist[0]['v'] + self.Synth[2]
                            R = np.matrix( [ [ cn, 0., 0. ], [ 0., ce, 0. ], [ 0.,
0., cv ] ] )

                            Mea = np.matrix( [ [ n ], [ e ], [ v ] ] )
                            res = Mea - self.H * self.Phi * self.State -
self.H * self.Phi * self.State2
```

```
							if( ( np.abs( res[0,0] ) < self.MaxOffset ) and
( np.abs( res[1,0] ) < self.MaxOffset ) and ( np.abs( res[2,0] ) < self.MaxOffset ) ):
								if( self.First_mea == True ):
									self.FirstMea( Mea )
									self.First_mea = False
								else:
									self.passMea( timest, Mea, R
)
							else:
								pass
						del measurementlist[0]
					except:
						logging.error( self.NAME + " could not process a
measurement." )
						cause = sys.exc_info()[1]
						for frame in traceback.extract_tb( sys.exc_info()[2] ):
							fname, lineno, fn, text = frame
							loggin.error( "ERROR - {} {} {} {} {}".format( cause,
fname, lineno, fn, text ) )
						del measurementlist[0]

	# test if data is not equal to zero
	def testZero( self, test ):
		if( test['n'] <> 0. ):
			return True
		if( test['e'] <> 0. ):
			return True
		if( test['v'] <> 0. ):
			return True
		return False

	# pause
	def Pause( self ):
		self.ptime = dt.now()
		while( self.curtime - self.ptime < td( seconds = 3 ) ):
			time.sleep( 2 )
			self.curtime = dt.now()

	# kill filter if no data recieved for long enough
	def KillFilter( self ):
		print "Starting kill process " + str( self.NAME )
		self.KillMe = True
		time.sleep( 1 )
		self.ConPipe.send( ['Kill', self.NAME ] )
```

```
			print "Request Sent for " + str( self.NAME )
			t = False
			while( t != True ):
				t = self.ConPipe.recv()
				print "Kalman t = " + str( t )
				if( t == True ):
					print str( self.NAME ) + " " + str( self.RConn.poll() )
					if( self.RConn.poll() == False ):
						logging.info( "Kill Filter " + str( self.NAME ) )
						print "Kill Filter " + str( self.NAME )
						self.FilterOff()
						self.ConPipe.recv()
						self.Running = False
						self.run = False
					else:
						logging.info( "Don't Kill Filter " + str( self.NAME ) )
						print "Don't Kill Filter " + str( self.NAME )
						self.ConPipe.send( False )
						self.ConPipe.recv()
						self.ConPipe.send( True )
						self.KillMe = False
						self.ConPipe.send( "Resend" )

# pack up data for turning off the filter
	def FilterOff( self ):
			Data = []
			Data.append( self.NAME )
			Data.append( self.K )
			Data.append( self.M )
			Data.append( self.P )
			Data.append( self.ResetP )
			Data.append( self.State )
			Data.append( self.State2 )
			Data.append( self.IState )
			Data.append( self.IState2 )
			Data.append( self.SMea )
			Data.append( self.offset )
			Data.append( self.Rcount )
			Data.append( self.InitP )
			Data.append( self.PCount )
			Data.append( self.SMCount )
			Data.append( self.EQCount )
			Data.append( self.prevTime )
			Data.append( self.Tag )
```

```
            Data.append( self.StartUp )
            self.ConPipe.send( Data )

# set variables when filter restarts
    def UpdateData( self, Data ):
            self.K = Data[1]
            self.M = Data[2]
            self.P = Data[3]
            self.ResetP = Data[4]
            self.State = Data[5]
            self.State2 = Data[6]
            self.IState = Data[7]
            self.IState2 = Data[8]
            self.SMea = Data[9]
            self.offset = Data[10]
            self.Rcount = Data[11]
            self.InitP = Data[12]
            self.PCount = Data[13]
            self.SMCount = Data[14]
            self.EQCount = Data[15]
            self.prevTime = Data[16]
            self.Tag = Data[17]
            self.StartUp = Data[18]

 # check offset flags
 def EQFlagTest( self ):
       if( self.EQFlag[0,0] == True ):
            return True
       elif( self.EQFlag[1,0] == True ):
            return True
       elif( self.EQFlag[2,0] == True ):
            return True
       else:
            return False

# check if the number of anomalous measurement is greater than MesWait
    def EQNumTest( self ):
            nu = self.EQCount[0,0]
            if( self.EQCount[1,0] > nu ):
                    nu = self.EQCount[1,0]
            if( self.EQCount[2,0] > nu ):
                    nu = self.EQCount[2,0]
            return nu
```

```
    # process first measurement differently than other measurements
        def FirstMea( self, Mea ):
                self.State2 = Mea * 1.0
                self.StartUp = True

    # set up measurement for processing
        def passMea( self, Time, Mea, R ):
                self.Ready = False
                self.Time = Time
                self.Mea = Mea
                self.R = R
                if( self.R[0,0] < self.defR ):
                        self.R[0,0] = self.defR
                if( self.R[1,1] < self.defR ):
                        self.R[1,1] = self.defR
                if( self.R[2,2] < self.defR ):
                        self.R[2,2] = self.defR
                if( self.offset == False ):
                        self.updateMat()
                else:
                        self.passupdateState()

    # update Riccati equations
        def updateMat( self ):
                if( self.prevTime <> 0 ):
                        self.delta_T = self.Time - self.prevTime
                        self.prevTime = self.Time
                        self.Q = np.matrix( [ [ self.delta_T, 0., 0. ], [ 0., self.delta_T, 0. ], [
0., 0., self.delta_T ] ] )
                self.M = self.Phi * self.P * self.Phi.T + self.Q
                interm = (self.H * self.M * self.H.T + self.R ).I
                self.K = self.M * self.H.T * interm
                self.P = ( self.iden - self.K * self.H ) * self.M
                self.calcRes()

      def calcRes( self ):
           self.Res = self.Mea - self.H * self.Phi * self.State - self.H * self.Phi * self.State2
           if( self.DispWork == True):
                print 'Mea = '
                print self.Mea
                print 'Res = '
                print self.Res
                  if( self.OverrideFlag == False ):
                    self.determineState()
```

```
	# determine state of filter
	def determineState( self ):
		if( ( self.SMCount >= self.smoothing ) and ( self.StartUp == True ) ):
			self.StartUp = False
		if( self.SMCount < self.smoothing ):
			self.EQCount = np.matrix( [ [ 0 ], [ 0 ], [ 0 ] ] )
			self.NormalMode()
			self.endProc()
		else:
			if( np.abs( self.Res[0,0] ) < np.sqrt( self.R[0,0] ) * self.EQThres ):
				self.EQFlag[0,0] = False
				self.EQCount[0,0] = 0
			else:
				self.EQFlag[0,0] = True
				self.EQCount[0,0] = self.EQCount[0,0] + 1
			if( np.abs( self.Res[1,0] ) < np.sqrt( self.R[1,1] ) * self.EQThres ):
				self.EQFlag[1,0] = False
				self.EQCount[1,0] = 0
			else:
				self.EQFlag[1,0] = True
				self.EQCount[1,0] = self.EQCount[1,0] + 1
			if( np.abs( self.Res[2,0] ) < np.sqrt( self.R[2,2] ) * self.EQThres ):
				self.EQFlag[2,0] = False
				self.EQCount[2,0] = 0
			else:
				self.EQFlag[2,0] = True
				self.EQCount[2,0] = self.EQCount[2,0] + 1
			if( ( self.EQFlagTest() == True ) and ( self.EQNumTest() > self.Wait ) 
and ( self.offset == True ) ):
				self.EQState()
			elif( ( self.EQFlagTest() == False ) and ( self.offset == True ) ):
				self.FalseEQState()
			elif( ( self.EQFlagTest() == True ) and ( self.offset == False ) ):
				self.BeginEQTestState()
			else:
				self.NormalMode()
				self.endProc()

	# process state as normal
	def NormalMode( self ):
		self.State = self.Phi * self.State + self.K * self.Res
		self.State2 = self.Phi * self.State2
		self.Tag = False
```

```
            if( self.DispWork == True):
                print 'State = '
                print self.State
            if( ( self.SMCount < self.smoothing ) and ( self.StartUp == False ) ):
                self.Tag = True
            self.send = {}
            self.send['site'] = self.NAME
            self.send['la'] = self.LAT
            self.send['lo'] = self.LON
            self.send['mn'] = self.Mea[0,0]
            self.send['me'] = self.Mea[1,0]
            self.send['mv'] = self.Mea[2,0]
            self.send['kn'] = self.State[0,0]
            self.send['ke'] = self.State[1,0]
            self.send['kv'] = self.State[2,0]
            self.send['cn'] = self.R[0,0]
            self.send['ce'] = self.R[1,1]
            self.send['cv'] = self.R[2,2]
            self.send['he'] = self.HEI
            self.send['ta'] = self.Tag
            self.send['st'] = self.StartUp
            self.send['time'] = self.Time

# process measurements when an offset has been detected
    def EQState( self ):
        if( self.EQPrint == True ):
            print 'Start EQ Process'
            print 'Time = ' + str( self.Time )
            print 'ResN = ' + str( self.Res[0,0] )
            print 'ResE = ' + str( self.Res[1,0] )
            print 'ResV = ' + str( self.Res[2,0] )
            print 'StateN = ' + str( self.State[0,0] )
            print 'StateE = ' + str( self.State[1,0] )
            print 'StateV = ' + str( self.State[2,0] )
            print 'RN = ' + str( np.sqrt( self.R[0,0] ) )
            print 'RE = ' + str( np.sqrt( self.R[1,1] ) )
            print 'RU = ' + str( np.sqrt( self.R[2,2] ) )


            self.offsetreset( )
        self.SMCount = 0
        self.SMea.append( [ self.Time, self.Mea, self.R ] )
        self.InitP = self.P[0,0]
        self.P = self.ResetP * 1.0
```

```
            self.Pcount = 0.
                    self.offset = False
                    self.OverrideFlag = True
            while( ( True ) and ( len( self.SMea ) > 1 ) ):
                  self.R = self.SMea[0][2]
                              self.Mea = self.SMea[0][1]
                              self.Time = self.SMea[0][0]

                              self.updateMat()
                  self.NormalMode()
                              del self.SMea[0]
                              if( len( self.SMea ) == 1 ):break
                    self.write = True
                    self.R = self.SMea[0][2]
            self.Mea = self.SMea[0][1]
                    self.Time = self.SMea[0][0]
                    self.OverrideFlag = False
            self.SMea = []

    # process measurements when a few anomalous measurements come in but
everything is normal
          def FalseEQState( self ):
            if( self.EQPrint == True ):
                  print 'Ending EQ test'
                  print 'ResN = ' + str( self.Res[0,0] )
                  print 'ResE = ' + str( self.Res[1,0] )
                  print 'ResV = ' + str( self.Res[2,0] )
                  print 'StateN = ' + str( self.State[0,0] )
                  print 'StateE = ' + str( self.State[1,0] )
                  print 'StateV = ' + str( self.State[2,0] )
                  print 'RN = ' + str( np.sqrt( self.R[0,0] ) )
                  print 'RE = ' + str( np.sqrt( self.R[1,1] ) )
                  print 'RU = ' + str( np.sqrt( self.R[2,2] ) )

                    self.write = True
                    self.endpassState( )
                    self.OverrideFlag = True
            self.SMea.append( [ self.Time, self.Mea, self.R ] )
            while( len( self.SMea ) > 1 ):
                  self.R = self.SMea[0][2]
                  self.Mea = self.SMea[0][1]
                              self.Time = self.SMea[0][0]
                  self.calcRes( )
                  self.NormalMode( )
```

```
                        self.updateMat( )
                        del self.SMea[0]
                        if( len( self.SMea ) == 1 ):break
                self.offset = False
                self.R = self.SMea[0][2]
            self.Mea = self.SMea[0][1]
                self.Time = self.SMea[0][0]
            self.SMea = []
                self.OverrideFlag = False

    # test if an eq has been detected
        def BeginEQTestState( self ):
            if( self.EQPrint == True ):
                print 'EQ potentially detected at time ' + str( self.Time )
                print 'ResN = ' + str( self.Res[0,0] )
                print 'ResE = ' + str( self.Res[1,0] )
                print 'ResV = ' + str( self.Res[2,0] )
                print 'StateN = ' + str( self.State[0,0] )
                print 'StateE = ' + str( self.State[1,0] )
                print 'StateV = ' + str( self.State[2,0] )
                print 'RN = ' + str( np.sqrt( self.R[0,0] ) )
                print 'RE = ' + str( np.sqrt( self.R[1,1] ) )
                print 'RU = ' + str( np.sqrt( self.R[2,2] ) )

            self.offset = True
                self.passStateStart( )
                self.write = False

    # begin killing filter
        def endProc( self ):
                if( self.offset == False ):
                        self.SMCount = self.SMCount + 1
                        self.NormalMode()
                else:
                        self.SMea.append( [ self.Time, self.Mea, self.R ] )
                try:
                        self.WConn.put( self.send, True, 15. )
                except:
                        logging.error( "Site " + str( self.NAME ) + " could not send data for
time " + str( self.send['time'] ) )
                del self.send
                self.StateData = []
                self.Ready = True
```

```
# save state after reset
    def passStateStart( self ):
        self.IState = self.State * 1.
        self.IState2 = self.State2 * 1.

# update state
    def passupdateState( self ):
        self.State = self.Phi * self.State
        self.State2 = self.Phi * self.State2
        self.calcRes()

# save state after reset
    def endpassState( self ):
        self.State = self.IState * 1.
        self.State2 = self.IState2 * 1.

# reset state after offset
    def offsetreset( self ):
        self.State2 = self.IState + self.IState2
        self.State = np.matrix( [ [ 0. ], [ 0. ], [ 0. ] ] )
```

Appendix D4 - DataWriter.py

```
#! /usr/bin/env python
# take data from the Kalman filters, organize it, and pass it through to SlipWriter

# imports
from multiprocessing import Pipe, Queue
from datetime import datetime
from datetime import timedelta
import time
import threading as thr
import logging

class DataWriter:

    # initialize variables
        def __init__( self, IPipe, OPipe, CPipe ):
                self.ConPipe = CPipe
                self.InputPipe = IPipe
                self.OutputPipe = OPipe
                self.cutoff = 0
                self.DataArray = []
                self.Sorted = []
                self.nextiter = 0
                self.curiter = 0
                self.delay = 15.
                self.run = True
                self.sendFreq = 1.
                self.nextSend = 0.

    # check for changes in the config file
        def __CPipeWatcher( self ):
                while( self.run == True ):
                        t = self.ConPipe.recv()
                        if( t != None ):
                                if( t[0] == "Delay" ):
                                        self.delay = float( t[1] )
                                if( t[0] == "SendFreq" ):
                                        self.sendFreq = float( t[1] )

    # main code
        def Run( self ):
                t = thr.Thread( target = self.__CPipeWatcher )
```

```
                    t.start()
                    First = True
                    print "NextIter = " + str( self.nextiter ) + " and CurIter = " + str( self.curiter
) + " and nextSend = " + str( self.nextSend )
                    while( True ):
            # sort code
                            self.Sorted = sorted( self.Sorted, key = lambda x: x[0] )
                            while( self.nextiter <= self.curiter - self.delay ):
                # select data to send
                                    short = []
                                    if( len( self.Sorted ) > 0 ):
                                            while( ( len( self.Sorted ) > 0 ) and (
self.Sorted[0][0] < self.nextiter ) ):
                                                    del self.Sorted[0]
                                            while( ( len( self.Sorted ) > 0 ) and (
self.Sorted[0][0] == self.nextiter ) and ( self.nextiter >= self.nextSend ) ):
                                                    short.append( self.Sorted[0] )
                                                    del self.Sorted[0]
                                                    if( len( self.Sorted ) == 0):
                                                            break
                                    if( len( short ) > 0 ):
                                            try:
                        # send data
                                                    self.OutputPipe.send( short )
                                                    logging.info( "DataWriter sent data for " +
str( self.nextiter ) )
                                                    self.nextSend = self.nextiter + self.sendFreq
                                            except:
                                                    logging.error( "DataWriter could not send
data for " + str( self.nextiter ) )
                                            del short
                                            time.sleep( 0.5 )
                                    self.nextiter = self.nextiter + 1
            # check input pipe
                            try:
                                    while( self.InputPipe.empty() == False ):
                                            t = self.InputPipe.get()
                                            if( t != None ):
                                                    epoch = t['time']
                                                    if( epoch > self.nextiter ):
                                                            self.Sorted.append( [ epoch, t ] )
                                                    if( epoch > self.curiter ):
                                                            self.curiter = epoch
```

```
                                                logging.info( "DataWriter recieved
data for " + str( epoch ) )
                                                if( self.nextiter == 0. ):
                                                        self.nextiter = self.curiter
                        except:
                                pass
```

Appendix D5 - TVLiveSlip.py

```
#! /usr/bin/env python
# take data from DataWriter, span off slip inversions as necessary

# imports
import subprocess as sub
from subprocess import PIPE, Popen
import numpy as np
import scipy as sp
from scipy import optimize
import math
from datetime import datetime as dt
import multiprocessing as mp
import time
import ok
from datetime import timedelta as td
import json
from multiprocessing import Lock
from pytz import timezone
import threading as thr
import logging
import sys
import traceback

class TVLiveSlip:

    # initialize variables
        def __init__( self, IPipe, OPipe, CPipe ):
                self.INDataPipe = IPipe
                self.OUTDataPipe = OPipe
                self.ConPipe = CPipe
                smoothing = True
                CornerFix = False
                shortSmoothing = True
                self.alpha = 1.0
                cutoff = 0.
                noise = 0.
                self.run = True
                self.maxChildren = 4.
                self.lock = Lock( )
                self.inversionList = []
                self.invLock = Lock( )
                self.inversionKillTime = 600.
```

```
		self.Convergence = -1.
		self.minOffset = -1.
		self.numFaults = -1.
		self.rangeThres = -1.
		self.SubfaultWid = 30.
		self.SubfaultLen = 60.
		self.label = ""
		self.model = ""
		self.tag = ""
		self.StrikeSlip = False

		self.Faults = []
		first = True
		with open( './subfaults.d', 'r' ) as file:
			while True:
				line = file.readline().split()
				if not line:break
				if( ( first == True ) and ( line[0] <> '#' ) ):
					self.SubfaultLen = float( line[0] )
					self.SubfaultWid = float( line[1] )
					first = False
				elif( line[0] <> '#' ):
					self.Faults.append( line )

		sites = []
		with open( './sites.d', 'r' ) as file:
			while True:
				line = file.readline().split()
				if not line:break
				if( line[0] <> '#' ):
					sites.append( line )

		sites.sort()

		self.numFaults = len( self.Faults )
		a = np.ndarray( [ 0. ] )
		Offset = a.copy()
		Offset.resize( ( 1, len( sites * 3 ) ) )
		for con in range( len( sites ) * 3 ):
			Offset[0][con] = 0.
		self.Correlate = []
		self.SubInputs = a.copy()
		self.SubInputs.resize( ( len( sites ) * 3, len( self.Faults ) ) )
# compute dummy sige variables
```

```
		for num in range( len( sites ) ):		# len sites
			self.Correlate.append( [ sites[num][0], sites[num][1],
sites[num][2] ] )
			curtime = dt.now()
			print "Running Site " + str( num ) + " " + str( curtime )
			for con in range( len( self.Faults ) ):
				com = []
				com.append( float( self.Faults[con][0] ) ) # Lat 0
				com.append( float( self.Faults[con][1] ) ) # Lon 1
				com.append( float( self.Faults[con][2] ) ) # Dep 2
				com.append( float( self.Faults[con][3] ) ) # Str 3
				com.append( float( self.Faults[con][4] ) ) # Dip 4
				com.append( 0 ) # Rake 5
				com.append( float( self.Faults[con][5] ) ) # Len 6
				com.append( float( self.Faults[con][6] ) ) # Wid 7
				com.append( 1 ) # Slip 8
				com.append( 0 ) # Ten 9
				com.append( float( sites[num][1] ) ) # station Lat 10
				com.append( float( sites[num][2] ) ) # Station Lon 11
				com.append( 0 ) # station Depth 12

				com[5] = 0.
				info = ok.dc3d( com[0], com[1], com[2], com[3], com[4],
com[5], com[6], com[7], com[8], com[9], com[10], com[11], com[12] )
				self.SubInputs[ num * 3 ][ con ] = float( info[0] )
				self.SubInputs[ num * 3 + 1 ][ con ] = float( info[1] )
				self.SubInputs[ num * 3 + 2 ][ con ] = float( info[2] )

	# create mask matrix
		Mask = a.copy()
		Mask.resize( ( len( sites ) * 3, 1 ) )
		for num in range( len( sites ) * 3 ):
				Mask[num][0] = 0.

	# create smoothing matrix
		self.smoothMat = a.copy()
		self.smoothMat.resize( ( len( self.Faults ), len( self.Faults ) ) )

		if( smoothing == True ):
			self.smoothMat = a.copy()
			self.smoothMat.resize( ( len( self.Faults  ), len( self.Faults ) ) )
			if( shortSmoothing == False ):
```

```
                                        limit = math.sqrt( math.pow( float( self.Faults[0][5] ) /
111., 2 ) + math.pow( float( self.Faults[0][6] ) / 111., 2 ) + math.pow( float(
self.Faults[0][2]) / 111.,  2 ) ) * 0.9
                                        for num in range( len( self.Faults ) ):
                                                for con in range( len( self.Faults ) ):
                                                        self.smoothMat[num,con] = 0.
                                        for num in range( len( self.Faults ) ):
                                                con = num + 1
                                        while( con < len( self.Faults ) ):
                                                if( math.sqrt( math.pow(float( self.Faults[num][0] )
- float( self.Faults[con][0] ), 2 ) + math.pow( float( self.Faults[num][1] ) - float(
self.Faults[con][1] ), 2 ) + math.pow( ( float( self.Faults[num][2] ) - float(
self.Faults[con][2] ) ) / 111., 2 ) ) < limit ):
                                                        self.smoothMat[num][con ] = 1.
                                                        self.smoothMat[con][num] = 1.
                                                        self.smoothMat[num][num] =
self.smoothMat[num,num] - 1.
                                                        self.smoothMat[con][con] =
self.smoothMat[con,con] - 1.
                                                        con = con + 1
                                else:
                                        for num in range( len( self.Faults ) ):
                                                self.smoothMat[num][ num ] = 0
                                                if( num > self.SubfaultLen ):
                                                        for con in range( 1 ):
                                                                self.smoothMat[num + con][num +
con] = -1
                                                                self.smoothMat[num -
self.SubfaultLen + con][ num + con] = 1
                                                                self.smoothMat[num + con][ num -
self.SubfaultLen + con] = 1
                                                if( num < ( self.SubfaultLen * ( self.SubfaultWid - 1 )
) ):
                                                        for con in range( 1 ):
                                                                self.smoothMat[num + con][ num +
con] = self.smoothMat[num + con][ num + con] - 1
                                                                self.smoothMat[num + con +
self.SubfaultLen][ num + con] = 1
                                                                self.smoothMat[num + con][ num +
con + self.SubfaultLen ] = 1
                                                if( num % self.SubfaultLen <> 0 ):
                                                        for con in range( 1 ):
                                                                self.smoothMat[num + con][ num +
con ] = self.smoothMat[num + con][ num + con] - 1
```

```
                                                self.smoothMat[num + con - 1][ num
+ con] = 1
                                                self.smoothMat[num + con][ num +
con - 1 ] = 1
                                        if( num % self.SubfaultLen <> self.SubfaultLen - 1 ):
                                                for con in range( 1 ):
                                                        self.smoothMat[num + con][ num +
con] = self.smoothMat[num + con][ num + con] - 1
                                                        self.smoothMat[num + con + 1][
num + con ] = 1
                                                        self.smoothMat[num + con][ num +
con + 1 ] = 1

                        if( CornerFix == True ):
                                for num in range( len( self.Faults ) ):
                                        self.smoothMat[num][num] = -4

                self.AddMatrix = a.copy()
                self.AddMatrix.resize( ( self.SubfaultLen, len( self.Faults ) ) )

                if( self.StrikeSlip == True ):
                        for num in range( len( self.SubfaultWid ) ):
                                for con in range( len( self.SubfaultLen ) ):
                                        self.AddMatrix[ num + con * self.SubfaultWid ][ con
] = 1.

                sit = 3 * len( sites )
                fau = len( self.Faults )

                tempSubMat = a.copy()
                tempSubMat.resize( ( sit, fau ) )

                tempOffMat = a.copy()
                tempOffMat.resize( ( 1, sit + fau ) )
                tempOffMat = []
                for num in range( len( sites ) * 3 ):
                        tempOffMat.append( 0. )

                tempMask = a.copy()
                tempMask.resize( ( sit + fau, 1 ) )

                for num in range( sit ):
                        for con in range( fau ):
                                tempSubMat[num][con] = self.SubInputs[num][con]
```

```
		for num in range( sit ):
			tempOffMat[num] = Offset[0][num]

		tempMask.fill(0)

		for num in range( fau ):
			tempMask[sit+num][0] = 1.

		Mask = tempMask.copy()

		self.InvSubInputs = tempSubMat.copy()


		self.Offset = tempOffMat

		self.stMask = Mask.copy()
		proclist = []
		print "Self.SubInputs = " + str( self.SubInputs.shape )

# check for changes to config file
	def __CPipeWatcher( self ):
		while( self.run == True ):
			t = self.ConPipe.recv()
			if( t != None ):
				if( t[0] == "Alpha" ):
					self.alpha = float( t[1] )
				if( t[0] == "MaxChildren" ):
					self.maxChildren = float( t[1] )
				if( t[0] == "InvKillTime" ):
					self.inversionKillTime = float( t[1] )
				if( t[0] == "Label" ):
					self.label = t[1]
				if( t[0] == "Model" ):
					self.model = t[1]
				if( t[0] == "Tag" ):
					self.tag = t[1]
				if( t[0] == "MinOffset" ):
					if( self.minOffset == -1. ):
						self.minOffset = float( t[1] )
				if( t[0] == "RangeThres" ):
					if( self.rangeThres == -1. ):
						self.rangeThres = float( t[1] )
				if( t[0] == "Convergence" ):
```

```
						if( self.Convergence == -1. ):
							self.Convergence = float( t[1] )
					if( t[0] == "StrikeSlip" ):
						if( t[1] == "True" ):
							self.StrikeSlip = True
						else:
							self.StrikeSlip = False
					if( t[0] == "Remove" ):
						logging.info( "TVSlip removing " + str( t[1] ) )
						i = 0
						while( ( i < len( self.Correlate ) ) and ( str(
self.Correlate[i][0] ) != t[1] ) ):
							i = i + 1
						if( i != len( self.Correlate ) ):
							try:
								with self.lock:
									del self.Correlate[i]
									for pil in range( 3 ):
										self.InvSubInputs =
np.delete( self.InvSubInputs, 3 * i, 0 )
										self.SubInputs =
np.delete( self.SubInputs, 3 * i, 0 )
										del self.Offset[ 3 * i ]
							except:
								cause = sys.exc_info()[1]
								for frame in traceback.extract_tb(
sys.exc_info()[2] ):
									fname, lineno,fn, text =
frame
									logging.error( "ERROR - {} {}
{} {} {}".format( cause, fname, lineno, fn, text ) )
						print "Removed " + str( t[1] )
						logging.info( "TVSlip removed " + str( t[1] ) )

					if( t[0] == "Add" ):
						logging.info( "TVSlip adding " + str( t[1] ) )
						line = ""
						a = np.ndarray( [ 0. ] )
						temp = a.copy()
						temp.resize( ( 3, len( self.Faults ) ) )
						with open( './site_lat_lon_ele.txt', 'r' ) as file:
							while True:
								line = file.readline().split()
								if not line:break
```

```
                    if( line[0] == t[1] ):break
            count = 0
            for num in range( len( self.Faults ) ):
                com = []
                com.append( float( self.Faults[num][0] ) )
                com.append( float( self.Faults[num][1] ) )
                com.append( float( self.Faults[num][2] ) )
                com.append( float( self.Faults[num][3] ) )
                com.append( float( self.Faults[num][4] ) )
                Rake = com[3] - self.Convergence
                Rake = Rake + 180.
                if( Rake < 0. ):
                    Rake = Rake + 360
                if( Rake > 360. ):
                    Rake = Rake - 360.
                com.append( Rake )
                com.append( float( self.Faults[num][5] ) )
                com.append( float( self.Faults[num][6] ) )
                com.append( 1 )
                com.append( 0 )
                try:
                    com.append( float( line[1] ) )
                    com.append( float( line[2] ) )
                    com.append( 0 )
                except:
                    com.append( 0 )
                    com.append( 0 )
                    com.append( 0 )
                info = ok.dc3d( com[0], com[1], com[2],
com[3], com[4], com[5], com[6], com[7], com[8], com[9], com[10], com[11], com[12] )
                temp[ 0][ num ] = float( info[0] )
                temp[ 1][ num ] = float( info[1] )
                temp[ 2][ num ] = float( info[2] )
                mag = np.sqrt( info[0]**2 + info[1]**2 +
info[2]**2 )
                if( ( mag < self.minOffset ) ):
                    count = count + 1.
            if( count / self.numFaults < self.rangeThres ):
                try:
                    self.ConPipe.send( [ "Add", t[1] ] )
                    with self.lock:
                        self.SubInputs = np.vstack( [
self.SubInputs, temp ] )
```

```
									self.InvSubInputs = 
np.vstack( [ self.SubInputs, self.smoothMat ] )
									self.Correlate.append( [ 
line[0], line[1], line[2] ]  )
									self.Offset.append( 0. )
									self.Offset.append( 0. )
									self.Offset.append( 0. )
							except:
								cause = sys.exc_info()[1]
								for frame in traceback.extract_tb( 
sys.exc_info()[2] ):
									fname, lineno, fn, text = 
frame
									logging.error( "ERROR - {} {} 
{} {} {}".format( cause, fname, lineno, fn, text ) )
						else:
							self.ConPipe.send( [ "Ignore", t[1] ] )

	# watch current running inversions and see if any have stalled
		def __InversionWatcher( self ):
			while( True ):
				time.sleep( 10 )
				now = dt.now()
				killlist = []
				try:
					with self.invLock:
						for inv in range( len( self.inversionList ) ):
							if( now - self.inversionList[inv][1] > td( 
seconds = self.inversionKillTime ) ):
								killlist.append( inv )
						i = len( killlist ) - 1
						while( i > -1 ):
							if( self.inversionList[ killlist[i] ][0].is_alive() 
== True ):
								self.inversionList[ killlist[i] 
][0].terminate()
								logging.warning( "Inversion for " + 
str( self.inversionList[ killlist[0] ][2] ) + " terminated" )
							del self.inversionList[ killlist[i] ]
							del killlist[i]
							i = i - 1
				except:
					cause = sys.exc_info()[1]
					for frame in traceback.extract_tb( sys.exc_info()[2] ):
```

```
                                        fname, lineno, fn, text = frame
                                        logging.error( "ERROR - {} {} {} {} {}".format( cause,
fname, lineno, fn, text ) )

  # main code
     def Run( self ):
          num = 0
          lock = Lock()

          t = thr.Thread( target = self.__CPipeWatcher )
          t.start()

          u = thr.Thread( target = self.__InversionWatcher )
          u.start()


          while( True ):
       # try to start another inversion
               try:
                    station = self.INDataPipe.recv()
                    num = num + 1
                    while( len( mp.active_children() ) >= self.maxChildren ):
                         time.sleep( 0.1 )
                    with self.lock:
                         p = mp.Process( target=self.SingleInverter, args=(
station, self.alpha, self.stMask, self.SubInputs, self.smoothMat, self.Offset,
self.OUTDataPipe, self.Faults, self.Correlate, lock, self.AddMatrix ) )
                         p.start()
                    with self.invLock:
                         now = dt.now()
                         self.inversionList.append( [ p, now, station[0][0] ] )
               except:
                    cause = sys.exc_info()[1]
                    for frame in traceback.extract_tb( sys.exc_info()[2] ):
                         fname, lineno, fn, text = frame
                         logging.error( "ERROR - {} {} {} {} {}".format( cause,
fname, lineno, fn, text ) )




     def SingleInverter( self, station, alpha, stMask, SubInputs, smoothMat, Offset,
Pipe, Faults, Correlate, lock, AddMatrix):
```

```
		date = dt.now()
		logging.info( "TVLiveSlip beginning inversion for " + str( station[0][0] ) )
		time = station[0][0]
		Mask = stMask.copy()
		npalpha = alpha
		npcutoff = 0.
		npnoise = 0.
		inv = 0
		a = np.ndarray( [ 0.] )
		Mask = a.copy()
		k = np.shape( Offset )[0] + len( Faults )
		Mask.resize( k, 1 )

		for i in range( len( Faults ) ):
			k = np.shape( Mask )[0] - i - 1
			Mask[k][ 0 ] = 1.

		lit = 0
		con = 0
# organize data
		while( con <  len( Correlate ) ):
			start = lit
			while True:
				if( len( Correlate[con] ) < 1 ):
					print "There is an error somewhere in Correlate"
				if( station[lit][1]['site'] == Correlate[con][0] ):
					Offset[con * 3] = float( station[lit][1]['kn'] )
					Offset[con * 3 + 1] = float( station[lit][1]['ke'] )
					Offset[con * 3 + 2] = float( station[lit][1]['kv'] )
					if( station[lit][1]['ta'] == False ):
						Offset[con * 3] = 0.
						Offset[con * 3 + 1] = 0.
						Offset[con * 3 + 2] = 0.
					con = con + 1
					inv = inv + 1
					break
				else:
					lit = lit + 1
					if( lit >= len( station ) ):
						lit = 0
					if( lit == start ):
						try:
							start = lit
							del Correlate[con]
```

```
                                        SubInputs = np.delete( SubInputs, [
con * 3, con * 3 + 1, con * 3 + 2 ], 0 )
                                        Offset = np.delete( Offset, [ con * 3,
con * 3 + 1, con * 3 + 2 ], 0 )
                                        if( con >= len( Correlate ) ):
                                                break
                                except:
                                        cause = sys.exc_info()[1]
                                        for frame in traceback.extract_tb(
sys.exc_info()[2] ):
                                                fname, lineno, fn, text =
frame
                                                logging.error( "ERROR - {} {}
{} {} {}".format( cause, fname, lineno, fn, text ) )
                                        logging.error( "ERROR - Len
Correlate = {} and Len SubInputs = {} and Len Offset = {} and lit = {} and con = {}".format(
len( Correlate ), len( SubInputs ), len( Offset ), lit, con ) )
                SubInputs = np.vstack( [ SubInputs, smoothMat ] )
                sttime = dt.now()
                SI = SubInputs
                OF = Offset
                for num in Faults:
                        OF = np.append( OF, 0. )
                invbegin = dt.now()
                Solution = sp.optimize.nnls( SI, OF )
                invend = dt.now()
        # run inversion
                Solution = Solution[0]
                print "Inversion finished in " + str( invend - invbegin )
                FaultSol = []
                curtime = dt.now()
                ttime = curtime - sttime
        # compute calculated offsets
                CalcOffset =  SubInputs.dot( Solution )

        # organize solutions
                for con in range( len( Solution ) ):
                        FaultSol.append( [] )
                        FaultSol[con].append( Faults[con][0] )
                        FaultSol[con].append( Faults[con][1] )
                        FaultSol[con].append( Faults[con][2] )
                        FaultSol[con].append( Faults[con][3] )
                        FaultSol[con].append( Faults[con][4] )
```

```
				rake = self.Faults[con][7]

				FaultSol[con].append( str( rake ) )
				FaultSol[con].append( Faults[con][5] )
				FaultSol[con].append( Faults[con][6] )
				Zero = False
				slip = Solution[con]
				FaultSol[con].append( str( slip ) )
				FaultSol[con].append( "0" )
				if( Zero == False ):
					FaultSol[con].append( Solution[con] )
				else:
					FaultSol[con].append( 0. )

			FinalCalc = []
			num = 0
			for con in range( len( station ) ):
				lit = 0
				while True:
					if( station[con][1]['site'] == Correlate[lit][0] ):
						FinalCalc.append( [ station[con][1]['site'],
station[con][1]['la'], station[con][1]['lo'], station[con][1]['he'], station[con][1]['kn'],
station[con][1]['ke'], station[con][1]['kv'], CalcOffset[lit * 3], CalcOffset[lit * 3 + 1],
CalcOffset[lit * 3 + 2] ] )
						if( station[con][1]['ta'] == False ):
							FinalCalc[num][4] = 0.
							FinalCalc[num][5] = 0.
							FinalCalc[num][6] = 0.
						num = num + 1
						break
					else:
						lit = lit + 1
					if( lit == len( Correlate ) ):
						break

			send = {}

			short = []

			for lit in station:
				short.append( [ lit[1]['site'], lit[1]['kn'], lit[1]['ke'], lit[1]['kv'],
lit[1]['ta'], lit[1]['mn'], lit[1]['me'], lit[1]['mv'], lit[1]['cn'], lit[1]['ce'], lit[1]['cv'] ] )
			send['data'] = short
			if( station[0][1]['time'] ):
```

```
                send['time'] = station[0][1]['time']

            short = []

            fin = dt.utcfromtimestamp( float( send['time'] ) )

            don = fin.strftime( "%Y-%m-%d %H:%M:%S %Z" )

            send['label'] = self.label + " " +  self.model + ' - ' +don + "UTC"

        # calculate moment and moment magnitude
            Magnitude = 0.0

            for con in FaultSol:
                Magnitude = Magnitude + float( con[6] ) * float( con[7] ) * np.abs(
float( con[8] ) ) * float( 1e12 )

            Magnitude = Magnitude * float( 3e11 )

            if( self.StrikeSlip == False ):
                for lit in FaultSol:
                    short.append( lit )
            else:
                for lit in range( int( self.SubfaultLen ) ):
                    temp = FaultSol[lit]
                    temp[8] = float( temp[8] )
                    for num in range( int( self.SubfaultWid ) ):
                        temp[8] = temp[8] + float( FaultSol[ lit + num * int(
self.SubfaultLen ) ][8] )
                    short.append( temp )
            send['slip'] = short
            short = []
            for lit in FinalCalc:
                short.append( [ lit[0], lit[1], lit[2], lit[3], lit[4], lit[5], lit[6], lit[7],
lit[8], lit[9] ] )
            send['estimates'] = short
            Mw = 0.
            if( Magnitude <> 0. ):
                Mw = 2./3. * np.log10( Magnitude ) - 10.7
                Magnitude = "{:.2E}".format( Magnitude )
                Mw = "{:.1f}".format( Mw )
            else:
                Mw = "NA"
                Magnitude = "{:.2E}".format( Magnitude )
```

```
		send['Moment'] = Magnitude

		send['Magnitude'] = Mw

	# send data
		try:
			lock.acquire( timeout=1 )
			self.OUTDataPipe.send( send )
			lock.release()
			logging.info( "TVLiveSlip finished inversion for " + str( station[0][0]
) + " taking " + str( dt.now() - date ) )
			print "Inverter finished in " + str( dt.now() - date )
		except:
			cause = sys.exc_info()[1]
			for frame in traceback.extract_tb( sys.exc_info()[2] ):
				fname, lineno, fn, text = frame
				logging.error( "ERROR - {} {} {} {} {}".format( cause, fname,
lineno, fn, text ) )
			logging.error( "TVLiveSlip could not send inversion data for " + str(
station[0][0] ) )
```

Appendix D6 - SlipWriter.py

```
#! /usr/bin/env python
# take data from slip inversion, organize it and pass it outside

# imports
from multiprocessing import Pipe
import json
import pymongo
from datetime import datetime as dt
from datetime import timedelta as td
from datetime import date as da
import subprocess as sub
from subprocess import PIPE, Popen
import threading as thr
import logging
import amqp
import pika
import asyncore
import socket
from amqp import Connection
import time
import sys
import traceback



class SlipWriter:

    # initialize variables
        def __init__( self, DPipe, CPipe ):
                self.InPipe = DPipe
                self.ConPipe = CPipe
                self.nextiter = 0.
                self.curiter = 0.
                self.run = True
                self.delay = 2.
                self.mag = 7.
                self.dur = 15.
                self.outputdata = False
                self.count = 140.
                self.email = ""
                self.model = ""
                self.tag = ""
```

```
			self.exchange_name = ''
			self.host = ""
			self.port = 0
			self.userid = ""
			self.virtual_host = ""
			self.password = ""

	# check for changes to config file
		def __CPipeWatcher( self ):
			while( self.run == True ):
				t = self.ConPipe.recv()
				if( t != None ):
					if( t[0] == "Email" ):
						self.email = t[1]
					if( t[0] == "Delay" ):
						self.delay = float( t[1] )
						print "SWDelay = " + str( self.delay )
					if( t[0] == "Magn" ):
						self.mag = float( t[1] )
					if( t[0] == "Dur" ):
						self.dur = float( t[1] )
					if( t[0] == "Output" ):
						self.outputdata = t[1]
					if( t[0] == "Model" ):
						self.model = t[1]
					if( t[0] == "Tag" ):
						self.tag = t[1]

	# main code
		def Run( self ):
			t = thr.Thread( target = self.__CPipeWatcher )
			t.start()
			defdate = dt( year = 1970, month = 1, day = 1 )
			sent = defdate
			now = dt.now()
			iterData = []
			self.connection = None
			self.channel = None
			credentials = pika.PlainCredentials( self.userid, self.password )
			parameters = pika.ConnectionParameters( self.host, self.port,
self.virtual_host, credentials )
			self.connection = pika.BlockingConnection( parameters )
			self.channel = self.connection.channel()
```

```
                    self.channel.exchange_declare( exchange = self.exchange_name,
type='topic', durable = True, auto_delete = False )
                    storeData = []
                    while True:
                            now = dt.now()
            # check whether to send data
                            if( self.nextiter < self.curiter - self.delay ):
                                    self.nextiter = self.curiter - self.delay
                                    ind = 0
                # send all data necessary
                                    while( ind < len( iterData ) ):
                                            if( iterData[ind]['t'] <= self.nextiter ):
                                                    Done = False
                                                    while( Done == False ):
                                                            try:
                                                                    time2 = iterData[ind]['t']
                                                                    print "Sent data for time = "
+ str( time2 )
                                                                    isend = json.dumps(
iterData[ind] )
                                                                    self.channel.basic_publish(
exchange = self.exchange_name, routing_key = self.model, body = isend )
                                                                    print isend
                                                                    Done = True
                                                            except:
                                                                    time.sleep( 1 )
                                                                    cause = sys.exc_info()[1]
                                                                    for frame in
traceback.extract_tb( sys.exc_info()[2] ):
                                                                            fname, lineno, fn, text
= frame
                                                                            print "{} {} {} {}
{}".format( cause, fname, lineno, fn, text )
                                                    logging.info( "SlipWriter sent data for " +
str( iterData[ind]['t'] ) )
                                                    del iterData[ind]
                                                    self.count = self.count + 1
                                            ind = ind + 1
                                    while( len( storeData ) > 0 ):
                                            if( storeData[0]['time'] <= self.nextiter ):
                                                    del storeData[0]


                            if( now - sent > td( minutes = self.dur ) ):
```

```
						sent = defdate

			# recieve data and organize for final output
					t = self.InPipe.recv()
					if( t != None ):
						print "Got Data"
						if( float( t['time'] ) > self.curiter ):
							self.curiter = float( t['time'] )

						estimates = []
						for num in t['estimates']:
							estimates.append( [ num[0], num[7], num[8] ] )

						slip = []
						for con in t['slip']:
							slip.append( con[8] )

						data = []
						for lit in t['data']:
							if( lit[4] == False ):
								data.append( [ lit[0], 0., 0. ] )
							else:
								data.append( [ lit[0], lit[1], lit[2] ] )

						short = { 'estimates':estimates, 'slip':slip, 'data':data,
'time':float( t['time'] ), 'label':t['label'], 'Mw':t['Magnitude'], 'M':t['Moment'] }

						fin = json.dumps( short )

						send = {}
						send['t'] = float( t['time'] )
						send['tag'] = self.tag
						send['model'] = self.model
						send['result'] = fin

						iterData.append( send )

						if( t['Magnitude'] <> "NA" ):
							if( ( float( t['Magnitude'] ) > self.mag ) ):
								self.count = 0


						cur = dt.now()
```

```
                    if( ( sent <> defdate ) and ( cur - sent > td( minutes =
self.dur ) ) ):
                         sent = defdate
```

Appendix D7 - RMQtoMDB.py

```
#! /opt/python3.5/bin/python3
# take the output from rabbitMQ, convert it, and send it out to another mongoDB

# imports
import pika
import json
import pymongo

# set initial variables
Ihost = ""
Iexchange = ''
Iuserid = ""
Ipassword = ""
Ivirtual_host = ""
Iport = 0
Ikey = "#"
Ohost = ""
Oport = 0
Ouserid = ""
Opassword = ""

# set up connection information
Icredentials = pika.PlainCredentials( Iuserid, Ipassword )
Iparameters = pika.ConnectionParameters( Ihost, Iport, Ivirtual_host, Icredentials )
Iconnection = pika.BlockingConnection( Iparameters )
Ichannel = Iconnection.channel()
Ichannel.exchange_declare( exchange = Iexchange, type = 'topic', passive = True )

Iresult = Ichannel.queue_declare()
Iqueue_name = Iresult.method.queue

Ichannel.queue_bind( exchange = Iexchange, queue = Iqueue_name, routing_key = Ikey
)
Oclient = pymongo.MongoClient( Ohost, Oport )
Odb = Oclient.products
Odb.authenticate( Ouserid, Opassword )
Ocollection = Odb.slip_inversions

# main code retrieve data from RMQ and pass it to MDB
def callback( ch, method, properties, body ):
	print( method.routing_key )
```

```
        simp = json.loads( body.decode( "utf-8" ) )
        print( simp )
        Ocollection.insert( simp )

Ichannel.basic_consume( callback, queue = Iqueue_name, no_ack = True )

Ichannel.start_consuming()
```

Appendix D8 - Config

Run = False

# Systemwide
email = email # system wide email for monitoring the system. Sends an email sometimes when something goes wrong

# TULiveFilter
ConfigCheck = 10. # How often to check for changes to the Config file (seconds)

# DataRouter
SendData = True # Used for debugging, stops the data router from passing data to the filters

# Kalman
EQPause = 120. # Offset Detection freeze after EQ (measurements)
EQThres = 5. # Detection limit, how many standard deviations to consider for anomylous measurements
MesWait = 6. # Number of measurements to wait for
DieTime = 30. # How long to wait with no measurements before turning filter off (seconds)
MinR = 0.0001 # Minimum measurement covariance value put into the Kalman Filter (prevents 0 from screwing up the system)
Offset = False # Whether to add a Synthetic to the data
MaxOffset = 40.0 # Maximum Offset allowed in the system, any residual above this is ignored (meters)

# DataWriter
DWDelay = 15. # Delay that the DataWriter waits before sending data to allow data to arrive (seconds)
SendFreq = 1. # How many measurements to wait before sending data to start the next inversion

# TVLiveSlip
Alpha = 1 # Smoothing Parameter
MaxChildren = 4. # Maximum amount of Parallelization
InvKillTime = 30. # Wait time before killing an inversion (SendFreq * [MaxChildren - 1])
Label = Alpha Version - San Andreas -
Tag = current

Only Read Once

MinOffset = 0.001 # Anything below this value will be treated as 0 when determining if a site should be run (Changes to this will not be immediately seen)
RangeThres = 1. # Maximum percentage of 0's allowed when determining if a site should be run (Changes to this will not be immediately seen)
Convergence = 0. # Movement direction of the footwall fault
StrikeSlip = True # Determine the output for map view

# SlipWriter:
SWDelay = 0. # Delay before SlipWriter sends data to the database (seconds)
SWMagnitude = 9.0 # Magnitude that an event must be before it emails about it (Mw)
SWDuration = 150. # How long between emails to wait at a minimum (so it doesn't email every inversion) (minutes)
Email = email # Who to email

# Cycler:
Cycle = False

Appendix D9 - Cycle.py

```
#! /usr/bin/env python
'''Cycle.py
Opens the Config file and switches SendData between True
and False every 2 minutes.  Used during testing to simulate
an interruption to the data streams.'''


import time

Cycle = True
x = True
Text = []

while( Cycle == True ):
	with open( 'Config', 'r' ) as File:
		Text = File.readlines()

	with open( 'Config', 'w' ) as File:
		for line in Text:
			line = line.split()
			if( len( line ) > 3 ):
				if( line[0] == 'SendData' ):
					line[2] = str( x )
					if( x == True ):
						x = False
					else:
						x = True
				if( line[0] == 'Cycle' ):
					if( line[2] == 'False' ):
						Cycle = False
					else:
						Cycle = True
			l = ''
			for word in line:
				l = l + str( word ) + ' '
			if( len( l ) > 0 ):
				File.write( l[:-1] + '\n' )
			else:
				File.write( '\n' )
	time.sleep( 120 )
```