High Performance Commercial Building Systems

Software Toolbox for Component-Level Model-Based Fault Detection and Diagnosis Methods

April 7, 2004

Peng Xu, Moosung Kim and Philip Haves

Lawrence Berkeley National Laboratory

Subtask 2.3.3 Develop semi-automated, component-level diagnostic procedures
Element 5 – Integrated Commissioning & Diagnostics

Table of Contents

1 Abstract

This document describes the contents of a toolbox for component-level model-based fault detection methods in commercial building HVAC systems. The toolbox consists of five basic modules: a parameter estimator for model calibration, a preprocessor, an AHU model simulator, a steady-state detector, and a comparator. Each of these modules and the fuzzy logic rules for fault diagnosis are described in detail. The toolbox is written in C++ and also invokes the SPARK simulation program.

2 Introduction

The aim of the work described here is to develop and implement model-based fault detection methods for HVAC components and subsystems. The software routines documented here are designed to be used with a set of component models implemented in the object-oriented simulation program SPARK, developed at LBNL. The models are documented in a companion report [library].

SPARK is used to execute the models at each stage of use of the tool. In the calibration stage, model parameters are identified on a component-by-component basis. In some cases, e.g. air handling units, the components are then aggregated to form subsystems whose boundaries are defined by the availability of reliable sensor measurements. In the fault detection stage, SPARK is used to simulate the subsystem model. The calibration routines support both linear and non-linear models. Currently, only static models are supported. The execution of the SPARK models is illustrated in Appendix I.

Model-based fault detection tools proposed here can be used in both commissioning and routine operation. As it is shown in Figure 1, each phase involves a distinct mode of execution and involves two different stages, which are performed sequentially – model calibration followed by fault detection. During the commissioning phase, design information and manufacturer's performance data are first used to calibrate the model. Functional test data taken during start-up tests are then used to detect any pre-existing faults. Once remedial work has been performed, the tests are repeated to confirm that the faults have been corrected. Once the test results are considered satisfactory, the test data are used to recalibrate the model for use in the on-line monitoring phase. The predictions of the recalibrated model are compared with routine operating data, in order to detect any faults that may arise during subsequent operation.

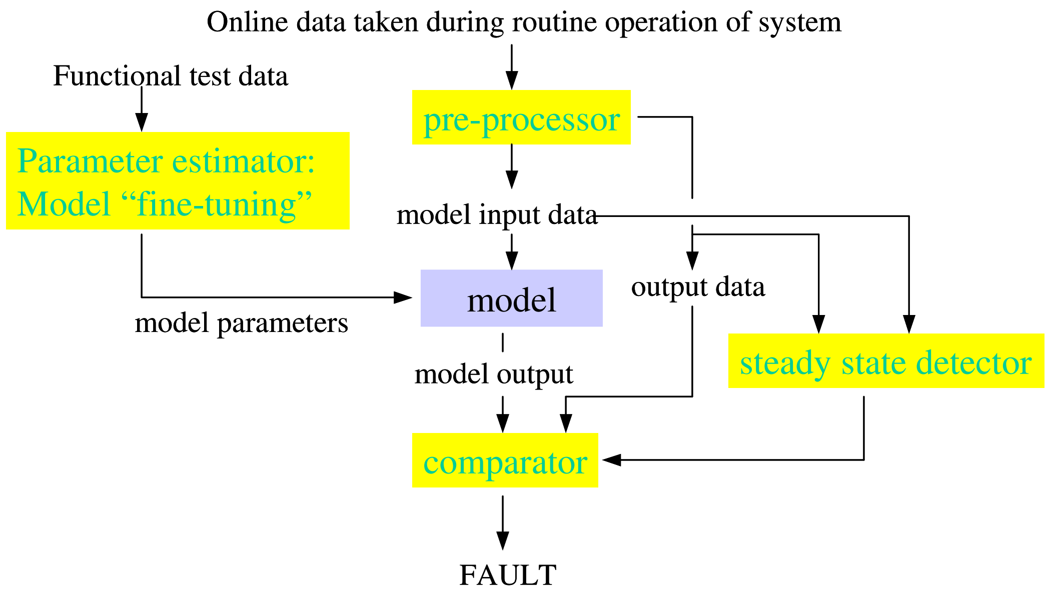


Figure 1 Fault detection schema

3 Online monitoring and fault detection modules

3.1 Pre-processor

On the top of Figure 1, the pre-processor module of the fault detection stage collects the online data taken during routine operation of the system. This program converts the raw data file, containing all data measured during the fault detection phase into processed data files having a standard format. There is a control file, which contains the standard format and naming conventions of the parameters to be converted. This control file is used to process the raw data file, convert it and separate it into two processed data files: one is the measured input parameter data file, and other is the measured output parameter data file. Data is also checked for errors prior to writing the input and output data files.

Several different modules use the input and output data files within the fault detection stage. One of these, the model simulation itself, has already been discussed in detail. Simulation of the AHU model in SPARK requires the use of model input parameter data obtained from the preprocessor module. This model input data, as well as the output data found in the files are used by the steady state detector module.

3.2 Steady-state detector

The steady-state detector module is to detect whether the system components under consideration are in under steady-state. The governing equations used to model all components of the system are static, since no dynamics have been modeled. Hence we can only consider fault detection during steady-state, and no transients are allowed. As such, there must be some threshold for determining the steady-state condition for both input and output data sets. Steady-state detection is performed by computing the EWMA (Exponentially-Weighted Moving Average) of the difference between consecutive time-indexed values for input and output parameter data values. If the absolute value of this computation for any parameter value is below a predetermined threshold value, then steady-state has been achieved.

The input and output data files are both needed by the steady-state detector module, and come from the preprocessing module. The control file required by the steady-state detector module contains the threshold values of each input and output parameters. The output file, stores the results of the detection. For each time-indexed data point, a value of 1 is recorded in this file if the component is found to be at steady-state, or 0 if it is not at steady-state.

3.3 Simulator

The C++ program file that implements the simulator module invokes the aggregate AHU (Air-handling unit) model generated by SPARK. All of the components of the AHU model's parameters are calibrated by the functional test data in a separate phase. These calibrated constant parameters are hardcoded. The preprocessor module automatically generates the measured operating AHU system input data, and the results are effectively written in the file pos.inp.

Both of these input files, pos.inp and constant.inp are referenced in the file AHU1.run, which is used by SPARK. Currently all of these calibrated parameters are hard-coded, not automatically written into the constant.inp file. It is possible that additional coding effort could be used to automate this process, making for a more seamless boundary between the two stages of model parameter calibration and fault detection.

There are over 100 parameters that SPARK requires for aggregate AHU system simulation. However, there are only about 10 that are actually calibrated. There are several software development options that exist and may need to be explored in terms of how to handle generalizing this parameter selection, calibration and component inclusion modeling issue. The AHU1.run file also references an output file, Sparkoutput.out. This is where the resulting simulated output data for the AHU model executed by SPARK will be stored. Other meta-data is also included in the AHU1.run file, such as the initial and final simulation times, as well as the simulation time increment, etc. There is also an AHU.prf file that is used by SPARK in order to set the specific parameters that control the methods and tolerances for analysis and solution of the governing equations. The executable file, AHU.exe, will generate the simulated output results of an overall model of the AHU system. Diagnostic information on SPARK's simulation is returned to the console upon completion of the simulation by default. This can be suppressed by redirecting this information to a log file, instead of to console I/O. This is named SPARK.log.

3.4 Comparator

This is implemented by the C++ program file comparator.cpp. The system must first be in steady-state, determined by data from the file SS.dat. Then the simulated outputs derived from the SPARK simulation module, Sparkoutput.dat, are compared with the real output of the system, realoutput.dat. The real output data comes from the preprocessor module. Depending on the magnitude of the absolute error between the two, a determination is made regarding whether or not a fault has occurred. The results are stored in the file named FaultReport.dat, and released to console I/O screen display.

Detection of a fault is based upon comparing the magnitude of the absolute error to some component-specific threshold. The thresholds are based upon empirical input parameter data found in the file realinput.dat, and characterize specific physical parameters of the components, in a least squares sense. These relationships are defined in a separate file, threshold.cpp, and called in the comparator module. Hence, the information in the file realinput.dat needs to be used by the comparator module. Although the least squares method is currently used to identify most of these thresholds empirically, in the case of the mixing box model these thresholds will be based upon the upper & lower limits defined by the SPARK model (Xu & Haves, 2001). Specifically, the outside air fraction (OAF) provides the basis for setting these thresholds in lieu of the mixed air temperature. These thresholds are defined over the entire operating range, and therefore can similarly be used for fault detection over the same range.

Analysis of the effects of measurement & modeling errors are not explicitly accounted for in this fault detection toolbox. It is assumed that some engineering judgment will need to be used to take these factors under consideration to prevent spurious false alarms and/or missed detections.

3.5 Calibration modules

There are several nonlinear optimization techniques that may be used for model calibration. The method implemented in the toolbox is Box's Complex Method for constrained nonlinear optimization (Box, 1965). A step-by-step description of the algorithm is in the appendix II.

The parameter estimator module uses functional test data to calibrate parameters for the specific component. The detailed the module structure working with current version of the SPARK is shown in Appendix V. The calibrations of the three main components within the AHU system are shown in Appendix VI, VII and VIII.

3.6 Fault Diagnosis

In the comparator module, fault detection is complemented with fault diagnosis functionality as well. Given that the AHU system is found to be in steady-state, fault diagnosis is performed by using the innovation (error) found in the detection phase as a fuzzy input to a diagnostic rulebase. The software used to generate the fuzzy rulebase, input sets and membership functions is TILShell 3.0, from Togai InfraLogic, Inc. C code can be automatically generated by this program, and the data structures from the resulting source code can be incorporated into the existing toolbox comparator module.

There are currently some problems with incorporating the automatically generated source code for use by the existing toolbox comparator module . However, the problem can be solved by manually implementing a coding workaround. As it turns out, some of the data structures that are used in the automatically generated code do not exist. Hence, to correct the problem, we can manually correct the problem by changing the data structures appropriately.

The fault diagnosis is based upon a list of possible faults that may occur over the entire operating range of the controlled input for a particular HVAC component. Because different faults occur in different sections of the operating range of the control signal, weighted bins are used to accumulate instances of detected faults together. Typically there are two or three bins weighted over sections of the operating range. These bins provide a way to partition the operating range into sections. In this way, the number of faults detected in any one bin can be used to set a threshold count for when an accurate computation of the moving average value can be made. Also, weighted bins can serve as a mitigating effect for the new incoming data in computation of the moving average, based upon the operating range location within the bin. This will influence the moving average filter, which uses a forgetting factor. The forgetting factor is used to temporally weight the incoming innovation data in order to allow for using a reasonably modest memory of previous inputs. The final result will be to provide moving average values of the innovation for each bin that will serve as fuzzy inputs into a rulebase for diagnosis.

The fault diagnosis of the three main component are shown in Appendix IX, X, and XI.

5 References

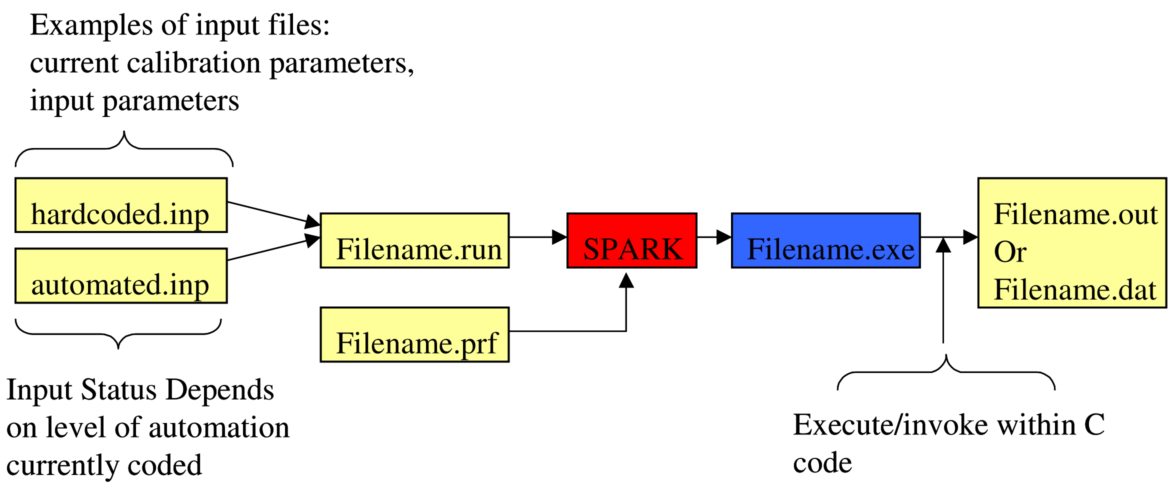
Box, M. J. (1965). A new method of constrained optimization and a comparison with other methods. *Computer Journal*, 8:42--52.

Peng Xu and Philip Haves. (2002). Field Testing of Component-Level Model-Based Fault Detection Methods for Mixing Boxes and VAV Fan Systems. *Proceedings of 2002 American Council ACEEE Summer Study on Energy Efficiency in Buildings.* Pacific Grove, CA. Accepted.

Peng Xu and Philip Haves. (2001). *Library of component reference models for fault detection (AHU and chiller).* Report to California Energy Commission. Berkeley, CA.: Lawrence Berkeley National Laboratory.

Appendix I Innovate SPARK models in the current version of toolbox

Three main components or subsystems within the air handling unit subsystem are the



mixing box, the fan-duct subsystem, and the cooling coil subsystem. To explore different implementation approaches, the simpler models are coded as C++ classes and SPARK is used for the more complex models, such as the cooling coil subsystem, which require iteration. Calling the current version of SPARK requires an EXEC call, which involves a significant overhead. When a more convenient form of SPARK becomes available, e.g. a DLL, SPARK will become the implementation method of choice for all components, providing a uniform way of calling and coupling models.

The basic file processing and data transfer requirements for generating a SPARK executable are shown in Figure 3. The input files are: 1) filename.run, a meta-file containing a list of the files referencing basic constant, input and calibration parameters required to solve the equation(s). It also contains other important parameters such as the name of the resulting output file, the initial & final simulation times, and time interval, etc., and 2) filename.prf, which contains the user preferences for how the governing equation(s) are to be solved and certain tolerances associated with obtaining the solutions. Given these two files, the SPARK software will create an executable file that can be invoked on the command line, and hence called from within a C++ program when necessary.

The files containing the basic constant, input and calibration parameters required to solve the governing equation(s) for the component(s) being simulated are typically created with extension *.inp. As shown in Figure 3, some *.inp files that are generated manually and others are generated automatically, either by C++ programs or from trend logs of EMCS databases. Table 1 illustrates the present status of automation vs. manual generation for the different uses of SPARK within the toolbox:

1. Total AHU model simulation for fault detection

2. Cooling coil subsystem model simulation for calibration

Table 1 Current status of hardcoded and automated use of SPARK

	Automated		**Manually Generated**
1)	Pos.inp (Input parameters) - 480 lines of data - should be written by preprocessor & read by SPARK		Constant.inp (Calibration parameters) - 1 line of data - hardcoded & read by SPARK
2)	ccvalveSim1.inp (Calibration parameters) -1 line of data - Continuously written by calibration routine & read by SPARK	ccvalvexReal1.inp (Input parameters) - 8 lines of data - Written once by calibration routine using data from Cccalibrationsansome.dat & read by SPARK	ccvalveCConstant1.inp (Constant parameters) -1 line of data - hardcoded & read by SPARK

Note that automation of the input parameter data file for use in the total AHU SPARK model simulation for fault detection has not yet been implemented. Furthermore, the naming convention for the files shown is provisional. In the future, the file naming convention for these input (and other) relevant files would be standardized.

Appendix II Complex method used for calibrations

Let:

n_c = number of parameters to be estimated

M = number of vertices (points) corresponding to the number of initial guesses for all calibration parameters values, representing a complex geometric figure in n_c{?}-space (the 'complex').

1. Find $M \geq n_c+1$ points that satisfy all implicit & explicit inequality constraints. There are two cases, depending on the nature of the constraints:
   a. For explicit inequality constraints, the complex is bounded by a hypercube defined by the constraints, which defines the region of feasible solutions. One point can be picked within the feasible range, and for the remaining points, a uniform random number generator can be used to ensure compliance with the bounding hypercube restriction.
   b. For implicit inequality constraints, each new test point is generated randomly and then tested for feasibility. If the point is infeasible, it is moved to a location halfway to the centroid of the already accepted points. This process is repeated until a set of *k{M}* feasible points is found.

Note: only explicit constraints are used in the current version of the toolbox.

2. Define an objective function to minimize.

In this case, the objective function is an error function, *f,* defined as the sum of the scaled absolute error between the measured actual output of the component or subsystem and model simulated output, averaged over a set of N measurements:

$$f_j = \frac{1}{N}\sum_{i=1}^{N}\left|y_i^{sim}(\mathbf{x}_i,\mathbf{c}_j) - y_i^{real}\right|, \qquad \forall j \in \{1 \ldots M\}$$

where f_j is the error for the *j*th point in the complex, y_i^{sim} is the model output for the *i*th measurement, $\mathbf{x}_i$ is the vector of measured inputs the *i*th measurement, $\mathbf{c}_j$ is the vector of parameters that define the *j*th point in the complex and y_i^{real} is the measured output for the *i*th measurement. There are M error functions to compute, one for each point in the complex.
The function is evaluated at each of the k points (vertices). The point that evaluates to the largest function value is reflected about the centroid of the remaining vertices of complex. The formula for reflection is:

$X_r = (1+a)X_0 - aX_h$

where X_r = new reflected point

X_0 = centroid of remaining points

X_h = current point evaluating to largest function value

a = positive-valued reflection coefficient (default value 1.3)

Note that this formula must be computed for each dimension of the parameter space. The intuition for the reflection formula stems from the fact that the new reflected point lies on a line joining the centroid of the remaining points and the high point. However, it lies closer to the centroid than to the high point. This is what we want, since we move away from the high point, and closer to potential low points. The distance away from the high point and to the centroid of remaining points is governed by selection of the reflection coefficient, a.

3. We must test for feasibility of the new reflected point (meaning the constraints must not be violated with this new reflected point).
   a. If the new point is feasible, and the function value is lower than the high value, we replace this high value with the new reflected point and proceed to finding another maximum function value and reflection in Step 2.
   b. If the function value is not lower, then the reflection coefficient must be too high. Hence, a new reflection point is found by halving the value of a, and re-computing the reflection point. This iteration continues until a point is found that results in a lower function value. However, we don't want this to go on indefinitely, so we must set some tolerance, ε_1, for a, at which this process will stop.
   c. If a lower function value is not found after reaching the ε_1 tolerance level, then we just throw away this point, and start Step 2 all over again with the 2nd highest function value.

4. If at any stage the reflected point is deemed to be infeasible, it is moved halfway in towards the centroid until it becomes feasible.

5. Convergence of the process (i.e. termination of the algorithm) is determined when the following conditions are met:
   a. The complex shrinks to a specified small size, i.e. the distance between any two vertices is smaller than some pre-specified tolerance, $\varepsilon_{2.}$
   b. The standard deviation of the function value becomes sufficiently small, below some prescribed tolerance, ε_3.

Appendix III Table of C++ files

File Name	Usage	Module	Content
PreProcessor.cpp	Online monitoring/fault detection	Preprocessor	Pre process the raw data collected from EMCS systems
SSDectector.cpp	Online monitoring/fault detection	Steady state detector	Determine whether the system is under steady state
simulation.cpp	Online monitoring/fault detection	Simulator	Conduct SPARK simulation
Comparator.cpp	Online monitoring/fault detection	Comparator	Compare the measured and simulated output and determine whether there is a fault
Threshold.cpp	Online monitoring/fault detection	Comparator	Determine the threshold of the output variable for comparator
coolingcoilvalvecalibration.cpp	Automatic calibration	Cooling coil and valve calibration	Cooling coil and valve system calibration
fancalibration.cpp	Automatic calibration	VAV fan calibration	VAV fan system calibration
mixingboxcalibration.cpp	Automatic calibration	Mixing box calibration	Mixing box calibration
bin.cpp	Automatic calibration	Bin module	Collapse a big data set from functional step test to a small training data set
plot.cpp	Automatic calibration	Calibration	Real time plot of all above calibrations

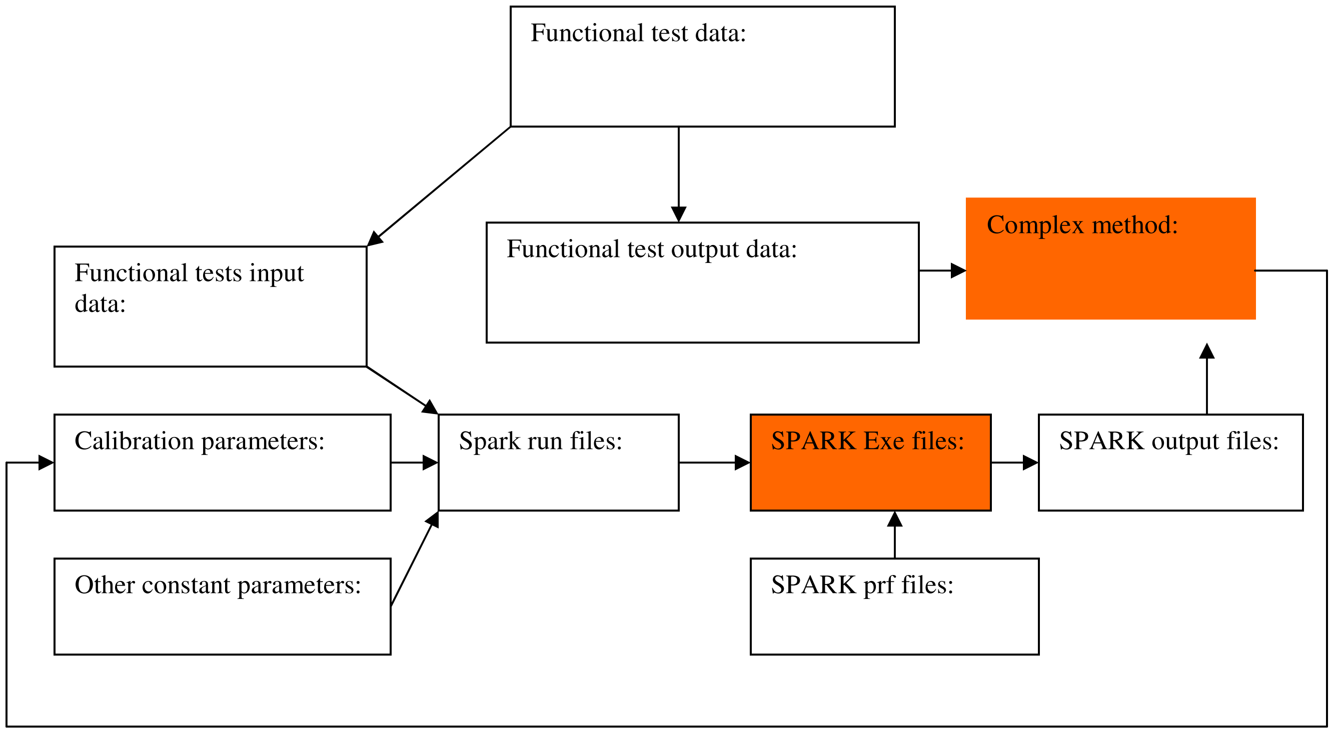
Appendix IV Table of data files (not C++ program files)

File Name	Called By/ Created By	Module	Content	Format
raw.dat	preprocessor.cpp/ EMCS system	Preprocessor	Time-indexed data dump from building EMCS system (1 11-hr working day, 10 sec intervals)	Time Parameter names from EMCS system
convert.con	preprocessor.cpp/ software user	Preprocessor	Control file that contains the standard format and naming conventions of the parameters to be converted	N/A
realinput.dat	SSDetector.cpp, comparator.cpp, threshold.cpp, AHU1.run SPARK /preprocessor.cpp	Steady-State Detector, Comparator, SPARK Model	Contains the input data read during routine operation of system, after being processed by preprocessor module	Time TLiqEntCC posValveCC TLiqEntHC posValveHC TAirRet wAirRet TAirOut wAirOut posDamper
realoutput.dat	SSDetector.cpp, comparator.cpp, threshold.cpp /preprocessor.cpp	Steady-State Detector, Comparator	Contains the output data read during routine operation of system, after being processed by preprocessor module	Time TAirLvgCC wAirLvgCC TLiqLvgCC mLiqCC TAirLvgHC TLiqLvgHC mLiqHC TAirSup wAirSup mAirSup powerTotSfan nSfan pSfan mAirRet powerTotRfan nRfan pRfan
SS_T_input.con	SSDetector.cpp/ Software user	Steady-State Detector	Control file required by the steady-state detector module containing the threshold values of each input parameter	TLiqEntCC posValveCC TLiqEntHC posValveHC TAirRet wAirRet TAirOut wAirOut posDamper
SS_T_output.con	SSDetector.cpp/ Software user	Steady-State Detector	Control file required by the steady-state detector module containing the threshold values of each output parameter	TLiqEntCC posValveCC TLiqEntHC posValveHC TAirRet wAirRet TAirOut wAirOut PosDamper
SS.dat	comparator.cpp/ SSDetector.cpp	Comparator	Stores the results of the detection as follows: for each time-indexed data point, record a value of 1 (Yes, at steady-state i.e. below threshold) or 0 (No, not at steady-state i.e. above threshold)	Time SS Value
FaultReport.dat	Software User/ comparator.cpp	Comparator	The final results are stored in the file	N/A

Pos.inp	AHU1.run, SPARK /preprocessor.cpp	SPARK Model	Contains input data from fault detection	Index MixposDamper MixTAirOut RetTAirRet
Constant.inp	AHU1.run, SPARK / Software User	SPARK Model	Contains calibration parameter values	See SPARK documentation for more details
Sparkoutput.out	comparator.cpp/ SPARK	SPARK Model	Contains SPARK simulation output results for aggregate AHU system	See SPARK documentation for more details
mixingcal.dat	mixingboxcalibration.cpp/ EMCS system	Parameter Estimator	Contains training data for calibration of mixing box model parameters	MixTAirOut RetTAirRet MixposDamper TAirEntCC
fanCal.dat	fancalibration.cpp/ EMCS system	Parameter Estimator	Contains training data for calibration of VAV fan-duct model parameters	nFan mAir PowerTot pFan Pstat
Cccalibrationsansome.dat	coolingcoilvalvecalibration .cpp/EMCS system	Parameter Estimator	Contains training data for calibration of coil-valve model parameters	TAirEnt wAirEnt TLiqEnt mAir mLiqOpen pos wAirLvg TairLvg
ccvalvexReal1.inp	ccvalve1.run, SPARK/ coolingcoilvalvecalibration .cpp, EMCS system	Parameter Estimator	Contains input training data for generation of simulated output for coil-valve model using SPARK (One-time read)	Index TAirEnt wAirEnt TLiqEnt mAir mLiqOpen pos
ccvalvecSim1.inp	ccvalve1.run, SPARK/ coolingcoilvalvecalibration .cpp	Parameter Estimator	Contains current calibration parameters for generation of simulated output for coil-valve model using SPARK (Continuously read)	Index C_{air} C_{water} Val_{c1} Val_{c2}
ccvalveCConstant1.inp	ccvalve1.run, SPARK/ Software User	Parameter Estimator	Contains constant parameters for generation of simulated output for coil-valve model using SPARK	A_{ext} A_{int} P_{atm} $Val_{Leakpar}$ Authority
ccvalveSparkoutput1.dat	coolingcoilvalvecalibration .cpp/SPARK	Parameter Estimator	Contains SPARK simulation output results for coil-valve model (Continuously written)	

Appendix V Current calibration module structure

The diagram illustrates the current version of the calibration process for all of the component model parameters.



The performance data used to calibrate the model parameters shown at the top of Fig. 2 is separated into two files at the beginning. One is the output data file contains information such as leaving air and water temperatures. The other is input data file contains valve position data. The file containing the constant parameters and the automated files are also shown in Fig. 2. These files contain the input and calibration parameters, respectively. Notice that the files in SPARK calls at the bottom of Fig. 2 correspond to the invocation of SPARK (Appendix I) - simulating the cooling coil-valve model.

The calibration parameter file is written continuously as required by the algorithm that is used to calibrate the model. After SPARK simulates the output during execution of the C++ program, the resulting data will be stored in the output file called. This information is written, and also read continuously by the same program, as needed by the algorithm to calibrate the model in a pseudo-information feedback loop. Once the algorithm is complete, the coil-valve model's calibrated parameters will appear on a visual display.

The resulting data presented in the visual display does not necessarily have to be coincident with the training data sets used to calibrate the models. In fact, there is a bin module, which is used to reduce the size of the training data set. This reduced sized training data set is used to calibrate the model, while the original larger data set is used to visually display the results of the calibration on the plot. This “bin” tool collapses one big data set into a small data set based on two criteria: that the system is in steady state, and the new control signal is significantly different than the previous control signal recorded in the small data set. The small data set can then be used for model calibration, by grasping the core characteristics of the larger data set for analysis.

The Appendix IV, V, VI highlight the idiosyncrasies of the code required to calibrate each one of the components using the aforementioned algorithm. Very little emphasis will be placed on describing the code required to update the visual display that shows the real and simulated data points as the calibration progresses. However, the output is very visually appealing because the fit of the calibrated model parameters to the actual data can be seen to converge in “real-time.” At most, a screenshot of the resultant fit will be provided, following a description of the code required to calibrate each component’s nuances in implementing the algorithm.

Appendix VI Mixing box calibration module

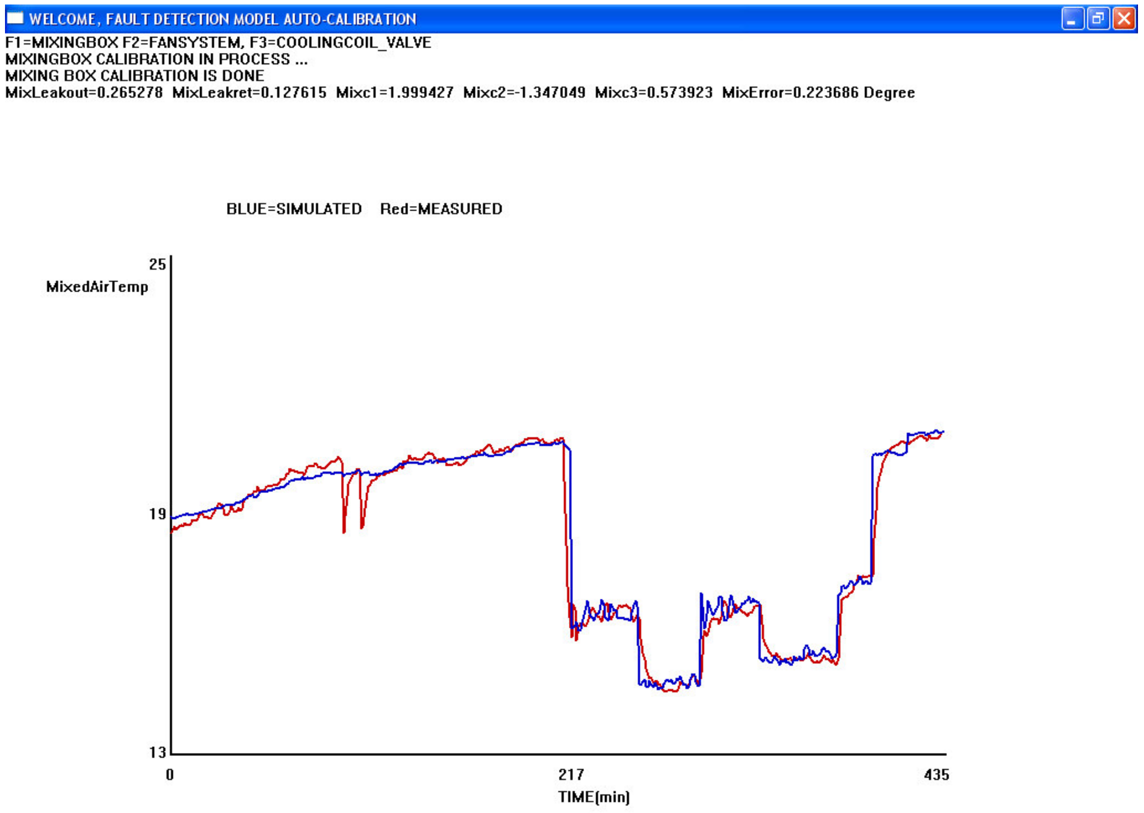
For the mixing box, the parameters to be calibrated are: $Leak_{Ret}$, $Leak_{Out}$, c_1, and c_2 (c_3 = $1-c_1-c_2$). The inputs are: pos, T_{ret}, and T_{out}. The output parameter is T_{mix}, and the governing relationship is as follows:

$$T_{mix}=(((1-Leak_{Ret})-Leak_{Out})(c_1*pos+c_2*pos^2+c_3*pos^3)+Leak_{Out})*(T_{out}-T_{ret})+T_{ret}$$

More details of this model are given in (Xu & Haves, 2001).

This module performs model calibration of the mixing box component of the AHU system. The parameters, inputs, outputs, and their relationships are listed above. Because the parameters are nonlinear with respect to the output parameter, the least squares method cannot be used to compute them. The least squares method can only be used when the output parameter is linearly dependent upon the calibration parameters.

The calibrated model parameters can easily be found by applying this technique. This technique is the Box-Complex constrained nonlinear optimization algorithm discussed in Appendix. The picture in Figure 3 illustrates the fit of the simulated model output compared to the real measured output mixed air temperature.



```
WELCOME, FAULT DETECTION MODEL AUTO-CALIBRATION
F1=MIXINGBOX F2=FANSYSTEM, F3=COOLINGCOIL_VALVE
MIXINGBOX CALIBRATION IN PROCESS ...
MIXING BOX CALIBRATION IS DONE
MixLeakout=0.265278  MixLeakret=0.127615  Mixc1=1.999427  Mixc2=-1.347049  Mixc3=0.573923  MixError=0.223686 Degree
```

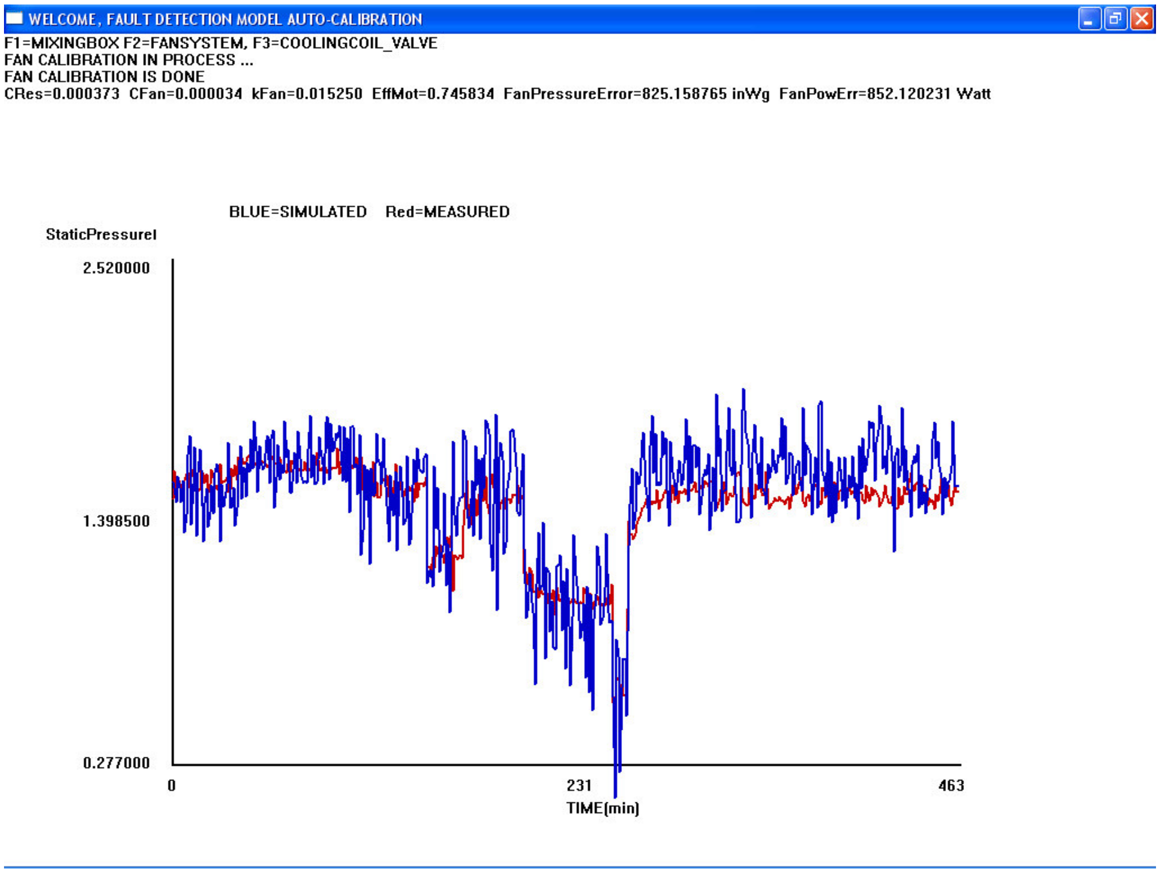
Appendix VII VAV Fan-duct system calibration module

For the fan-duct system, the parameters to be calibrated are: C_{Res}, C_{Fan}, k_{Fan}, and $\eta_{Mot.}$ The constants are: Mot_{Frac}, $\eta_{ShaftMax}$, P_{Atm}, Area, c_{Eff}, and density(ρ). The input parameters are: n_{Fan} and $m_{Air.}$ The output parameters are: $Power_{Tot}$, P_{fan}, P_{Stat}. The governing relationships are:

$$P_{fan}=k_{fan}*n_{Fan}^2-C_{Fan}*m_{Air}^2$$
$$P_{Stat}=P_{fan}-c_{Res}*m_{Air}^2$$
$$Power_{Tot} = m_{Air}*P_{fan}/(\rho * \eta_{Mot})$$

Again, for more details on this model refer to (Xu & Haves, 2001). Note that unlike the mixing box calibration, there are three calibrations that must take place because there are three output parameters and three governing equations. These calibrations must take place sequentially, and the result of the first calibration must be used in the second and third calibrations. Other than this major difference, the underlying principle of calibration is the same.

Note that the second governing equation to be calibrated is linear in c_{Res}. Hence, the least squares method could be used to obtain this value. However, for consistency, we'll use the same Box - Complex constrained nonlinear optimization algorithm as described previously for all three calibrations. As a result of applying the algorithm, we will obtain the calibrated model parameters: C_{Res}, C_{Fan}, k_{Fan}, η_{Mot}. The picture in Figure 4 illustrates the fit of the simulated model output compared to the second real measured output, static fan pressure.



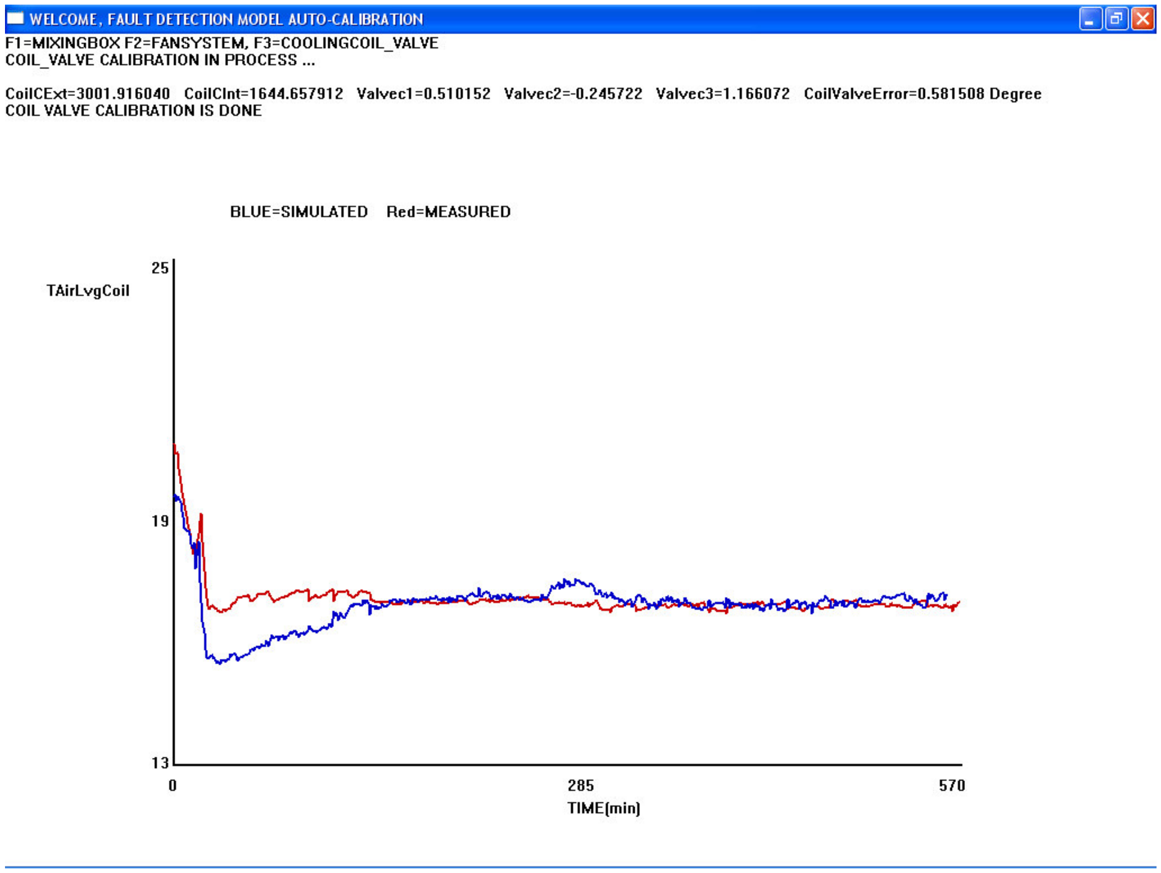
WELCOME, FAULT DETECTION MODEL AUTO-CALIBRATION

```
F1=MIXINGBOX F2=FANSYSTEM, F3=COOLINGCOIL_VALVE
FAN CALIBRATION IN PROCESS ...
FAN CALIBRATION IS DONE
CRes=0.000373  CFan=0.000034  kFan=0.015250  EffMot=0.745834  FanPressureError=825.158765 inWg  FanPowErr=852.120231 Watt
```

Appendix VIII Cooling coil and valve system calibration module

Finally, for the cooling coil-valve system calibration, the parameters to be calibrated are: C_{air}, C_{water}, Val_{c1}, and $Val_{c2.}$ The constants are: A_{ext}, A_{int}, P_{atm}, $Val_{Leakpar}$, and Authority. The input parameters are: T_{airEnt}, w_{AirEnt}, T_{liqEnt}, m_{Air}, $m_{LiqOpen}$, and valve position. The output parameters are: T_{airLvg}, w_{AirLvg}, and T_{liqlvg}, although the latter two parameters have been left out at this stage.

The governing relationships are a bit complicated to express here. The details are provided in (Xu & Haves, 2001). There are ideally 3 calibrations to perform, similar to the fan system calibration. The calibrations must take place sequentially. The underlying principle of calibration is the same. As a result of applying this algorithm, the calibrated model parameters: C_{air}, C_{water}, Val_{c1}, and Val_{c2} will be obtained. The picture in Figure 5 illustrates the fit of the simulated model output compared to the real measured output, air temperature downstream of the cooling coil.



```
WELCOME, FAULT DETECTION MODEL AUTO-CALIBRATION
F1=MIXINGBOX F2=FANSYSTEM, F3=COOLINGCOIL_VALVE
COIL_VALVE CALIBRATION IN PROCESS ...

CoilCExt=3001.916040   CoilCInt=1644.657912   Valvec1=0.510152   Valvec2=-0.245722   Valvec3=1.166072   CoilValveError=0.581508 Degree
COIL VALVE CALIBRATION IS DONE
```

Appendix IX Cooling and Heating coil fault diagnosis

The cooling & heating coil fault diagnosis is based upon the premise that there are two fuzzy inputs: innovation at full duty and innovation at minimal duty of the heating or cooling coil valve. Two bins are used to aggregate the data for each of the fuzzy inputs. There are four possible faults: valve leakage, coil fouling, and positive or negative sensor offset. The following table summarizes a fuzzy rulebase, based upon combinations of all possible faults & bins:

Fault Type	Innovation at Minimal Duty (ε_0)	Innovation at Full Duty (ε_{100})
Valve Leakage	+	0
Coil Fouling	0	-
Positive Sensor Offset	+	+
Negative Sensor Offset	-	-

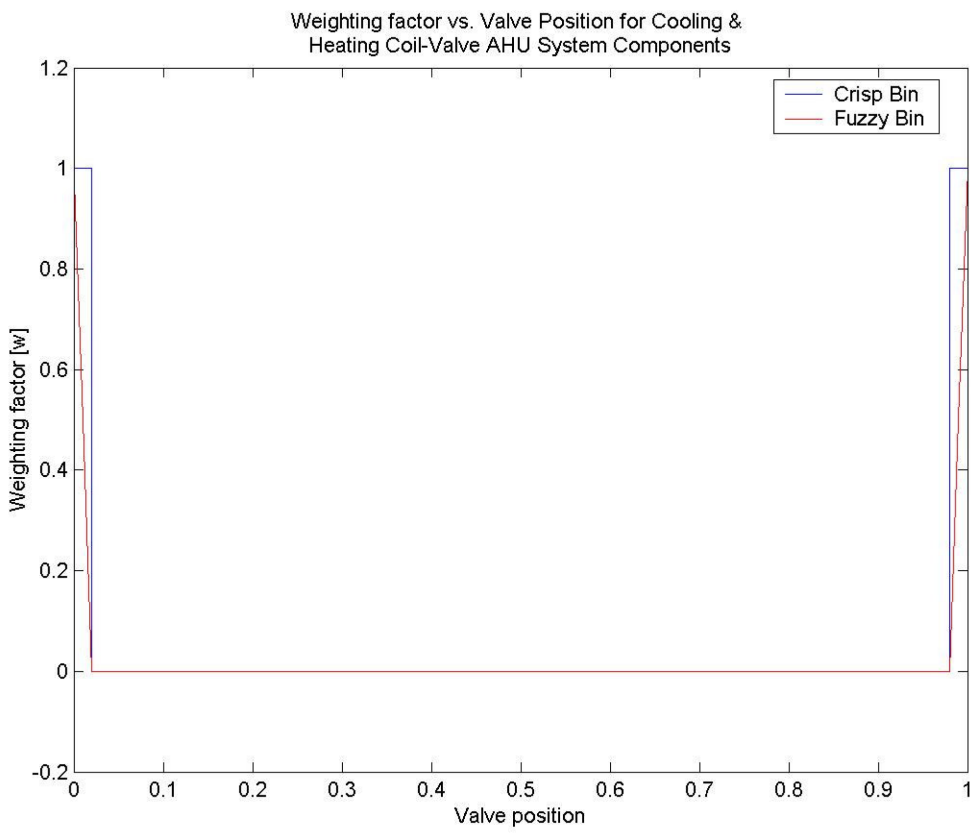
The weighting factor for both inputs are binned according to the graph below illustrating both the crisp and fuzzy bins for the heating & cooling coil-valve weighting factor. As shown in following figure, the weighting factor is meant to act as a mitigating technique in computation of the moving average when using the fuzzy bin method. The reason for using this technique is to weight the new incoming data according to the operating range location. The result that this might have on the final updated moving average value can be either positive or negative, depending on the value of the difference between the new incoming data and the current moving average value. The mitigating effect is evident by the weighting factor's linear decrease from 1 to 0. It can also be seen that the weighting factor is always equal to one for the crisp bin method. Hence the following equation for the moving average is:

$$y_n = y_{n-1} w(1-\alpha) + [1 - w(1-\alpha)] x_n$$

and when w=1 it becomes:

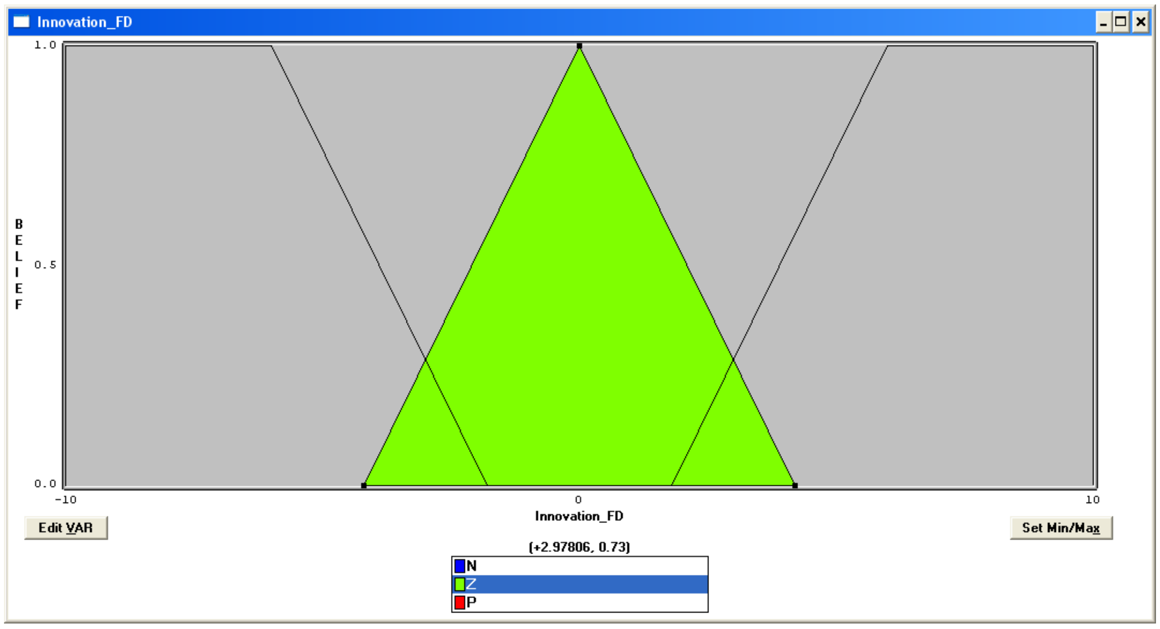
$$y_n = y_{n-1}(1-\alpha) + \alpha\, x_n$$

where α is the forgetting factor, x_n is the new data, y_{n-1} is the current moving average value, and y_n is the new moving average value.



Note that the moving average computation is not valid before 50 detections have been counted. However, it is still computed regardless of the number of counts, and displayed upon each detection, along with a warning stating that the value may not be valid. To clarify, the detection count will always be computed, as well the moving average. However, the diagnostic code is invoked only if the count for both bins exceeds 50. The resulting diagnostic information is displayed only if there is a detection (threshold exceedance) that triggers it, regardless if the count for both bins exceeds 50 or not. The bins shown above are only defined on the very small regions between 0 and 0.02, representing minimal duty, and between 0.98 and 1, representing full duty.

The fuzzy membership functions defined for the input variables are created in the TILShell 3.0 software. For both minimal and full duty innovations, the membership functions for the sets are as in figure:



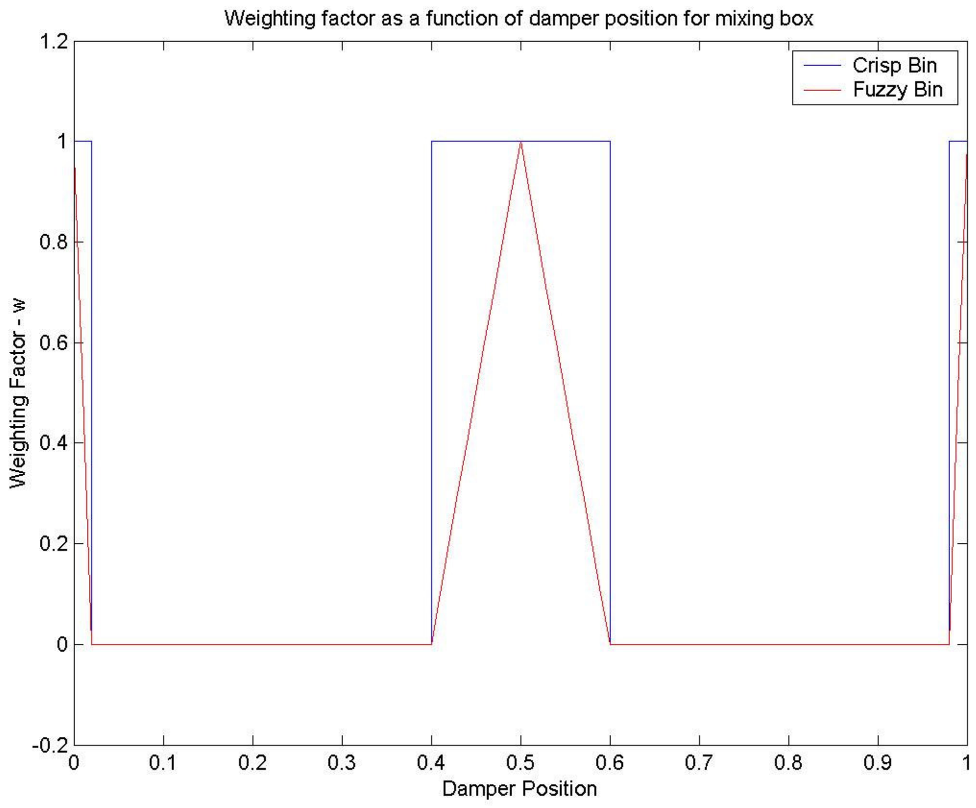
The rule-based associated with the fuzzy system is based upon the table shown previously. There are three basic fuzzy sets for both inputs: zero, positive and negative, which is evident from the diagram above and the values listed in the table. Each of the rules will fire based upon the computed inputs and defined fuzzy membership functions. As a result, the belief values associated with each of the rules will be used as the diagnostic information that is displayed upon an instance of triggered fault detection.

Appendix X Mixing box fault diagnosis

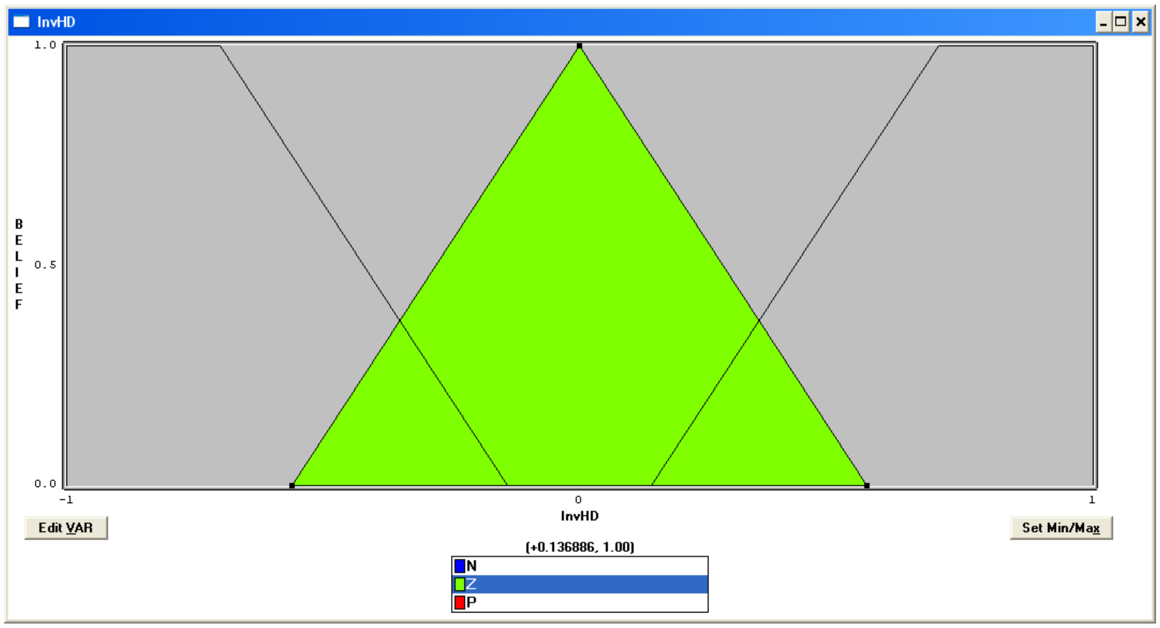
The mixing box fault diagnosis is based upon the premise that there are three independent fuzzy systems, one for each type of possible fault. There are three possible faults: outside air damper leakage, return air damper leakage, and all other types of faults lumped together in one category including nonlinearities. Each fuzzy system has its own rulebase, with innovations binned over specific sections of the operating range that act as inputs. They are as follows: innovation at minimal duty, innovation at half duty, and innovation at full duty of the mixing box damper position. The inputs correspond respectively to the faults mentioned previously. The following table summarizes the fuzzy rulebases, based upon combinations of all possible faults & bins:

Fault Type	Innovation at Minimal Duty (ε_0)	Innovation at Half Duty (ε_{50})	Innovation at Full Duty (ε_{100})
Outside Air Damper Leakage	-		
Return Air Damper Leakage			+
Nonlinearity and other faults		+	
Nonlinearity and other faults		-	

The weighting factor for both inputs are binned according to the Figure 9 below illustrating both crisp and fuzzy bins for the mixing box weighting factor.



The bins shown above are defined on the regions between 0 and 0.02, representing minimal duty, between 0.4 and 0.6 representing half duty, and between 0.98 and 1, representing full duty. The three basic fuzzy sets for all inputs are: zero, positive and negative. For all three innovation inputs, the membership functions for the sets are as follows:



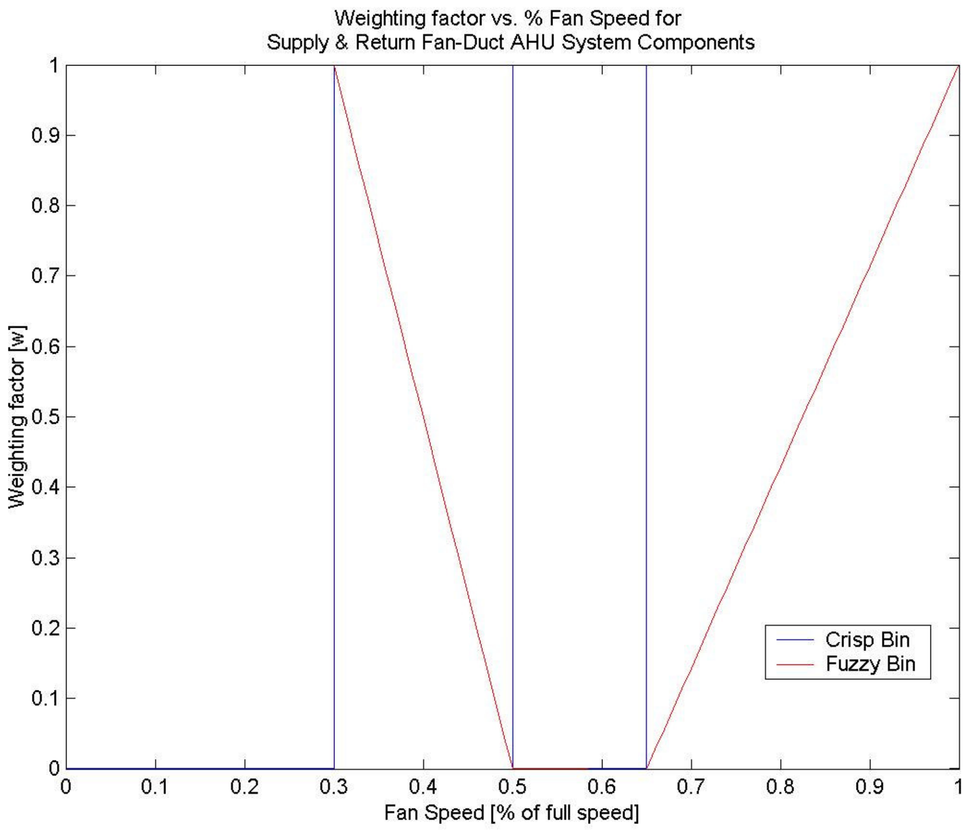
Again, each of the rules will fire based upon the computed inputs and defined fuzzy membership functions. The belief values associated with each of the rules will then be used as the diagnostic information displayed upon an instance of a triggered fault detection.

Appendix XI Supply and return fan diagnosis

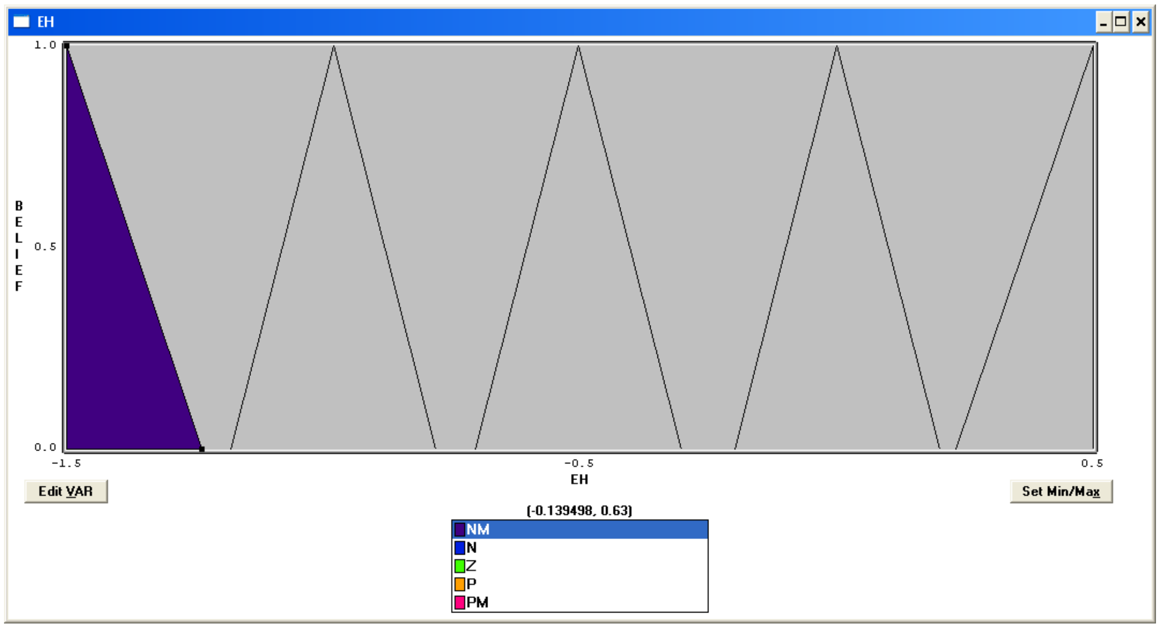
The supply & return fan duct-system fault diagnosis is based upon the premise that that there are two fuzzy inputs: innovation at low speed and innovation at high speed of the supply & return fans. Two bins are used to aggregate the data for each of the fuzzy inputs. There are eight possible faults: positive and negative sensor offset, fan stuck at full and intermediate speed, fan motor failure, fan reduced capacity, slipping fan belt, and all others lumped into a single generic category. The following table summarizes a fuzzy rulebase, based upon combinations of all possible faults & bins:

Fault Type	Innovation at Minimal Duty (ε_0)	Innovation at Full Duty (ε_{100})
Positive sensor offset	+	+
Negative sensor offset	S-	S-
Fan stuck at full speed	+	0
Fan stuck at intermediate speed	+	S-
Fan motor failure	L-	L-
Fan reduced capacity	S-	+
Slipping fan belt	M-	M-
Others	0	S-

The five basic fuzzy sets for both inputs shown in the above table are: positive (+), zero (0), small negative (S-), medium negative (M-), and large negative (L-). The weighting factor for both inputs are binned according to the graph below illustrating both the crisp and fuzzy bins for the supply & return fan duct system weighting factor in Figure 11.



The bins shown above are defined on the regions between 0.3 and 0.5, representing low fan speed, and between 0.65 and 1 representing high fan speed. The membership functions for the five basic fuzzy sets described earlier, for both innovation inputs, are as:



Note that the fuzzy sets shown in the legend of the illustration above (NM, N, Z, P, PM) do not correspond exactly with the sets described previously (positive (+), zero (0), small negative (S-), medium negative (M-), and large negative (L-)). The sets, however, have to be translated respectively because the software defaults to the former nomenclature. As always, each of the rules will fire based upon the computed inputs and defined fuzzy membership functions. The belief values associated with each of the rules will then be used as the diagnostic information displayed upon an instance of a triggered fault detection.