7 July 2009

UCONN Very Short Term Load Forecasting (VSTLF) Design Note

1 Change Log

This is the preliminary version of this design note. Future revisions will enumerate the changes that have been made on this page.

Table of Content

2 About this Document

This document serves as a design note for the Very Short Term Load Forecasting system currently under development at the University of Connecticut.

3 Document Scope

The purpose of this document is to provide details on the implementation of the very short-term load forecasting module created at UCONN. The design of this system is heavily influenced by the STLF design implemented by our team. Some modules are changed only slightly, or not at all, from the previous product. Despite this, the system architecture is described fully in this design note, and the reader should be able to become familiar with the system without any knowledge of the previous STLF project.

In particular, this document describes
The high-level design, i.e., the overall architecture of the forecaster.
The specification of each input-output data transformation that takes place when information flows from the database to the neural network
Detailed implementation level specification
Installation instructions and a users' guide.

4 Introduction

4.1 Purpose of (V)STLF

Very short-term load forecasting (VSTLF) predicts the load over one hour into the future in five minute steps based on real time collected data from gathering devices every four seconds averaged into five minutes, and performs the moving forecast every five minutes. It also can be applied to the predictions with different data collecting frequencies (e.g., three seconds) and looking into the future with different resolutions (e.g., half an hour in one minute step.) An accurate forecasting has been essential for area generation control and resource dispatch, and ensures revenue adequacy within the Independent System Operator (ISO) multi-settlement market by reducing the ex-ante dispatch. Effective VSTLF, however, is difficult with respect to possible malfunctioning of data gathering devices and complicated load features in real time.

Methods for very short-term load forecasting are limited. Existing methods include extrapolation, auto regression, fuzzy logic, Kalman filtering, and neural networks. The first two is often easier than the latter three with respect to the calculation of the model parameters. Usually, weather condition is ignored because of the large time constant of load as a function of weather. These practical models will be briefly reviewed in Section II. Among these forecasting models, neural networks have been widely used. On a different front, progress has been made on short-term load forecasting. The similar day method, single-level wavelet decomposition, and neural networks were combined (Chen and at al., 2010) to effectively predict the 24-hour load of tomorrow. A correction coefficient scheme (Zhao and at al., 2009) was further applied to enhance the predictions on holidays and days before and after. These methods open the door for analyzing complex load features at different frequencies. However, the short-term load forecasting problem does not have to consider the spikes since the operator has enough time to fix them, but it is very necessary for very short-term load to effectively handle noisy load in real time because the spikes highly affect the prediction. Another important aspect is the complicated features of very short-term load, which are different from those of short-term load. Thereafter, new methods have to be developed for VSTLF.

The purpose of this work is to design a multilevel wavelet neural network with novel data pre-filtering to forecast next hour's load in five-minute steps and perform the moving forecast every five minutes. The spikes do not reflect the true load, affect the network training, and degrade the prediction quality. To reduce effects of spikes on the prediction, a novel data filtering technique is developed to detect and correct spikes from the normal load. This method produces a new smooth signal in real time. Specifically, it corrects spikes without altering the normal load.

The very short-term load has different frequency components. The structure of the multilevel wavelet neural network (MWNN) is developed to capture load components at different frequencies. A wavelet technique is chosen to decompose the load in to several components, and each component is then appropriately transformed and fed with other proper inputs into a separate neural network so that each component can be accurately captured. Forecasts from individual neural networks are then transformed back and combined to form the over one hour forecasting into the future in five minute step. To take full use of up-to-date data and produce an accurate forecasting, several MWNN structures were further applied to form a moving forecast, and each handles the one-hour load input.

5 UConn Forecasting Method

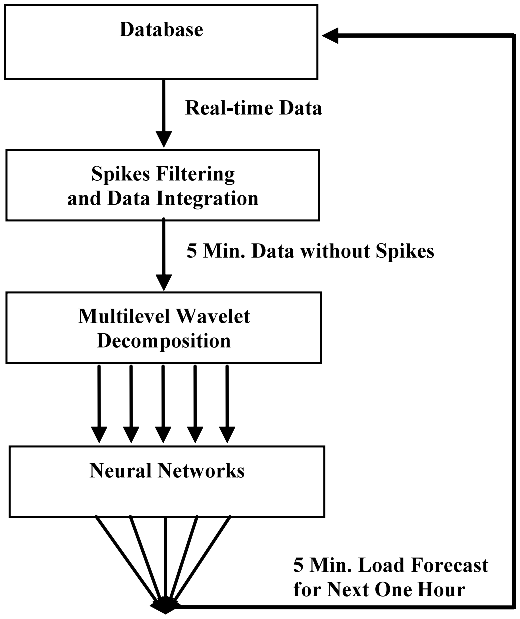

Database
Real-time Data
Spikes Filtering and Data Integration
5 Min. Data without Spikes
Multilevel Wavelet Decomposition
Neural Networks
5 Min. Load Forecast for Next One Hour

Figure 1: Overall data flow for VSTLF Process

The architecture of design can be simply depicted as in figure 1. Four second data points are received in real time. Preprocessing routines are applied to remove spikes and decrease the time scale of the data series before the information is fed through a multilevel wavelet decomposition to a set of artificial neural networks.

5.1 Preprocessing of Data Series

The processing method for load data includes micro and macro spike filtering as well as data integration. As shown in figure 2, types of spikes are analyzed first (refer to 5.1.1.) Real-time raw data is filtered to remove micro spikes (refer to 5.1.2) first and then integrated into 5 min data (refer to 5.1.3). The 5 minutes data may still exhibit macro spikes. Hence, this new signal must undergo a macro-level filtering process. Once the five minute load data points are available, the load signal is separated into five components corresponding to five frequency bands. Finally, each of the five resulting preprocessed data series is fed to a separate neural network to predict that specific frequency component of the load at 5-minute intervals. The output from the five networks is aggregated to give the final load prediction at 5 minutes intervals for an entire hour.

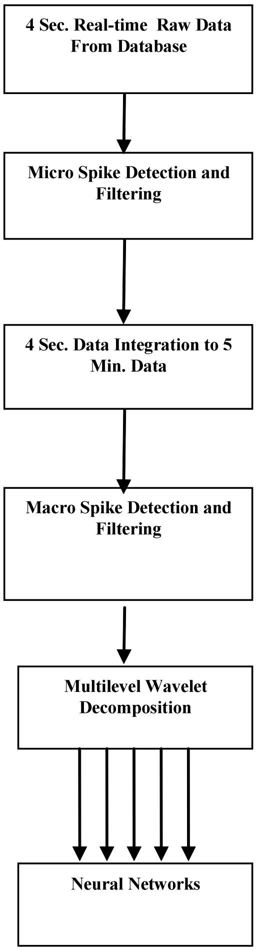

4 Sec. Real-time Raw Data From Database
Micro Spike Detection and Filtering
4 Sec. Data Integration to 5 Min. Data
Macro Spike Detection and Filtering
Multilevel Wavelet Decomposition
Neural Networks

Figure 2: Overall data flow of data series preprocessing

5.1.1 Types of Spikes and Spike Filtering Methods

Spikes may have short, long or very long durations and small, large or very large magnitudes at the levels of four seconds or five minutes. They are randomly distributed over time. Empirically, short or very short durations usually have small or very small abnormal magnitudes, whereas long or very long durations have different abnormal magnitudes. Here, they are respectively denoted as “micro spikes” and “macro spikes” compared to the “normal” data, and will be treated differently.

In our work, the micro and macro spike filtering are presented for two types of the spikes. The micro spike filtering method detects and repairs the micro spike with the smoothed load at different time resolutions. It first applies to four second load and removes spikes with short durations and small magnitudes but may not remove ones with long durations, or short durations with large magnitudes. After integrating the load of four seconds into five minutes, the spike size is further reduced. Thus this micro spike filtering can still handle them. However, observation shows that a spike can last up to two hours. The micro spike filtering can handle this case at neither four second levels nor five minute levels. Using the same detection method but correcting the macro spike via the linear interpolation, the macro spike filtering is presented for spikes at five minutes level. Since spikes are complicated with respect to durations, magnitudes and time resolutions, Table I summarizes different types of spikes at levels of four seconds and five minutes and with a spike filtering method for each.

TABLE I
SPIKE FILTERING METHODS FOR DIFFERENT TYPES OF SPIKES

Types of Spikes Duration (Points): Short / Long / Very Long Magnitude (MW): Small / Large / Very Large		Applied Spike Filtering Methods
Duration& Magnitude at Four Second Level	Duration & Magnit. After Micro Spike Filtering & Integ. at Five Min. Level	
Short & Small	No Need	_1
Short & Large	Very Short & Small	_1, 1_
Short &Very Large	Very Short & Large	_1, 1_, 2
Long & Small	Short & Very Small	_1, 1_
Long & Large	Short & Small	_1, 1_
Long &Very Large	Short & Large	_1, 1_, 2
Very Long & Small	Long & Small	_1, 3
Very Long & Large	Long & Large	_1, 1_, 2
Very Long & Very Large	Long & Very Large	_1, 1_, 2
Note: _1 : Micro Spike Filtering before Integration 1_ : Micro Spike Filtering before Integration 2 : Macro Spike Filtering after 1_ 3 : No Filtering Applied		

5.1.2 Micro (four second) spikes filtering

Filtering out micro-spikes is broken down into two tasks: (1) detecting the spikes and (2) eliminating them.

Spike Detection. The key idea for spike detection is to use a "zero phase filter" over the raw 4 seconds signal to smooth it out and obtain, via differencing with the original signal, the location of the spikes. I.e., this phase identifies the points in time in the original signal where the load significantly deviates from the sliding average. If the difference exceeds a configurable threshold, a spike is "flagged" (micro spikes detection).

Smoothing the load is an important step in spike filtering. The approach is to perform the zero phase filtering by taking the average of the input load X = {x(n-m+1)…x(N).

Here, n is the time index, N is the length of the processed load, and m is a filter order which informs the length of a filter window. y (n) is produced through (1)

$$y(n) = \sum_{i=n-m+1}^{N} x(i) \Big/ m \,. \qquad (1)$$

The smoothed sequence Y = {y(n) … y(N)} is sequentially produced through this equation. After filtering in the forward direction, the filtered sequence is appended by{x(N+1) … x(N+m-1)}, and then reversed and run through the filter. The final filtered load is the time reverse of the output of this second filtering operation (time-reversed processing operation). Note the length of the input X must be more than three times the order m so that the filter can work normally (Signal Processing Toolbox of MATLAB).

The difference between the smoothed and the actual Z = {z (n)… z (n+N)} is produced by (2)

$$z(n) = y(n) - x(n)\,. \qquad (2)$$

where large values (either positive or negative) for $z(n)$ are indicative of a spike. Figure 3 shows details of the process. A micro-spike is flagged at time t when the absolute value $|z(t)|$ exceeds a configurable threshold THD1. In the current implementation THD1 is set to 50MW by default.

Spike Correction. The second task replaces the spikes with a corrected signal. Several methods to generate a correction were considered. The simplest approach (depicted in Figure 3) is to replace the content of a window by the subsequence y for any flagged window. This is a localized correction that does not alter the normal (spike-free) segments of the original signal. The repair method used in the implementation is slightly more sophisticated. It first detects the edges of the spike within a flagged window and replaces the data points between the two edges by a linear interpolation.

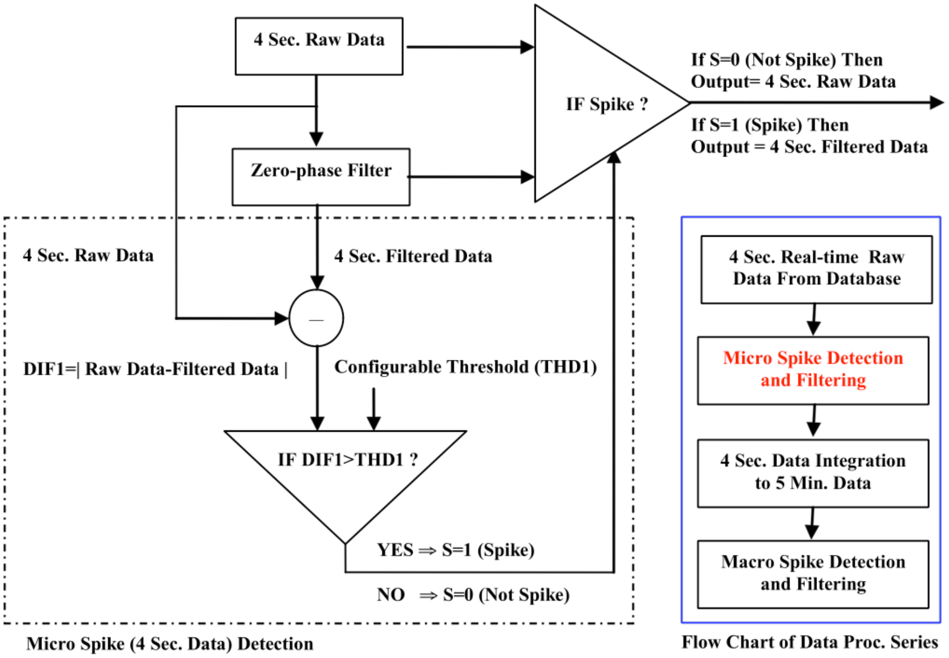

4 Sec. Raw Data
IF Spike ?
If S=0 (Not Spike) Then
Output= 4 Sec. Raw Data
If S=1 (Spike) Then
Output = 4 Sec. Filtered Data
Zero-phase Filter
4 Sec. Raw Data
4 Sec. Filtered Data
–
DIF1=| Raw Data-Filtered Data |
Configurable Threshold (THD1)
IF DIF1>THD1 ?
YES ⇒ S=1 (Spike)
NO ⇒ S=0 (Not Spike)
Micro Spike (4 Sec. Data) Detection
4 Sec. Real-time Raw Data From Database
Micro Spike Detection and Filtering
4 Sec. Data Integration to 5 Min. Data
Macro Spike Detection and Filtering
Flow Chart of Data Proc. Series

Figure 3: Micro spikes filtering

Figure 4 depicts a four-second load series before and after the presented micro filtering method has been applied. Red circles mark their differences: It can be observed that there are no spikes in these locations after the filter has been applied. At the other points it can be seen that the original load series remains unaltered. Note that macro spikes (characterized by a duration that exceeds 40 seconds – the longest window) are undetected and therefore unaffected by these repairs.

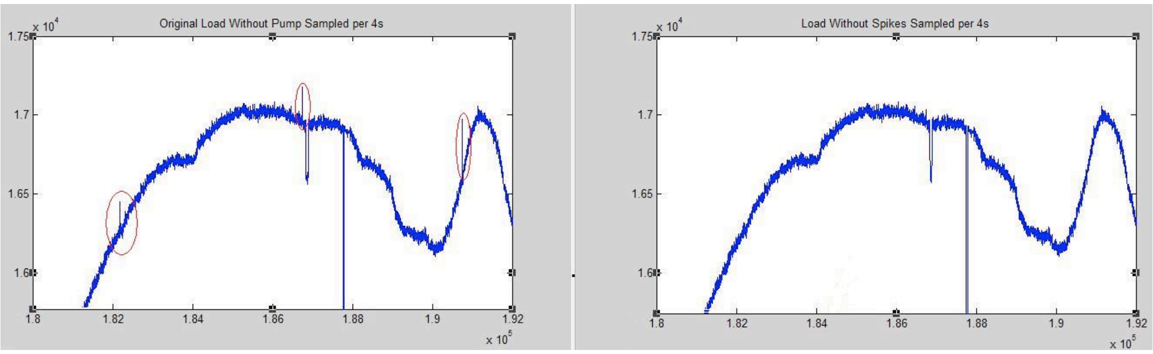


Figure 4: Micro spikes of 4 second load filtering before (left) and after (right)

5.1.3 Data Integration

Each value of five minute data point equals the average of the 75 load data points from the past five minute interval (15 data points per minute since the load is sampled every four seconds), which is described as follows

$$L_{5\min.}(n) = \frac{\sum_{i=75(n-1)+1}^{75n} L_{4\sec.}(i)}{75} \qquad n = 1,2,3...$$

5.1.4 Macro (Five Minute) spikes filtering

The idea for macro spike filtering is very similar to the filtering done at the micro level. Four second data is first integrated into a 5 minute signal. This coarser signal is then filtered with the same technique, i.e., a "zero phase filter" to detect the spikes. The repair method must, however, be modified. Indeed, simply performing a local replacement of the original signal by its filtered counter-part won't be effective given that the average will be very much affected by the magnitude and length of the spike. When applying the repair once, the spike is, at best, attenuated. But it is still present. See for instance, Figure 5. However, if the process is iterated, subsequent repair will progressively dampen the spike. A simple stopping criterion can be used (e.g., a maximum number of iterative repairs or a fixpoint condition such as "the signal is no longer changed by repairs" or "the 0-phase filter no longer sees a spike in that window").

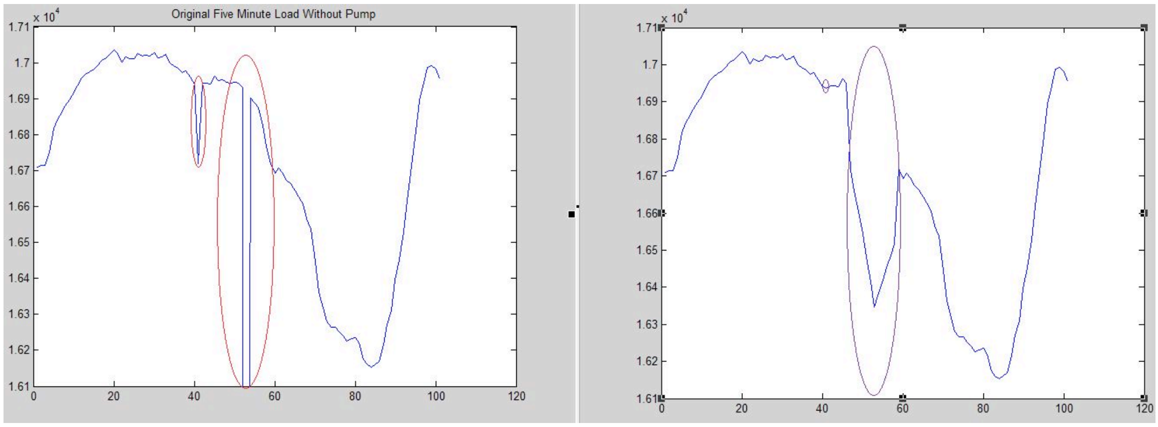


Figure 5. Two macro-level spikes are attenuated using a single pass of the 0-phase filter.

The approach implemented in VSTLF relies on filtering with short window length (2) to identify the two edges of a spike and applies a repair based on linear interpolation between the identified edges. The intent is to side-step the excessive averaging that happens right at the edge of the spike. See Figure 6 for a comparison of the two repair methods. The actual implementation corresponds to the yellow line (the iterative dampening would deliver the purple curve).

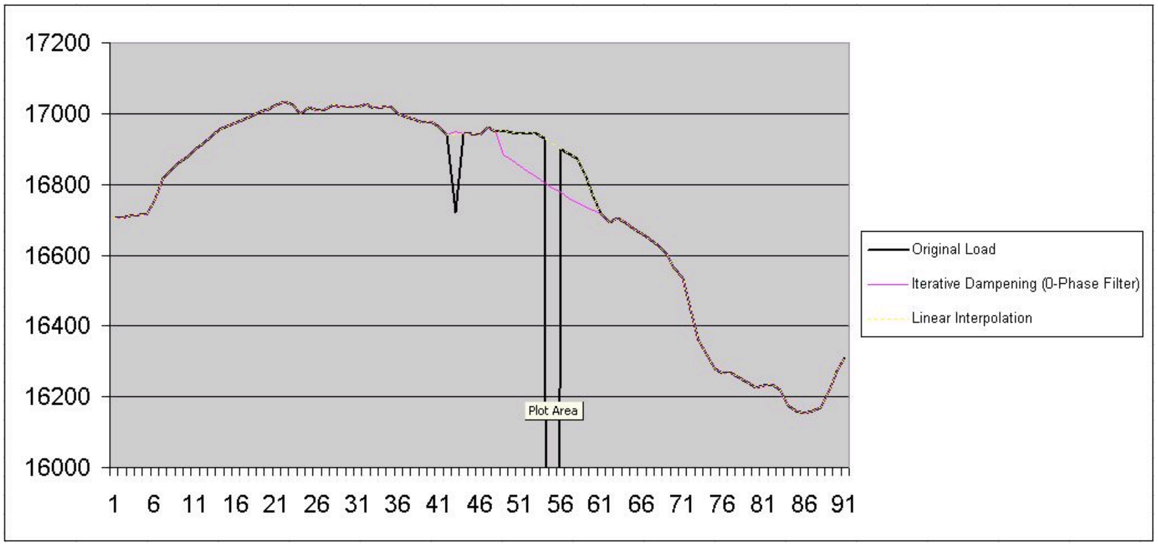


Figure 6. Comparing macro-level spike removal methods

Figure 7 shows details of the macro level spike filtering process. The zero-phase filter on the five minute signal first detects the macro-spikes. Once again, comparing the absolute difference between the raw and the filtered data (DIF2) with a configurable thresholdTHD2 (the default value of THD2 is 200MW), makes it possible to flag the windows containing such spikes.

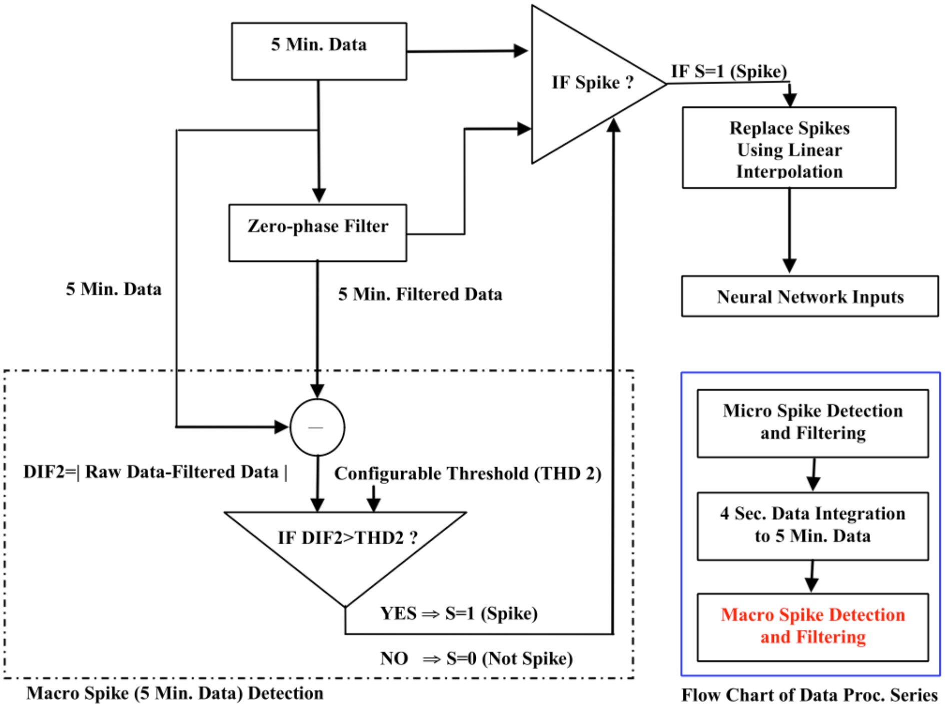

5 Min. Data
IF Spike ?
IF S=1 (Spike)
Replace Spikes Using Linear Interpolation
Neural Network Inputs
Zero-phase Filter
5 Min. Data
5 Min. Filtered Data
—
DIF2=| Raw Data-Filtered Data |
Configurable Threshold (THD 2)
IF DIF2>THD2 ?
YES ⇒ S=1 (Spike)
NO ⇒ S=0 (Not Spike)
Macro Spike (5 Min. Data) Detection
Micro Spike Detection and Filtering
4 Sec. Data Integration to 5 Min. Data
Macro Spike Detection and Filtering
Flow Chart of Data Proc. Series

Figure 7: Macro spikes filtering

5.1.5 Multilevel wavelet decomposition

The very short term load has complicated features, which includes slowly changing signals (low frequency) and fast changing signals or transients (high frequency), with the low frequency and the high frequency respectively representing the “approximations” and the “details” of the signals.

Decomposing a time series $s(t)$ into low and high-frequencies has an added benefit. Let $R(s) = [min(s)..max(s)]$ be the dynamic range of the time series $s(t)$ where $min(s)$ and $max(s)$ are, respectively, the smallest and largest value in $s(t)$. The dynamic range of the high frequency component is often much smaller than the dynamic range of the original signal. Artificial Neural Network always expect their input to be in the [0..1] range. It is therefore necessary to map any input whose range is [min..max] into [0..1]. Clearly, the larger the dynamic range of the input, the coarser the resolution when the input is normalized to [0..1]. For instance, consider the time series $s(t)$ = [4,8,2,1,2,5,11] with a dynamic range $R(s)$= [1..10] and the wish to learn the mapping [1 → 50, 2 → 90 ,3 → 95, 4 → 100, 5 → 200,...]. Clearly, a neural network must first map all possible inputs into the 0..1 range. Therefore [4,8,2,1,2,5,11] is mapped to [0.3,0.7,0.1,0.0,0.1,0.4,1.0] and the two inputs 0.3 and 0.4 are quite a bit apart (0.1 to be precise) which helps the learning algorithm to separate its responses to these two stimuli. If, instead, the input is [4,8,2,1,2,5,1000000], the dynamic range R(s) is much larger at [1..1000000]. Consequently the two inputs 4 and 5 are now mapped to two numbers in the [0..1] that are very close to each other and will be harder to deal with during learning.

The Daubechies wavelet is a decomposition tool that effectively breaks a time series into low and high frequency components that correspond to disjoint frequency bands with narrower dynamic ranges. Separating the aggregate load into these components allows the use of multiple neural networks (one per component) to capture the features of each and to forecast them independently as shown in Figure 4. An isolated component of the load can be predicted with greater precision. The final forecast can be obtained by adding the forecasts for all components.

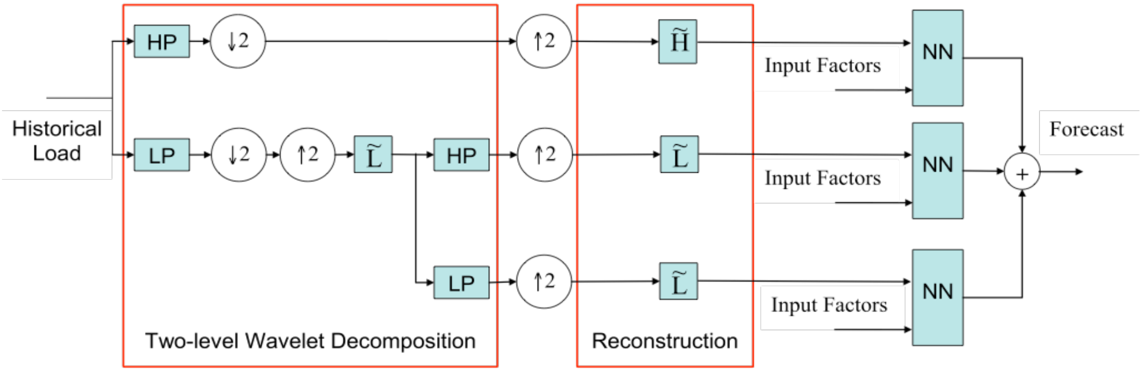


Figure 10: Structure of Two-level Wavelet Decomposition

Daubechies wavelet forms an entire family of decompositions that differ in the fixed cut-off frequencies that “separate” the bands. Picking a specific degree for the Daubechies wavelet yields a cut-off frequency. The actual degree of the Daubechies wavelet in the implementation was chosen from a set of experiments as Daubechies-2.

A wavelet decomposition breaks down a signal into a low and a high frequency component which are themselves time series. Clearly, the wavelet decomposition can therefore be applied recursively to further decompose a signal into additional bands. The *level of a decomposition* refers to the number of recursive decomposition for the low-frequency component. A one level decomposition of $s(t)$ yields two time series *low*(t) and *high*(t). A two-level decomposition further breaks down *low*(t) into *lowlow*(t) and *lowhigh*(t). Therefore a two-level decomposition yields three time series (*lowlow(t)*, *lowhigh(t)* and *high(t)*) that can be independently predicted.

A noteworthy property of the wavelet pertains to which entries of the original time series *s* influence a value at time t in *low*(t) and *high*(t). The t^{th} entry of low or high depends not only on $s(t)$ but also on s(t)’s neighboring entries, both *backward and forward.* Specifically, there is a window of *influence* surrounding entry t and expanding k slots in both directions. The length 2.k of this window is a function of the Daubechies degree and of the number of levels of recursive decomposition.

$K = D*(2^L)-2^{(2^\wedge L-1)}$

During testing, we also implemented a four-level decomposition that yields five components. However, testing results show that it is not as good as the results from two level decomposition.

5.2 The Artificial Neural Networks (ANNs)

A core component of the VSTLF system is a program module implementing a multi-layer perceptron network. This is an artificial neural network that can be trained to learn future load as a function of processed historical data. The neural network implementation used in our system is completely identical (same Java classes) to the implementation used by the STLF project. As a result, the description of the processes underlying the network's functionality appearing in this design note mirrors exactly that appearing in the documentation of the previous STLF system.

The implementation uses a set of five artificial neural networks to perform each forecast. Within the set, the network responsible for the lowest frequency band has a unique input format, while the other four share the same configuration. The two configurations present in a set of networks are depicted below.

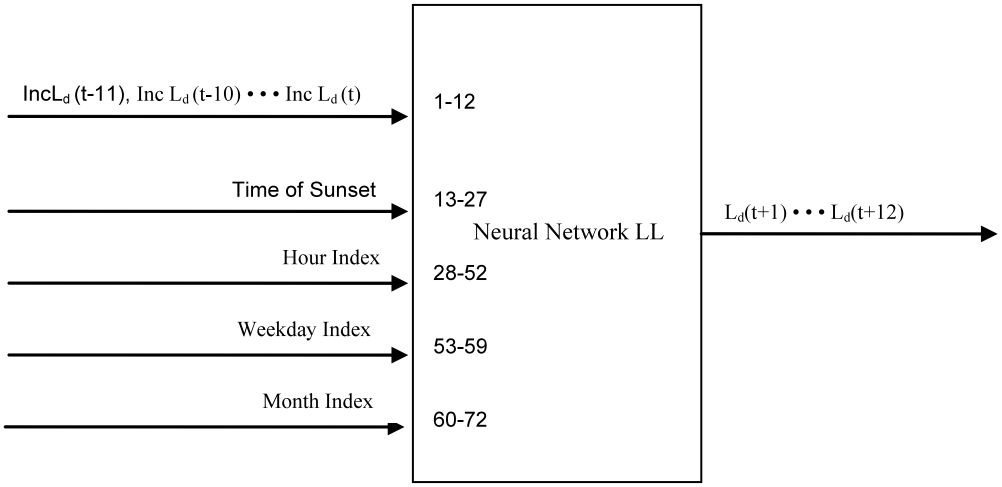


Figure 9: LL - Neural Network for VSTLF

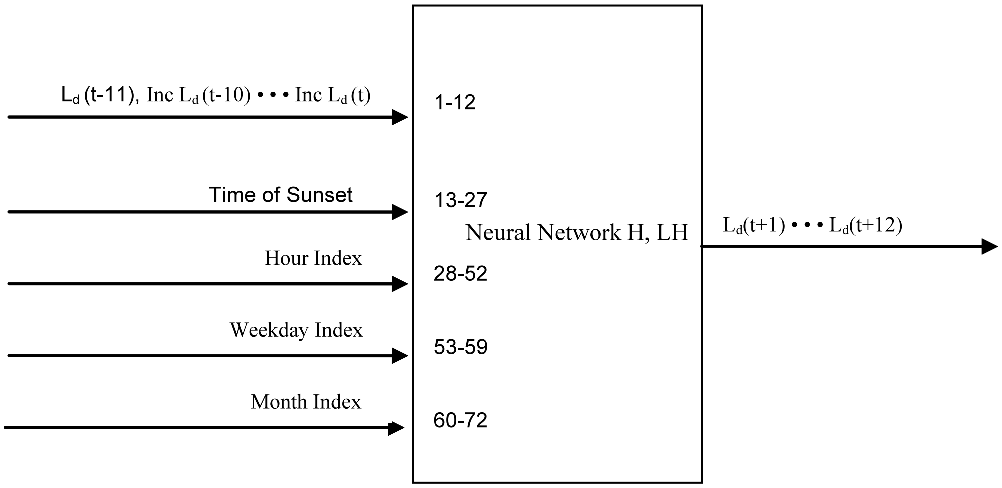


Figure 10 - Layout of the four High Frequency neural networks.

As shown in figure 9, the neural network predicts the 5 minute load for next hour by using the load of previous hour, as well as month, weekday and hour indices. The input also includes a 15-input encoding of the time of sunset on the current day, which will be described in the following section. Because there is no input indicating the time within the hour, UConn's VSTLF implementation uses a separate set of five networks to forecast from each five minute offset within the hour. As a result, the system maintains twelve sets of five networks each.

In Figure 9, the symbol "Inc" defines relative increment in load by the following equation

$$IncL_d(t) = \frac{L_d(t) - L_d(t-1)}{L_d(t-1)}$$

where t represents the sampled point every 5 minute, subscript d is the day index.

This relative increment is used in the input for the LLLL network. Both the previous hour's load and the similar hour's load are expressed in the form of a relative increment, expect for one absolute load value ($L_d(t\text{-}11)$), which is used to indicate the magnitude of the load, a piece of information not carried by the series of relative values.

The rationale supporting the use of relative increment in load as inputs is that even if the current weather conditions are different from those used for training, the increment may still have similar trend or curvature.

Computation. The ANN produces an output vector y that is a function of the input vector x, i.e., it computes $y = f(x)$. The weights in the network characterize the function f being computed. The evaluation of f proceeds in the standard manner with the signal proceeding through the layers of the network towards the output. Each neuron k proceed by computing

$$\sigma_k = \sum_{i \in I(k)} o_i \times w_{l,i,k}$$

where $I(k)$ is the set of inputs to neuron k, o_i is the output of neuron i that connect to neuron k via weight w_{lik}. The neuron then uses the sigmoid function to compute its own output o_k with

$$o_k = \frac{1}{1 + e^{-\sigma_k}}$$

The computation proceeds through the layer for a total cost (time) linear in the size of the network (in its number of weights).

Training. Training a multi-layer perceptron is called supervised learning and is typically done with a back-propagation algorithm that iteratively corrects the weights of the network until its output matches the desired output. Given a training set T containing pairs of (input, desired-output), the training algorithm iterates over all pairs and submit the input to the network to compute its output. It then compares the actual and desired output and obtains an error measure (typically, mean square error) and uses the error within a backward pass over the layers of the network and tweaks the weights to ensure that the next prediction for this input will yield a smaller error. The process is repeated over all input-output pairs until a target error is achieved. The back-propagation algorithm is textbook material and completely standard.

Normalization. The ANNs receive input and produce output values in the interval [0,1], and in order to perform its computation on data lying outside of this interval, the networks must receive the data in a normalized form.

There is a simple process through which a set of values can be normalized. First, the minimum and maximum values across the set, x_{min} and x_{max} are found. Once these are obtained, the normal form of each x in the data set can be found through the following equation.

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Once an ANN has completed its computation, producing a set of values in the [0,1] interval, the resulting values must be denormalized, or expanded into the interval from which the ANN's input data originally came. If x_{max} and x_{min} have been retained, then this can be done easily using the inverse of the function above.

In order for the normalized values used in the VSTLF system to be meaningfully representative of the original values from which they come, the normalization of any value must also be consistent with each other normalization that the system performs. In order to achieve this consistence, the system normalizes every value using the same upper and lower bounds, which need only to be values defining an interval that contains any value than an input to the ANN may take.

5.3 Sunset Time

Testing of the system described above, revealed that the largest errors were occurring near dusk. This observation motivated the addition of a set of inputs to capture dusk time where additional load tends to manifest (e.g., lightning load). The neural networks was augmented with information about the precise time of sunset on the current day and experiments showed improvements in the quality of forecasts one hour into the future. The revised design is integrated into the final system, as illustrated in the above neural network diagrams.

The precise time of sunset on the day on which a forecast is made can be computed with the help of a software package that implements the algorithm described in the US Naval Observatory's *Almanac for Computers*, (1990; Nautical Almanac Office, United States Naval Observatory, Washington, DC 20392).

The encoding of the sunset time uses fifteen input nodes, and is as follows:

(1) Each of the 15 input values are, by default, zero.

(2) If the hour index (a number 1-24) of the current day’s sunset is different from that of the current time by more than one, all values are left at zero.

(3) If the hour index of the current day’s sunset is one less than that of the current time, then the first of the 15 values is toggled to one (hour before sunset).

(4) If, instead, the hour index of the current day’s sunset is one greater than that of the current time, then the second of the 15 values is toggled to one (hour after sunset).

(5) If the two indices agree precisely, then the third input value is toggled to one (hour of the sunset).

(6) Additionally, exactly one of the remaining twelve values is also set to 1 to represent the offset within the current hour (in units of 5 minutes) at which the sunset occurs.

5.4 Alternative Methods

The preliminary design for UConn's VSTLF system incorporated the methods described below. During testing, however, it was concluded that the additional information did not contribute to forecast quality. As a result, these methods were removed from the forecast process in order to decrease system complexity.

5.4.1 ACE/Frequency Analysis

Initially, the forecast method was to include neural network input representing the grid's frequency and ACE time series over the previous hour. However, tests in which these data were included showed no improvement in forecast quality. As a result, the use of ACE/frequency information in VSTLF has been abandoned for the present time.

5.4.2 Similar Hour Selection

Another input that was ultimately removed from the design was the load of a similar hour. The load for a historical hour similar to the target hour for a forecast was to appear among the input to the multilayer perceptron corresponding to the lowest frequency band. The strategy for choosing the best similar hour is to select the best similar day as it is done in the STLF system, and then select the corresponding hour of that day.

The best similar day, for a given day *d*, is the day among all previous days with the same weekday index as *d*, which has the lowest similarity index with *d*. The similarity index can be computed with the formula,

$$Similarity = \frac{\sum_{i=1}^{24}(|W_T - W_S| - |H_T - H_S|)}{24}$$

In which *W* denotes the wind-chill temperature and *H* the humidity for each hour of a day, *t* denoting the target day and *s* the day whose index is to be computed.

This input, like the ACE and Frequency, was shown not to contribute to forecasts quality.

6 High Level Requirements

6.1 High Level Data Requirements

In order to perform its function, the VSTLF system requires a data store containing the recorded load for a period of history leading up to the present, as well as in which to store the load forecasts that it creates, and may need a database in which to store the weights associated with the connections in the artificial neural networks that it uses. To meet these needs, VSTLF uses the Java-friendly Perst library for persistent object storage. The data that are to be stored by the system in its Perst databases will be discussed in greater detail in sections 7 and 8.

6.2 High Level Application Design

The VSTLF system is implemented by a collection of Java classes. These classes are responsible for all of the system's interaction with its operating environment, and implement all the data transformations that occur during a load prediction, as well as the artificial neural network and the API through which the system is used. Each class is discussed in detail in the following sections.

7 Functional Design

7.1 System Functions

The sole function of the VSTLF system is to produce regular load forecasts. Each forecast should consist of twelve values describing the load over a one-hour period. A forecast for the next hour should be produced every five minutes.

7.1.1 Prediction

The forecast module executes every 5 minutes (from the top of the hour) and its purpose is to produce a forecast for the next 60 minutes. This forecast is a collection of 12 predictions, one for every 5 minute blocks that follows the time of prediction.The prediction module assumes that the data pre-processing of the 4-second data stream has already occurred. Specifically, it assumes that all 5-minutes load-aggregates are available and fully pre-processed (i.e., micro and macro filtering already took place). It then retrieves the load for the previous hour (twelve observations at 5 minutes intervals), the load of a similar hour (again, twelve observations) where similarity is based on same-hour within a day and same day within a week. The loads are passed through the wavelet to obtain their corresponding 5 components and the inputs of the five network are *normalized* in the [0..1] range. The ANN produces a prediction that is then de-normalized (via a simple affine transform that inverts the initial normalization) and aggregated to obtain the final forecast.

7.1.2 Training and Update

The previous STLF system provided the user with training and update functions that allowed the system's neural networks to learn to forecast with greater accuracy based on new historical information. In the STLF system, these were manually initiated operations that system operators could choose to perform. In the VSTLF system, however, neither of these operations is included in the interface for the forecasting engine. The incremental update process is handled automatically by the system, which updates a subset of its neural networks once during each five minute period. Training is initiated manually by the operator with a stand-alone application that generates the neural network files to be loaded by the real time engine. This loading process is discussed in further detail further below.

7.2 Database Schema

The Perst persistent object database library is used to meet the VSTLF system's data storage and retrieval needs. The system expects a stream of recorded load data, which should be provided at approximate four-second intervals via calls to methods on the object representing the main body of the system. To limit the amount of four second data to be kept on disk, the system periodically integrate the 4s data it has accumulated as discussed in section five. The result of the integration is stored in a Perst file allocated for the historical 5-minute load. A second Perst file is kept in order to store the filtered form of the 5-minute load signal produced by the macro filter, also discussed in section five.

7.3 Java Classes

Here follows a brief summary of the Java classes created for the implementation of VSTLF. Each group of classes mentioned here is discussed in detail within the following section.

7.3.1 Database

A set of classes will use the Perst object database library to store and retrieve the data necessary to the VSTLF system's functions

7.3.2 Artificial Neural Network

A small set of classes implements the multilayer perceptron, with all methods required for it to learn and compute a function.

7.3.3 Data Transformation

Java classes will also implement all data preprocessing tasks, including the removal of spikes from load series, aggregation from 4-second to 5-minute resolution, and normalization of input to the ANNs.

7.3.4 Forecast Engine

A higher level set of classes uses the functions provided by the database, ANN, and data transformation classes to implement the primary function of VSTLF, processing input data in real time and producing a new forecast every five minutes.

8 Software Design

8.1 Database Schema

The VSTLF system does not interface with a relational database. Instead, it keeps its own database in the form of a set of Perst storage files.

To function, the system requires only two data series to be stored – raw load and filtered load (both per 5 min), and one Perst object is maintained by the Java program to store both. Users of the system are not required to directly manipulate these Perst databases. Instead, they periodically submit new data using method calls. The relevant methods are all detailed in the following section, and appear in the description of the VSTLFEngine class.

Each of the input data submitted to the system in this way is stored using a Perst TimeSeries object. Specifically each element of the raw and filtered five minute load TimeSeries includes fields denoting

(1) TimeStamp

(2) Load

8.2 Java Packages

edu.uconn.vstlf.realtime

This package contains the primary public classes in the UConn VSTLF release. It provides the real-time data processing for vstlf including the four second data integration, micro filtering, five minute integration, macro filtering and predictions.The classes described here provide the API through which the VSTLF system can be used.

VSTLFEngine

A VSTLFEngine object is the central component of the system. To begin the forecasting process, create a VSTLFEngine object and begin supplying data using its addObservation method. The engine, once started, expects a continuous stream of data. In return it will initiate a similar stream of callbacks announcing various events, including data aggregation, forecasts, and the occurence of holes in the data provided.

Method List
VSTLFEngine(**String** rawStore, **String** filtStore) Object constructor. Creates a new forecasting engine. The two arguments are file names for the two Perst databases that the object expects to find when it starts running. These files should both contain the five minute load for one day leading up to the time at which the engine is initialized.
void startup() Sets the engine running. From the point at which this method is called, the engine will expect a continuous stream of input. It starts from the current time
void startup(Date at) Sets the engine running. From the point at which this method is called, the engine will expect a continuous stream of input. It starts

from a specific date ‘at’.
void startCollecting() The engine starts collecting the realtime data. It micro filter the four second data and integrate it into five minute data.
void startProcessing () The engine starts extracting the five minute data and use it to generate the forecas. (calling startup is the same as first calling startCollection then calling startProcessing)
void addObservation(**VSTLFNotificationCenter** center, Date at, **double** val) Feed the raw real-time data to the engine. It also extract any messages and dispatch them to the notification center.

FourSecondProcess

The four second process reads from the raw data stream, integrates them into four second load. Then It uses micro filter to filter the load and combines them into five minute loads.

Method List
FourSecondProcess(PCBuffer<VSTLFMessage> notif, PCBuffer<VSTLFObservationPoint> buf,PCBuffer<VSTLF5MPoint> out) Object constructor. Creates a FourSecondProcess object. Notif is the message queue for recording the events happened. Buf is the raw data from VSTLF engine. Out is the generated five minute loads.
void prepare(Date st,int rate) Integrate the raw data into four second data every 4s.
void run() Micro filter the four second loads and combine them into five minute loads every 5min.
void setFilterThreshold(double t) Set the threshold for the micro filter..

FiveMinuteProcess

The five minute process reads the raw five minute loads, use macro filter to filter them. Then it use the filtered data to update the neural network and predict the loads in one hour every 5 minutes. It also stores the raw five minute loads and filtered five minute loads into the perst database.

Method List

FiveMinuteProcess(PCBuffer<VSTLFMessage> notif, PCBuffer<VSTLF5MPoint> buf, PowerDB db) Object constructor. Creates a FiveMinuteProcess object. Notif is the message queue for recording the events happened. Buf is the raw five minute data from FourSecondProcess. db is the perst database for recording five minute loads.
void run() Macro filter the raw five minute loads. Update the ANN and predict the loads in one hour every 5 minute. The raw and filtered loads are also recorded in perst database
void setFilterThreshold(double t) Set the threshold for the macro filter..
Series[] inputSetFor(Date t,Calendar cal, PowerDB pdb) Provide the input data for prediction in neural networks. It applies multilevel wavelet transformations here.
Series[] targetSetFor(Date t,Calendar cal, PowerDB pdb) Provide the target data for updating in neural networks. It applies multilevel wavelet transformations here.

VSTLFNotificationCenter

It is an interface class for receiving messages posted by FourSecondProcess and FiveMinuteProcess. To handle these messages a class must implement this interface.

Method List
void fourSTick(Date at, double val, int nbObs) A four second load is generated.
void fiveMTick(Date at, double val, int nbObs); A five minute load is generated.
void didPrediction(Date at,double[] val) A prediction is done.
void beginTraining(Date at, Date from, Date to) Begin the training of neural networks.
void endTraining(Date at, Date til) The training of neural networks end.
void missing4SDataPoint(Date at)

A four second load does not appear from the source.
void missing5MDataPoint(Date at) No load is generated in five minute. void refined4SPoint(Date at, double oldVal, double newVal); A micro filtered four second load is generated.
void refined5MPoint(Date at, double oldVal, double newVal) A macro filtered five minute load is generated.
void exceptionAlert(Exception e) A exception occurred.

IsoVstlf

An IsoVstlf contains a VSTLF engine. It continues feeding four second raw loads to the engine and outputs the messages to a DataStream object. It does not output any information to user. It is used by the edu.uconn.vstlf.shutil.RunHeadless class.

Method List
IsoVstlf(boolean coldstart, String dbName, String currentDataXml, String dailyDataXml) Object constructor. Creates an IsoVStlf object that will setup the datasoure files.
void init() Initialize the object. Setup the database file, the VSTLF engine and start the engine.
void setTestTime(Date tt) Set the test time that is the intial timestamp of the VSTLF engine pulse. This method can be used to test historical data.
void setClockRate(int rate) Set the rate to feed raw data into the VSTLF engine. In practice it should be set to 4s. (4000 milliseconds)
boolean run(Date at) The implementation of pulse action. It feeds raw data into the VSTLF engine when a pulse is triggered. The rate of triggering pulse is set by the setClockRate method.

edu.uconn.vstlf.batch

Audition

This class contains several methods for generating the analysis of the VSTLF system.

Method List

static void main(String[] args) Generate the max error, min error and Mape of the vstlf system. It is called by RunAudition.java in edu.uconn.vstlf.shutil
static void printOptions() Print the possible options of command arguments.

VSTLFTrainer

This class contains methods for training the neural networks.

Method List
static double[][] test (String loadFile, Date stTest, Date edTest, int lo, int up) Test the performance of trained ANNs.
static void train(String loadFile, Date stTrain, Date edTrain, int lo, int up) Train the ANNs using the historical data in a specified period.
static Series[] inputSetFor(Date t,Calendar cal, PowerDB pdb) Define the input vector set.
static Series[] targetSetFor(Date t,Calendar cal, PowerDB pdb) Define the target vector set
Date lastTick(**int** inc, Date t) Returns the latest date that is both an even multiple of inc seconds and not after the given date t.
Date now() Returns the current time

edu.uconn.data

Calendar

An extension of the java.util.Calendar designed to more easily manipulate the Java Date objects used within the VSTLF system.

Method List
Calendar(String locale) Object constructor. Creates a calendar that will created and manipulate dates in accordance with the local calendar for the specified locale name.
static Date newDate(**int** year**, int** month**, int** day**, int** hour**, int** min**, int** sec) Creates a new date object with the given parameters. Java indexes the months beginning

with zero.
int getYear(Date t), getMonth(Date t), getDate(Date t), getHour(Date t), getMinute(Date t), getSecond(Date t), getDayOfWeek(Date t) Each of these methods returns an integer corresponding to the requesting value for the given date..
Date addYearsTo(Date t, **int** n), addMonthsTo(Date t, **int** n), addDaysTo(Date t, **int** n), addHoursTo(Date t, **int** n), addMinutesTo(Date t, **int** n), addSecondsTo(Date t, **int** n) Each of these months returns the date that occurs n time-units after the given date t.
Date lastTick(**int** inc, Date t) Returns the latest date that is both an even multiple of inc seconds and not after the given date t.
Date now() Returns the current time

edu.uconn.data.doubleprecision
The key class in this package is the Series class, which is used to handle the bulk of data preprocessing, such as spike-filtering, discrete wavelet transform, and integration from 4second to 5minute resolutions.

Series

A Series object stores a sequence of doubles in an array. Each element of the sequence can be accessed using the element method.

Method List
Series(**double**[] array) Object constructor. Creates a Series object containing a copy of the contents of ‘array’.
Series(**double**[] array, **boolean** copy) Object constructor. Same as above but, if ‘copy’ has the value false, the given array itself is used, instead of a deep copy.
double element(**int** i) Retrieves the element of the series at index i. Indices start at 1. If the specified index does not fall between 1 and the length of the series, then the value is computed using a mirror of the stored series.
int length() Returns the number of elements in the series

(also the index of the last element.

Series append(Series s)

Returns a new series with the contents of s appended to the contents of this

Series patchSpikesLargerThan(double t)

Uses the method described in the section on macro spike filtering to remove spikes with height exceeding the given threshold. Returns a new series with the filtered data.

Series[] daub4Separation(**int** levels, **double**[] decCoefs, **double**[] recCoefs)

Returns the result of a daubechies wavelet decomposition of this series. The object returned is an array of Series objects. Each series is a component of the original signal. They are ordered from highest in frequency to lowest.

double elementM(**int** i)

Retrieves the element of the series at index i. Indices start at 1. If the specified index does not fall between 1 and the length of the series, then the value is computed using a mirror of the stored series.

Series subseries(int i, int j)

Return the subseries start from index i to index j.

Series reverse(int i, int j)

Reverse the series

Series LowPassWithShift**(int w)**

Returns a new series produced from this series using a "shifty" moving average of length w over this series.

Series pad(int front, int back)

Returns a mirror padded series starting from front to back.

To facilitate the processing of a array of data. This package also contains a lot of auxiliary function for producing mean, minimum, maximum data and so on. These function all derives from the Function Class.

edu.uconn.database
This package contains the interface class for the power database PowerDB. It is a super class for PerstPowerDB.

PowerDB

Method List

void addLoad(Date t, **double** v) Adds or overwrites the given time/value pair in this database.
void open() Opens this database object for reading and writing.
void close() Closes this database, commiting any pending changes.
double getLoad(Date t) Returns the load value stored in this database for the specified timestamp.
Series getLoad(Date st, Date ed) Returns a Series object containing the stored load values for the interval (st,ed].
void startTransaction(); Start a database transaction
void endTransaction(); Commit a database transaction
Series getLoad(Date st, Date ed) Returns a Series object containing the stored load values for the interval (st,ed].
void fill(String s, Collection<LoadData> set) Fill the database with a collection of data. s indicates the data type: raw data (before the macro filter) or filtered data (after the macro filter)
Date begin(String s) Returns the earliest date that contains data of type ‘s’
Date last(String s) Returns the last date that contains data of type ‘s’
Date last(String s) Returns the last date that contains data of type ‘s’
Series getForecast(Date t) Get a Series object containing the forecast on ‘t’.
double[] getForecastArray(Date t) Get an array containing the forecast on ‘t’.
void addForecast(Date time, Series forecast) Add the forecast completed on ‘time’.
void addForecastArray(Date time, double[] forecast)

Add the forecast completed on ‘time’.
void addForecastArray(Date time, double[] forecast) Add the forecast completed on ‘time’.
void addForecastArray(Date time, double[] forecast) Add the forecast completed on ‘time’.

edu.uconn.database.perst
The sole purpose of this package is to allow the storage and retrieval of load time series in a persistant file using the Perst object database library. The package contains one main class, “PerstPowerDB” which uses several helper classes to interface with the database library. This main class is documented here.

PerstPowerDB

Implemented using a Perst storage that contains a single Timeseries object, this class is designed to store a series of numeric values (double) to represent load. Each value in the series is associated with a timestamp and it is assumed that the interval between two consecutive entries in the sequence will be consistent throughout the series. The length of this interval, in seconds, should be specified when the PerstPowerDB object is constructed.

Method List
PerstPowerDB(String filename, **int** inc) Object constructor. The arguments specify the name of the storage file on disk, and the number of seconds between entries in the sequence. If the specified file does not already exist, a new, empty database will be created.
static PerstPowerDB fromXML(String srcFile, String dstFile **int** inc) Adds the series of time/value pair supplied in the XML file ‘srcFile’ to the perst store specified by ‘dstFile’. Inc indicates the number of seconds between timestamps appearing in the destination file. If necessary, the data in the XML will be aggregated in order to match the resolution of the destination file.
void addLoad(Date t, **double** v) Adds or overwrites the given time/value pair in this database.
void open() Opens this database object for reading and writing.
void close() Closes this database, commiting any pending changes.

double getLoad(Date t) Returns the load value stored in this database for the specified timestamp.
Series getLoad(Date st, Date ed) Returns a Series object containing the stored load values for the interval (st,ed].

edu.uconn.vstlf.gui

This package provides the GUI (Graphical User Interface) of the VSTLF system. The main class is the IsoVstlfGui. It feeds the raw loads from the source file into a VSTLF engine and outputs the prediction result and messages generated in the whole process. It plays the same role as edu.uconn.vstlf.realtime.IsoVstlf. The only difference is that it will display the results in the GUI.

Method List
IsoVstlfGui(boolean coldstart, String dbName, String currentDataXml, String dailyDataXml) Object constructor. If coldStart is true the historical data are from xml files. Otherwise they are from Perst databases. The currentDataXml is the file for the real-time data. The dailyDataxml contains the five minute data in the last 24 hours to start up the system.
void init() Start the VSTLF engine and feed data into the engine. It also configure the GUI for displaying the results.
void setTestTime(Date tt) Set the test time that is the intial timestamp of the VSTLF engine pulse. This method can be used to test historical data.
void setClockRate(int rate) Set the rate to feed raw data into the VSTLF engine. In practice it should be set to 4s. (4000 milliseconds)
boolean run(Date at) The implementation of pulse action. It feeds raw data into the VSTLF engine when a pulse is triggered. The rate of triggering pulse is set by the setClockRate method.
void addLoad(Date t, **double** v) Adds or overwrites the given time/value pair in this database.
void open() Opens this database object for reading and writing.

void close() Closes this database, commiting any pending changes.
double getLoad(Date t) Returns the load value stored in this database for the specified timestamp.
Series getLoad(Date st, Date ed) Returns a Series object containing the stored load values for the interval (st,ed].
void fourSTick(Date at, double val, int nbObs) Display the message that a four second load is generated.
void fiveMTick(Date at, double val, int nbObs); Display the message that a five minute load is generated.
void didPrediction(Date at,double[] val) Display the message when a prediction is done.
void beginTraining(Date at, Date from, Date to) Display the message at the beginning the training of neural networks.
void endTraining(Date at, Date til) Display the message that the training of neural networks end.
void missing4SDataPoint(Date at) Display the message that no load comes from the source in four seconds.
void missing5MDataPoint(Date at) Display the message that no load is generated in five minutes. void refined4SPoint(Date at, double oldVal, double newVal); Display the message that a micro filtered four second load is generated.
void refined5MPoint(Date at, double oldVal, double newVal) Display the message that a macro filtered five minute load is generated.
void exceptionAlert(Exception e) Display the message that an exception occurred.

edu.uconn.vstlf.shutil
This is the package that provides the shell utilities for running the VSTLF systems in various configurations.

Reset

Method List

static void main(String[] args) Reset the vstlf system. It will delete the .vstlf file which contains the execution history of the vstlf system.

RunAudition

Method List
static void main(String[] args) Run the audition module in vstlf.batch. Output the statistic analysis of the whole system.

RunGUI

Method List
static void main(String[] args) Create an IsoVstlfGui object and run the gui. The main entrance of the vstlf system.

RunHeadless

Method List
static void main(String[] args) Create an IsoVstlf object and run the system heedlessly. Do not out the results.

RunTraining

Method List
static void main(String[] args) Use VSTLFTrainer to train the ANNs.

RunValidation

Method List
static void main(String[] args) Use VSTLFTrainer to test the performance of the ANNs

edu.uconn.neuro
This is the package that provides functionality pertaining to the multilayer perceptron network. Two java classes in this package are used by the VSTLFEngine to create and train neural networks that can be used in the VSTLF process.

SimpleFeedForwardANN

This class implements the multilayer perceptron network. It constructs itself out of FeedForwardNeurons, and supports methods to perform its feed forward computation, as well as the backpropagation learning algorithm.

Method List
SimpleFeedForwardANN(**double**[][][][] weights) Object constructor. Given a three dimensional matrix of weights, constructs a new FeedForwardANN with the number of layers and nodes per layer dictated by the dimensions of the matrix, and with weight values dictated by the contents of the matrix.
static SimpleFeedForwardANN newUntrainedANN(**int**[] layers) Creates a new FeedForwardANN with a layer of neurons for every value in the given layers array, each layer containing the number of neurons specified by the corresponding value.
double[] execute(**double**[] input) Copies the contents of the given array into the input layer of the ANN, and then performs the ANN's feed forward computation routine. When the method returns, the net's output layer will store the result of the computation.
void update(**double**[] desiredOutput) Performs the backpropagation learning algorithm to update the weights of the network using the difference between the ANN's most recent output and the given desired output.
void train(**double**[][] in, **double**[][] targ, int sec) Trains the network by repeatedly iterating over the given sequence of input/target pair until the given number of seconds has elapsed.

In order to instantiate the artificial neural network class, the constructor must receive a three dimensional array containing all of the network's connection weights. The constructor infers the topology of the network to be constructed from the form of the weight matrix, and the way that the matrix is assembled determines precisely the layout of the new ANN.

Each cell in the matrix has three indices, and contains the value of the weight that the network is to place on the connection specified by the indices. These correspond to (1) the identifier of the layer of the network in whom lies the neuron that maintains the weight in question, (2) the identifier of the neuron within that layer, and (3) the identifier of the connection coming into the specified neuron on which the weight is to be placed.

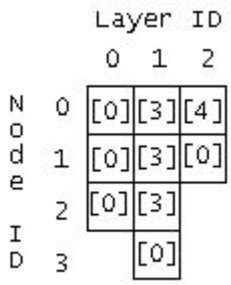
Because the matrix contains every one of the network's connection weights, and because the values stored inside the matrix are indexed in this way, the constructor is able to infer the dimensions of the network from the shape of the array. The number of layers in the network is clearly equal to the length of the weight array along its first dimension. The ith layer in the network contains a number of neurons equal to the number of one-dimensional arrays stored in the ith cell of the 3D array. Each node in the layer corresponds to one of these one-dimensional arrays, and is to maintain one connection for each of the weight values contained therein. The length of any of these one-dimensional arrays must be equal to the number of connections from the previous layer, and therefore equal to the number of 1D arrays in the previous cell of the 3D matrix.

There are a couple of things worth noting about the way the matrix must be assembled in order to produce a valid network.

First, the neurons in a network's input layer maintain no incoming connections. They must, however, still be represented in the matrix in order for the input layer to be properly constructed. The first cell of the matrix, then, must be an array of one-dimensional weight arrays, each with length zero.

The other factor that must be taken into consideration is that each layer of the ANN under construction will contain one "bias node" in addition to the set of nodes specified in a description of the neural network to be simulated. The output of any of these biases will be weighted and considered by every neuron in the subsequent layer, and so each bias must also be represented in the matrix by an array of length zero.

As an example, the 3D weight matrix used to construct a simple ANN with 2 inputs, a single output, and three hidden neurons might be represented as follows.



		Layer ID		
		0	1	2
Node ID	0	[0]	[3]	[4]
	1	[0]	[3]	[0]
	2	[0]	[3]	
	3		[0]	

ANNBank

An object of this class maintains an array of neural network objects. Such an object is used by the VSTLFEngine to represent each of the twelve sets of networks maintained by the system. The class member functions facilitate the execution, training, or update of the entire array with a single method call.

Method List
ANNBAnk(**int**[][] lyrSz) Object constructor. Receives an array of network topologies and creates the corresponding array of networks.
double[][] execute(**double**[][] input) Receives a series of one input for each ANN and returns a series containing the output corresponding to each.
void update(**double**[][] desiredOutput) Performs the operation on all ANNs in the bank, as with execute.
void train(**double**[][][] in, **double**[][][] targ, int[] sec) Performs the operation on all ANNs in the bank, as with execute.

9 Software Installation and Maintenance

Since UConn's VSTLF system comprises only a library of Java classes, the installation process is minimal. The API provided by the classes in this release is available to the user whenever the included edu.uconn packages are found on the Java class path. Once these packages are in place, the user is free to create and use the VSTLFEngine.

For the VSTLFEngine constructor to succeed, however, there must be Perst files describing its twelve sets of neural networks in the “anns/” subfolder of the current working directory. These files are created using the Trainer utility described in the previous section.

To train the networks

1) Run the training utility, passing start and end dates for the training period as arguments (the format of the command to run this program can be seen in the Trainer description, part of section 8.2). The amount of time required for this training process depends on the length of the training period. Periods exceeding one year can take several hours per set of networks.

2) Once the training is complete, twelve files are available
"bank0.ann", "bank1.ann", ..., "bank11.ann", in its working directory. These files should be copied to the directory in which the VSTLF system will be running, and will be loaded by any VSTLFEngine object when it is created.

10 User’s Guide

The user’s interface to VSTLF system consists of two classes – the VSTLFEngine and the VSTLFNotificationCenter. The use of these classes is described in detail in section

8.2. For simplicity, an overview of the system's use is included here. The user is expected to refer to the detailed class descriptions in the section above. Once the engine is started, the system requires a continuous feed of load observations (4 seconds data) and hourly weather updates to stay up to date.

To run VSTLF:

1) Create Perst Load databases "RawFiveMinuteLoad.pod" and "FilteredFiveMinuteLoad.pod", both containing five minute load data from August 2006 up to the present. The user can create each of these files by instantiating a new PerstPowerDB object and filling it with the data one value at a time using the object's addLoad method.

2) Instantiate the desired subclass of VSTLFNotificationCenter. A reference to a notification center should be included with every submission of new data to the VSTLF system. The referenced center will then process any notifications that the system has generated since the previous submission.

3) Instantiate a new VSTLFEngine object. This is the object to which all future calls will be sent.

4) From this point, every time a new load measurement is available (approximately each four seconds), make a call to the addObservation method of the VSTLFEngine object, passing the new measurement, a java.util.Date representing the measurement's timestamp, and a reference to the notification center that should handle any pending notifications.

5) Immediately after the stream of input has begun, the VSTLFEngine's startup method may be called. This will begin the continual process of handling input and generating notifications. I new forecast will appear among the notifications handled by the system every five minutes.

6) The user may do whatever he desires with the data contained in the notification objects that are passed to the VSTLFNotificationCenter. The interfaces presented by the various notification objects are detailed in section 8.2 of this document.

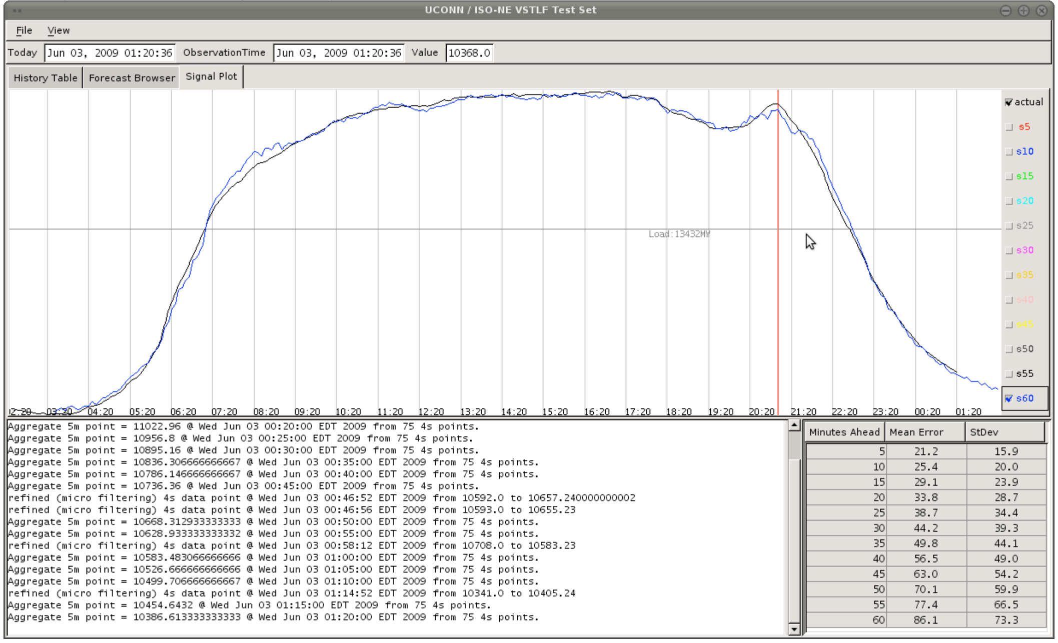
11 Testing

Validation tests for the UConn's VSTLF method were performed using accelerated simulations based on historical data rather than using the production system, which must process a load observation for every four-second tick, and cannot produce a large body of test results without significant amounts of stored data and CPU time.

The Java program used to run these accelerated tests is the same as the training utility described in the previous section, which terminates by printing the MAE obtained over the specified testing period. As a standard test, a set of perceptron networks was trained using the dataset of historical 5-minute data for the period between October 8, 2006 and January 1, 2008. The accuracy of forecasts was then measured across the period of time from January 1, 2008 to Jun 30, 2008, again using only preprocessed 5-minute load data.

The final stage of testing was performed using the finished VSTLF engine, run in accelerated time to produce a year's worth of predictions in a period of several days. The results of this final, validating test are included in a spreadsheet accompanying this document.

These accelerated tests of the realtime VSTLF engine were performed using a Java application developed specifically for this purpose. The application implements the Notification Center interface and runs a VSTLFEngine object at an accelerated clock rate, feeding it with observations from a historical database. The program stores each of the forecasts produced by the engine in a Perst database for later analysis, and reports information about the load and forecast histories through a Graphical interface developed using Java's Swing library.



Today Jun 03, 2009 01:20:36 ObservationTime Jun 03, 2009 01:20:36 Value 10368.0

```
Aggregate 5m point = 11022.96 @ Wed Jun 03 00:20:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10956.8 @ Wed Jun 03 00:25:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10895.16 @ Wed Jun 03 00:30:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10836.306666666667 @ Wed Jun 03 00:35:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10786.146666666667 @ Wed Jun 03 00:40:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10736.36 @ Wed Jun 03 00:45:00 EDT 2009 from 75 4s points.
refined (micro filtering) 4s data point @ Wed Jun 03 00:46:52 EDT 2009 from 10592.0 to 10657.240000000002
refined (micro filtering) 4s data point @ Wed Jun 03 00:46:56 EDT 2009 from 10593.0 to 10655.23
Aggregate 5m point = 10668.312933333333 @ Wed Jun 03 00:50:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10628.933333333332 @ Wed Jun 03 00:55:00 EDT 2009 from 75 4s points.
refined (micro filtering) 4s data point @ Wed Jun 03 00:58:12 EDT 2009 from 10708.0 to 10583.23
Aggregate 5m point = 10583.483066666666 @ Wed Jun 03 01:00:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10526.666666666666 @ Wed Jun 03 01:05:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10499.706666666667 @ Wed Jun 03 01:10:00 EDT 2009 from 75 4s points.
refined (micro filtering) 4s data point @ Wed Jun 03 01:14:52 EDT 2009 from 10341.0 to 10405.24
Aggregate 5m point = 10454.6432 @ Wed Jun 03 01:15:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10386.613333333333 @ Wed Jun 03 01:20:00 EDT 2009 from 75 4s points.
```

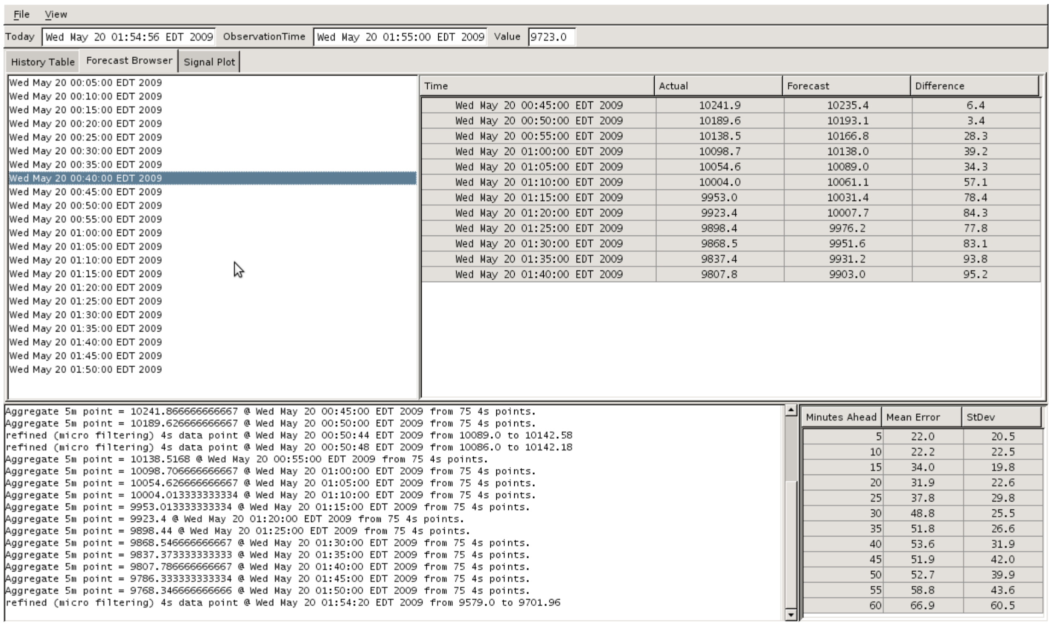
Minutes Ahead	Mean Error	StDev
5	21.2	15.9
10	25.4	20.0
15	29.1	23.9
20	33.8	28.7
25	38.7	34.4
30	44.2	39.3
35	49.8	44.1
40	56.5	49.0
45	63.0	54.2
50	70.1	59.9
55	77.4	66.5
60	86.1	73.3

Figure 11 The graphics interface used for system testing, including a plot of forecast vs actual load.

As shown above, the test application records the data found in observation and forecast notifications to construct a plot of actual versus forecasted load. A bank of checkboxes to the right of the plot allows the user to choose to view a plot of forecasts made any amount of time in advance, from five minutes up to one hour.

A table in the lower-right corner of the window displays information about the cumulative distribution of forecast error. Specifically, the table includes the mean and standard deviation (in MW) of forecast error over the entire history of the test. A log panel in the lower left of the display contains textual reports of every notification produced by the forecast engine.

The plot area may be replaced with a forecast browser menu as shown below, which allows the user to view any individual one-hour forecast, compared with actual load value.



Today: Wed May 20 01:54:56 EDT 2009 ObservationTime: Wed May 20 01:55:00 EDT 2009 Value: 9723.0

Wed May 20 00:05:00 EDT 2009
Wed May 20 00:10:00 EDT 2009
Wed May 20 00:15:00 EDT 2009
Wed May 20 00:20:00 EDT 2009
Wed May 20 00:25:00 EDT 2009
Wed May 20 00:30:00 EDT 2009
Wed May 20 00:35:00 EDT 2009
Wed May 20 00:40:00 EDT 2009
Wed May 20 00:45:00 EDT 2009
Wed May 20 00:50:00 EDT 2009
Wed May 20 00:55:00 EDT 2009
Wed May 20 01:00:00 EDT 2009
Wed May 20 01:05:00 EDT 2009
Wed May 20 01:10:00 EDT 2009
Wed May 20 01:15:00 EDT 2009
Wed May 20 01:20:00 EDT 2009
Wed May 20 01:25:00 EDT 2009
Wed May 20 01:30:00 EDT 2009
Wed May 20 01:35:00 EDT 2009
Wed May 20 01:40:00 EDT 2009
Wed May 20 01:45:00 EDT 2009
Wed May 20 01:50:00 EDT 2009

Time	Actual	Forecast	Difference
Wed May 20 00:45:00 EDT 2009	10241.9	10235.4	6.4
Wed May 20 00:50:00 EDT 2009	10189.6	10193.1	3.4
Wed May 20 00:55:00 EDT 2009	10138.5	10166.8	28.3
Wed May 20 01:00:00 EDT 2009	10098.7	10138.0	39.2
Wed May 20 01:05:00 EDT 2009	10054.6	10089.0	34.3
Wed May 20 01:10:00 EDT 2009	10004.0	10061.1	57.1
Wed May 20 01:15:00 EDT 2009	9953.0	10031.4	78.4
Wed May 20 01:20:00 EDT 2009	9923.4	10007.7	84.3
Wed May 20 01:25:00 EDT 2009	9898.4	9976.2	77.8
Wed May 20 01:30:00 EDT 2009	9868.5	9951.6	83.1
Wed May 20 01:35:00 EDT 2009	9837.4	9931.2	93.8
Wed May 20 01:40:00 EDT 2009	9807.8	9903.0	95.2

```
Aggregate 5m point = 10241.866666666667 @ Wed May 20 00:45:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10189.626666666667 @ Wed May 20 00:50:00 EDT 2009 from 75 4s points.
refined (micro filtering) 4s data point @ Wed May 20 00:50:44 EDT 2009 from 10089.0 to 10142.58
refined (micro filtering) 4s data point @ Wed May 20 00:50:48 EDT 2009 from 10086.0 to 10142.18
Aggregate 5m point = 10138.5168 @ Wed May 20 00:55:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10098.706666666667 @ Wed May 20 01:00:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10054.626666666667 @ Wed May 20 01:05:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 10004.013333333334 @ Wed May 20 01:10:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9953.013333333334 @ Wed May 20 01:15:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9923.4 @ Wed May 20 01:20:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9898.44 @ Wed May 20 01:25:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9868.546666666667 @ Wed May 20 01:30:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9837.373333333333 @ Wed May 20 01:35:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9807.786666666667 @ Wed May 20 01:40:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9786.333333333334 @ Wed May 20 01:45:00 EDT 2009 from 75 4s points.
Aggregate 5m point = 9768.346666666666 @ Wed May 20 01:50:00 EDT 2009 from 75 4s points.
refined (micro filtering) 4s data point @ Wed May 20 01:54:20 EDT 2009 from 9579.0 to 9701.96
```

Minutes Ahead	Mean Error	StDev
5	22.0	20.5
10	22.2	22.5
15	34.0	19.8
20	31.9	22.6
25	37.8	29.8
30	48.8	25.5
35	51.8	26.6
40	53.6	31.9
45	51.9	42.0
50	52.7	39.9
55	58.8	43.6
60	66.9	60.5

Figure 12 The GUI also includes a forecast browser to allow the examination of individual forecasts during the test.

12 Known Issues

The test results show that the VSTLF system is generally working as expected. A close analysis of system behavior during tests has shown that the worst forecasts produced by the engine coincide with points at which the input load signal becomes perfectly flat for an extended period of time. It is unlikely that the real load was stationary during these portions of history, so it may be that the problem lies with the supply of test data, and not with the forecast system. However, the source of these flat sections is unknown.

13 References

[1] K. Xie, F. Wang, E. K. Yu, "Very Short-Term Load Forecasting by Kalman Filter Algorithm," Proceedings of the CSEE, vol. 16, no. 4, July, 1996, pp. 245-249.

[2] D. J. Trudnowski, W. L. Mcreynolds, "Real-Time Very Short-Term Load Prediction for Power-System Automatic Generation Control," IEEE Transaction on Power Systems, vol. 9, no. 2, March 2001, pp. 254-260.

[3] K. Liu, S. Subbarayan, R. R. Shoults, M. T. Manry, C. Kwan, F. L. Lewis and J. Naccarino, "Comparison of Very Short-Term Load Forecasting Techniques," IEEE Transaction on Power Systems, vol. 11, no. 2, May, 1996, pp. 877-882.

[4] W. Charytoniuk, M. S. Chen, "Very Short-Term Load Forecasting Using Artificial Neural Networks," IEEE Transaction on Power Systems, vol. 15, no. 1, Feb., 2000, pp. 263-268.

[5] P. Shamsollahi, K. W. Cheung, Q. Chen, E. H. Germain, "A Neural Network Based Very Short Term Load Forecaster For the Interim ISO New England electricity Market System," IEEE Power Industry Computer Applications Conference, 2001, pp. 217-222