Korean J. Chem. Eng., **25**(5), 955-965 (2008)

Data reconciliation: Development of an object-oriented software tool

Ali Farzi*,†, Arjomand Mehrabani-Zeinabad*, and Ramin Bozorgmehry Boozarjomehry**

*Department of Chemical Engineering, Isfahan University of Technology, Isfahan, Iran, Postal Code: 84156-83111
**Department of Chemical Engineering and Petroleum, Sharif University of Technology, Tehran, Iran

(*Received 12 August 2007 • accepted 5 March 2008*)

Abstract−Object-oriented modeling methodology is used for encapsulating different methods and attributes of data reconciliation (DR) in classes. Classes which are defined for DR, cover steady-state, dynamic, linear and nonlinear DR problems. Two main classes are *Constraints* and *DR* and defined for manipulating constraints and general DR problem. The remaining classes are derived from these two classes. A class namely *DDRMethod* is developed for encapsulating all common attributes and methods needed for any DDR method. Developed DR software and the method of performing dynamic DR are discussed in this paper. Two illustrative examples of Extended Kalman Filtering and artificial neural networks are used for DDR and two classes of *DDRByKalman* and *NetDDRMethod* developed by inheritance from *DDRMethod* class for these two methods. Performance of the proposed method is investigated by DDR of temperature measurements of a distillation column.

Key words: Data Reconciliation, Object-Oriented Programming, Extended Kalman Filtering, Artificial Neural Network, Dynamic Simulation

INTRODUCTION

Measurements always contain some type of error and it is necessary to adjust their values to know objectively the operating state of the process. Two types of errors can be identified in plant data: random and gross errors. Random errors are small errors that occur due to the normal fluctuation of the process or random variations inherent in instrument operation. Gross errors are larger errors which occur due to incorrect calibration or malfunction of the instruments, process leaks, etc. Gross errors occur occasionally, that is, their number is small when compared to the total number of instruments in a chemical plant.

Data reconciliation (DR) is an optimization method for elimination of random errors from measured data in a processing system. It uses process model as constraint and statistical properties of measurements. DR can be performed in both steady-state and dynamic conditions. Many works have been done within the framework of linear and nonlinear steady-state DR (LSSDR and NSSDR) and linear and nonlinear dynamic DR (LDDR and NDDR), and some methods have been proposed [2-10].

Data reconciliation is commonly a complex problem for nonlinear and dynamic systems, specifically for large scale systems. The management of complexity of such problems is not easy and a framework is needed to encapsulate all common properties of DR problems. Object-oriented modeling is one of the important software development methods and has the benefits of *complexity management* and *code reusability*. In this paper we present the object-oriented programming methodology which is used for the definition and encapsulation of properties of several types of DR problems. Complexity management property of object-oriented modeling makes it possible to subdivide large scale and complex operating systems to simple subsystems and apply DR on the simple subsystems instead of the whole system. The concepts of object-oriented modeling are discussed in the literature [11] and will not be discussed here. UML modeling language [11] is used to show classes, attributes, methods and relationships between them in a standard form.

Extended Kalman filtering (EKF) and artificial neural networks (ANNs) are used for dynamic data reconciliation (DDR) by implementing defined concepts and classes. Classes defined for these methods are applied on a case study to show the performance of the proposed framework for data reconciliation.

†To whom correspondence should be addressed.
E-mail: Ali_Farzi@yahoo.com

CLASS DEFINITIONS

Data reconciliation as a concept has some attributes as measurements, reconciled values, and variance-covariance matrices and can form a class. Some methods can be defined for performing data acquisition and reconciliation. The general DR problem can be defined as:

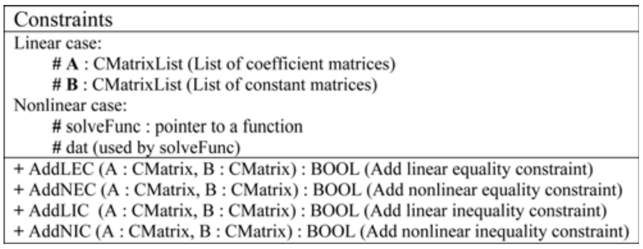
$$\underset{x}{\mathrm{Min}}(\mathbf{y}-\mathbf{x})^T\mathbf{W}(\mathbf{y}-\mathbf{x}) \tag{1}$$

subject to constraints.

where **x** and **y** are vectors of true values and measurements, respectively, and **W** is a weight matrix which is usually set to the inverse of covariance matrix of measurement errors (**y**−**x**). Constraints are usually mass and energy balance equations of the processing system and can be linear or nonlinear, and steady-state or dynamic depending on the nature of the system. It is a quadratic programming optimization problem that can be solved by different optimization methods.

1. Steady-State Processes

For a steady-state system, the accumulation term of mass and energy balance equations is zero. Thus for a linear steady state-system, the model of the system can be written in matrix form as:



Constraints

Linear case:

A : CMatrixList (List of coefficient matrices)

B : CMatrixList (List of constant matrices)

Nonlinear case:

solveFunc : pointer to a function

dat (used by solveFunc)

+ AddLEC (A : CMatrix, B : CMatrix) : BOOL (Add linear equality constraint)
+ AddNEC (A : CMatrix, B : CMatrix) : BOOL (Add nonlinear equality constraint)
+ AddLIC (A : CMatrix, B : CMatrix) : BOOL (Add linear inequality constraint)
+ AddNIC (A : CMatrix, B : CMatrix) : BOOL (Add nonlinear inequality constraint)

Fig. 1. A view of *Constraints* class and some of its methods and attributes.

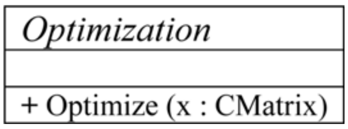
$$\mathbf{Ax=B} \tag{2}$$

where **x** is the vector of model variables and contains measured and unmeasured variables. **A** and **B** are coefficients and constants matrices, respectively.

Before defining DR classes, a class must be defined to gather information about equality or inequality and linear or nonlinear constraints. The class is shown in Fig. 1 and is called *Constraints* class. The class can store multitudes of constraints by using its attributes and methods. Some methods are not appearing here in class definition.

As described before, UML modeling language is used to show the structure of classes. In the above figure, class name, attributes and methods are shown in separate boxes. The −, #, and + signs represent private, protected and public members respectively. *CMatrix* class is defined for storing matrices and manipulating matrix operations, while *CMatrixList* class is used for collecting multitudes of matrices.

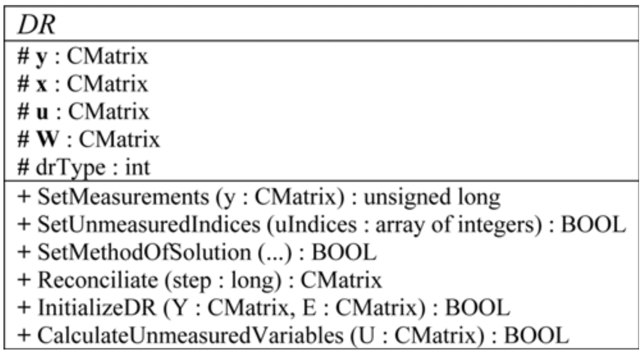
A class can also be defined for optimization problems. It inherits its members of *Constraints* class and is defined generally so that any optimization method can inherit from this class. The class is called *Optimization*. It has no specific attribute and has only general methods for performing optimization. The class is made abstract in order to prevent users from creating objects directly from it. Abstract classes are shown by italic font in diagrams.



Optimization

+ Optimize (x : CMatrix)

Fig. 2. Optimization class.



DR

y : CMatrix

x : CMatrix

u : CMatrix

W : CMatrix

drType : int

+ SetMeasurements (y : CMatrix) : unsigned long
+ SetUnmeasuredIndices (uIndices : array of integers) : BOOL
+ SetMethodOfSolution (...) : BOOL
+ Reconciliate (step : long) : CMatrix
+ InitializeDR (Y : CMatrix, E : CMatrix) : BOOL
+ CalculateUnmeasuredVariables (U : CMatrix) : BOOL

Fig. 3. Structure of *DR* class.

The data reconciliation class (*DR*) encapsulates all attributes and methods concerning DR problem definition. A pointer to an *Optimization* object is declared as an attribute of this class (*aggregation* relationship). The structure of the class is shown in Fig. 3.

Method *SetMethodOfSolution* initializes pointer to optimization class using its input arguments. The initialization method of the class is *InitializeDR* which performs all required tasks before performing DR. Because *Constraints* class manipulates all linear and nonlinear equality constraints, there is no need to classes for Linear DR and Nonlinear DR.

If some of the model variables are unmeasured, they must be removed from the model before DR is performed. Then the remaining measured variables can be reconciled, and observable unmeasured variables can be estimated by using reconciled measured ones. In this case Eq. (2) can be written as:

$$\mathbf{A_1z+A_2u=B} \tag{3}$$

where $\mathbf{A}=[\mathbf{A_1}\ \mathbf{A_2}]$ and $\mathbf{x}=[\mathbf{z}^T\ \mathbf{u}^T]^T$. **z** and **u** are vectors of measured and unmeasured variables, respectively. Matrix $\mathbf{A_2}$ can be decomposed into two orthogonal matrices of **Q** and **R** using QR decomposition method [12].

$$\begin{aligned} &\mathbf{A_2=QR} \\ &\mathbf{Q}^T\mathbf{Q=I} \\ &\mathbf{R}^T\mathbf{R=I} \\ &\mathbf{Q}=[\mathbf{Q_1}\ \ \mathbf{Q_2}] \\ &\mathbf{R}=\begin{bmatrix} \mathbf{R_1} & \mathbf{R_2} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \end{aligned} \tag{4}$$

The following property holds:

$$\mathbf{Q_2^TA_2=0} \tag{5}$$

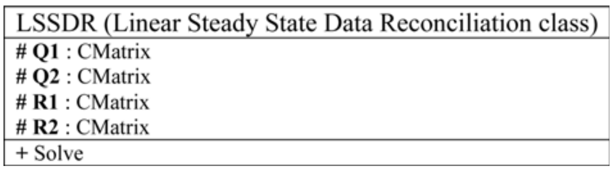
Thus the new model equations can be derived by removing unmeasured variables as below:

$$\begin{aligned} &\mathbf{A_zz=Q_2^TB} \\ &\mathbf{A_z=Q_2^TA_1} \end{aligned} \tag{6}$$

which contains only measured variables and has the same form of Eq. (2). This method can also be applied for linear dynamic systems.

A class is defined for steady-state processes which inherits members of the *DR* class. It is defined for generality and is called *SSDR* and has no specific attributes, but has some methods for performing steady-state calculations which are common for both linear and nonlinear systems.

Now a new class can be defined for encapsulating the above parameters together with inheriting members of *SSDR* class. This class, which is called *LSSDR*, is shown in Fig. 4. Matrices **A** and **B** can be implemented by using the *Constraints* class without any need to include in *LSSDR* definition. Because the solution of Eq. (1) with



LSSDR (Linear Steady State Data Reconciliation class)

Q1 : CMatrix

Q2 : CMatrix

R1 : CMatrix

R2 : CMatrix

+ Solve

Fig. 4. Structure of *LSSDR* class.

DDR (Dynamic Data Reconciliation class)
initialized : BOOL
+ InitializeDDR (x0 : CMatrix, covX0 : CMatrix, y : CMatrix, covY : CMatrix) : BOOL + AddNewMeasurement (y : CMatrix) : BOOL + SetDDRMethod (ddrMethod : DDRMethod) : BOOL + GetCalculatedData (step : long) : CMatrix + TerminateDDR ( ) : CMatrix

Fig. 5. Structure of *DDR* class.

linear constrains (Eq. (2)) is straightforward [12], no more class definitions are needed to handle this type of DR problems. Solution of LSSDR problem is performed by *Solve* method of *LSSDR* class.

2. Dynamic Processes

For dynamic systems, a class, namely *DDR*, is defined for manipulating dynamic DR problems. Its structure is shown in Fig. 5. Any derived object from this class is initialized by calling *InitializeDDR* method. *SetDDRMethod* method is called by *InitializeDDR* before initializing DDR. This method initializes a pointer to a *DDRMethod* class object which is described below. Kalman filtering (KF) and ANNs are implemented for NDDR by inheriting from this class. During DDR, *AddNewMeasuremnt*, *GetCalculatedData* and *Reconciliate* methods which are inherited from *DR* class are called recursively to perform DDR in each time step of getting measurements. After completion of DDR, *TerminateDDR* method is called for terminating DDR.

As the solution of linear and nonlinear DDR problems is not straightforward and several methods are developed by different researchers, a general class is needed to encapsulate common properties of these methods. For this reason, a class, namely *DDRMethod* is defined which is shownin Fig. 6.

The *linear* attribute determines whether the DDR method is for linear or nonlinear cases. An object of this class is defined as an attribute of *DDR* class which is used for DDR by applying user-specific methods. This class has three main methods, which are used for DR. These methods are called by their counterparts from *DDR* class for solving DR problems depending on the user's DDR method.

DDRMethod (Dynamic Data Reconciliation Method class)
linear : BOOL
+ CustomInitializeDDR ( ) : BOOL + CustomReconciliate (y : CMatrix) : BOOL + CustomTerminateDDR ( ) : BOOL

Fig. 6. Structure of *DDRMethod* class.

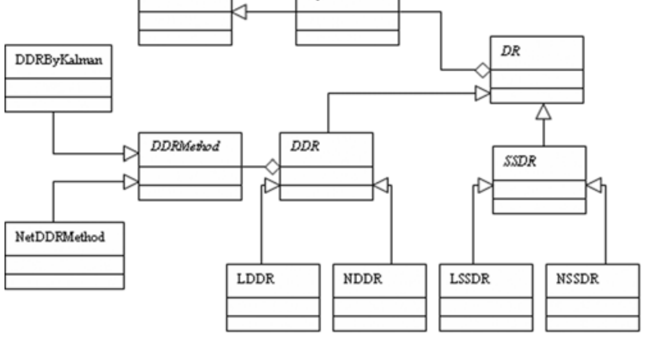


Fig. 7. Relationships between defined classes.

These methods perform different operations depending on different methods for DDR (*polymorphism* concept of OOP).

A class is defined for linear DDR (*LDDR*) which is derived from *DDR* class and only customizes its methods for LDDR. Another class, namely *NDDR*, is defined for nonlinear dynamic DR. It is also derived from *DDR* class and customizes its methods.

Relationships between defined classes are shown in Fig. 7. Lines with triangles show an inheritance relationship, while lines with diamonds represent an aggregation relationship.

APPLICATION OF DEVELOPED CLASSES FOR DATA RECONCILIATION

Defined classes were coded in Visual C++ programming environment for developing a general purpose software tool for DR. Any types of DR can be performed with this software including linear, nonlinear, steady-state and dynamic DR problems. The software will be discussed later in this paper.

For linear steady-state DR the solution of problem (1) is straightforward and there is no need for a specific optimization method. Therefore, no extra class definitions other than the defined ones (*DR*, *SSDR* and *LSSDR*) are needed. For dynamic cases including linear and nonlinear problems, different methods are developed [4-10]. Therefore, the *DDRMethod* class was defined and implemented by *DDR*, *LDDR* and *NDDR* classes by using the aggregation relationship to facilitate application of different methods for DDR. This class is a general class and users can apply their specific methods by deriving their classes from this class using the inheritance relationship and customizing methods of *DDRMethod*. This class is made abstract and no instance of it can be created directly during run-time. It requires that users inherit new classes from it and then create objects from their-own classes.

1. Kalman Filtering Applied as a Class for Dynamic Data Reconciliation

As it is shown by different authors, Kalman filtering can detect and remove noise from measurements and states [13-15] and is a specific case of DR [12] and can be used for DDR. To illustrate the suitability of *DDRMethod* class, Kalman filtering (KF) and extended Kalman filtering (EKF) methods were classified in a class, namely *DDRByKalman*, and inherited methods and attributes of *DDRMethod* class. For KF the discrete state-space form of the model of the processing system is needed. The general form of continuous state-space model of a processing system can be written as:

$$\frac{\mathbf{dx}}{\mathbf{dt}}=\mathbf{f}(\mathbf{x},\mathbf{u})$$

$$\mathbf{y}=\mathbf{h}(\mathbf{x},\mathbf{u})+\varepsilon \tag{7}$$

In the above equations, **x**, **u**, and **y** are vectors of state, input and measured variables, respectively. **f** and **h** can be linear or nonlinear functions of **x** and **u**. ε is measurement error. It is considered that no disturbances or modeling errors are present.

Discrete EKF is used for nonlinear cases and requires that the above model to be linearized first and then discretized recursively in each time step. The details are not presented here. The following equations are the linearized and discretized form of the above model.

$$\mathbf{x}[k]=\mathbf{A}.\mathbf{x}[k-1]+\mathbf{B}.\mathbf{u}[k-1]+\mathbf{M}$$

$$\mathbf{z}[k]=\mathbf{J}_h\mathbf{x}[k]+\varepsilon$$

$$\mathbf{A}\triangleq \mathbf{e}^{J_{f_1}T},\ \ \mathbf{B}\triangleq(\mathbf{A}-\mathbf{I})\mathbf{J}_{f_1}^{-1}\mathbf{J}_{f_2},\ \ \mathbf{M}\triangleq(\mathbf{I}-\mathbf{A})[\mathbf{x}_1-\mathbf{J}_{f_1}^{-1}\mathbf{f}(\mathbf{x}_1,\mathbf{u}_1)]-\mathbf{B}\mathbf{u}_1 \quad (8)$$

where $\mathbf{J}_{f_1}$ and $\mathbf{J}_{f_2}$ are Jacobian matrices of **f** with respect to **x** and **u** evaluated at $\mathbf{x}_1$ and $\mathbf{u}_1$. $\mathbf{J}_h$ is the Jacobian matrix of **h** with respect to **x** at $\mathbf{x}_1$. It is considered that no measurements of input variables contain errors. T and k are sampling time and step, respectively.

EKF steps are just like traditional KF except that in each or some time step, a linearization on the model equations must be done in order to get a linear set of state-space equations for use in KF. The recursive nature of KF makes it suitable for *on-line* dynamic DR. In KF *a priori* and *a posteriori* states are obtained by using state-space equations. For the KF method, linear state-space equations are defined as [13]:

$$\begin{aligned}&\mathbf{x}[k]=\Phi[k-1]\mathbf{x}[k-1]+\mathbf{B}[k-1]\mathbf{u}[k-1]+\mathbf{w}[k-1]\\&\mathbf{y}[k]=\mathbf{C}[k]\mathbf{x}[k]+\mathbf{v}[k]\end{aligned} \quad (9)$$

where **w** and **v** are white noise in states and measurements:

$$\begin{aligned}&\mathbf{E}(\mathbf{w}[k])=\mathbf{0}, && \mathbf{Cov}(\mathbf{w}[k])=\mathbf{Q}[k]\\&\mathbf{E}(\mathbf{v}[k])=\mathbf{0}, && \mathbf{Cov}(\mathbf{v}[k])=\mathbf{R}[k]\end{aligned} \quad (10)$$

The initial condition of the above differential equation is shown as **x**[0], where:

$$\mathbf{E}(\mathbf{x}[0])=\hat{\mathbf{x}}[0], \quad \mathbf{Cov}\ (\mathbf{x}[0])=\mathbf{P}[0] \quad (11)$$

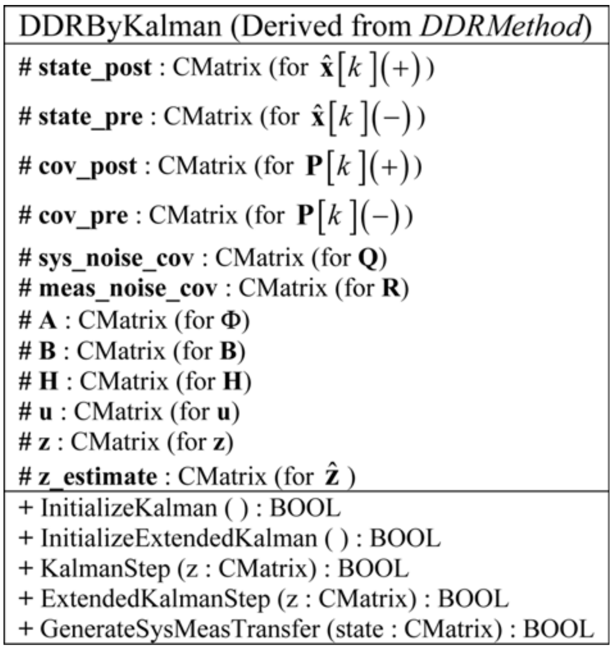
where $\hat{\mathbf{x}}[0]$ are true values of **x**[0]. The KF steps are:

a. State estimate extrapolation :: $\hat{\mathbf{x}}[k](-)=\Phi[k-1]\hat{\mathbf{x}}[k-1](+)$

b. Error covariance extrapolation::
$$\mathbf{P}[k](-)=\Phi[k-1]\mathbf{P}[k-1](+)\Phi[k-1]^T+\mathbf{Q}[k-1]$$

c. KF matrix gain calculation::
$$\mathbf{K}[k]=\mathbf{P}[k](-)\mathbf{C}[k]^T[\mathbf{C}[k]\mathbf{P}(-)\mathbf{C}[k]^T+\mathbf{R}[k]]^{-1}$$

d. State estimate observational update::
$$\hat{\mathbf{x}}[k](+)=\hat{\mathbf{x}}[k](-)+\mathbf{K}[k](\mathbf{z}[k]-\mathbf{C}[k]\hat{\mathbf{x}}[k](-)]$$



DDRByKalman (Derived from *DDRMethod*)

state_post : CMatrix (for $\hat{\mathbf{x}}[k](+)$)
state_pre : CMatrix (for $\hat{\mathbf{x}}[k](-)$)
cov_post : CMatrix (for $\mathbf{P}[k](+)$)
cov_pre : CMatrix (for $\mathbf{P}[k](-)$)
sys_noise_cov : CMatrix (for **Q**)
meas_noise_cov : CMatrix (for **R**)
A : CMatrix (for **Φ**)
B : CMatrix (for **B**)
H : CMatrix (for **H**)
u : CMatrix (for **u**)
z : CMatrix (for **z**)
z_estimate : CMatrix (for $\hat{\mathbf{z}}$)
+ InitializeKalman ( ) : BOOL
+ InitializeExtendedKalman ( ) : BOOL
+ KalmanStep (z : CMatrix) : BOOL
+ ExtendedKalmanStep (z : CMatrix) : BOOL
+ GenerateSysMeasTransfer (state : CMatrix) : BOOL

Fig. 8. Structure of *DDRByKalman* class.

e. Error covariance update :: $\mathbf{P}[k](+)=(\mathbf{I}-\mathbf{K}[k]\mathbf{C}[k])\mathbf{P}[k](-)$

f. Measurement estimation :: $\hat{\mathbf{z}}[k]=\mathbf{C}\hat{\mathbf{x}}[k](+)$

All of the above steps are performed for k=1, 2, …. The initial condition is:

$$\hat{\mathbf{x}}[0](+)=\hat{\mathbf{x}}[0], \quad \mathbf{P}[0](+)=\mathbf{P}[0] \quad (12)$$

According to the above discussion, the structure of *DDRByKalman* is defined and is shown in Fig. 8.

This class has inherited and customized methods of *DDRMethod* class. First, the *CustomInitializeDDR* method is called by *InitializeDDR* from *LDDR* or *NDDR* class, which itself calls the *InitializeKalman* method. Then *InitializeKalman* determines whether the model is linear or nonlinear by using its input arguments. If the model is nonlinear, it calls *InitializeExtendedKalman*. After initialization of the filter, *KalmanStep* is called repeatedly for filtering measurements arrived at in each time step. If the model is nonlinear, this method calls *ExtendedKalmanStep*. Then *ExtendedKalmanStep* calls *GenerateSysMeasTransfer* for linearizing and discretizing of the model. In next step, *state_pre*, *state_post*, *cov_pre*, *cov_post*, and *z_estimate* are updated. The last result of filtering in each step is stored in *z_estimate* which is then used by *LDDR* or *NDDR* objects as the reconciled values.

2. Artificial Neural Networks Applied as a Class for Dynamic Data Reconciliation

Another method that has been developed is based on feed-forward ANNs and is called NetDDR method. Advantages such as learning and estimation capabilities and applicability characteristic of the ANNs are the main reasons of their success in different fields such as fault detection, signal processing, identification and control. A complete review and discussion of the application of ANNs in different areas of chemical engineering can be found in [16].

A feed-forward ANN is comprised of units, namely neurons similar to a brain neural network. Any neuron can have any number of inputs. Each neuron multiplies its inputs by their weight and then applies an operator such as summation on them and sends the output to a transfer function. The output from the transfer function is sent to other neurons. Neurons with the same inputs form a layer. If inputs are from the surroundings, the layer is called the input layer. The layer between outputs of ANN and the surroundings is called the output layer. The layers between these two layers are called hidden. Each layer can have one or more neurons. Depending on the complexity of the process to be trained, the number of hidden layers and neurons can change. Fig. 9 shows a feed-forward ANN which is shown in matrix form. Commonly, each neuron has one more input with unit value as bias. In the following figure the bias weights are shown by b.

Training of ANN is the adjustment of weights of inputs to the neurons such that the output of the ANN based on its inputs is very close or equivalent to the output of the real system.

In this research a type of ANN similar to system identification ANNs is developed and used for DDR. One standard model that has been used to represent general discrete-time nonlinear systems is the **n**onlinear **a**uto-regressive-**m**oving **a**verage (NARMA) model:

$$\mathbf{y}(k)=\mathbf{G}[\mathbf{y}(k-1), \mathbf{y}(k-2), \ldots, \mathbf{y}(k-n), \mathbf{u}(k), \mathbf{u}(k-1), \ldots, \mathbf{u}(k-n)] \quad (13)$$

where, **u**(k) is the vector of system inputs, **y**(k) is the vector of sys-

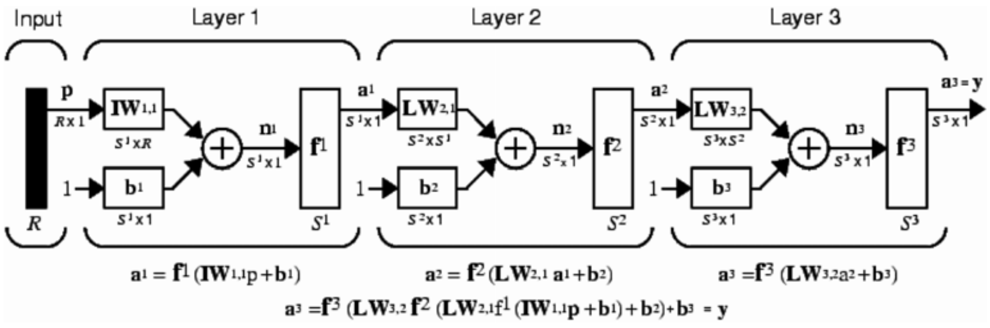


Fig. 9. Different layers of neurons in an ANN shown in matrix form.

tem outputs (measurements in here), and n is the number of delays. For the identification phase, a neural network must be trained to approximate the nonlinear function **G**. The above model was modified and used for DDR:

$$\hat{\mathbf{y}}(k)=\mathbf{G}[\mathbf{y}(k), \mathbf{y}(k-1), \hat{\mathbf{y}}(k-1), \ldots, \mathbf{y}(k-n), \hat{\mathbf{y}}(k-n), \mathbf{u}(k), \mathbf{u}(k-1), \ldots, \mathbf{u}(k-n)] \quad (14)$$

where $\hat{\mathbf{y}}$ is the vector of reconciled values. The problem of using this model is that for training a neural network to estimate the function **G** that minimizes mean square error, dynamic back-propagation must be implemented ([17,18]), which can be quite slow. One proposed solution is the use of approximate models to represent the system [19]:

$$\hat{\mathbf{y}}(k)=\mathbf{f}[\mathbf{y}(k), \mathbf{y}(k-1), \hat{\mathbf{y}}(k-1), \ldots, \mathbf{y}(k-n), \hat{\mathbf{y}}(k-n), \mathbf{u}(k-1), \ldots, \mathbf{u}(k-n)] +\mathbf{g}[\mathbf{y}(k), \mathbf{y}(k-1), \hat{\mathbf{y}}(k-1), \ldots, \mathbf{y}(k-n), \hat{\mathbf{y}}(k-n), \mathbf{u}(k-1), \ldots, \mathbf{u}(k-n)]\mathbf{u}(k) \quad (15)$$

This model is in companion form, where the plant input **u**(k) is not contained inside the nonlinearity. Fig. 10 shows the structure of the ANN representation of the above model.

In the following figure, the ⊗ notation indicates a multiplying neuron while the ⊕ notation shows a summation neuron. TD boxes represent delayed inputs to the network. As indicated in the figure, this network is comprised of one input layer, some hidden layers for functions **f** and **g** and one output layer. The number of neurons in the first layer is equal to the number of inputs. Then it varies linearly with increasing the layer number and reaches a value specified by the user for the middle layer. Then it again varies linearly while reaching the number of outputs. If there are p inputs and m measurements, with n number of delays, the number of inputs to each of the ANNs will be:

$$p(n+1)+m(2n+1) \quad (16)$$

where coefficient 2 in the second term is because of including reconciled values from previous steps. Thus according to the above figure, total number of inputs to the whole ANN will be:

$$p+2[p(n+1)+m(2n+1)]=p(2n+3)+2m(2n+1) \quad (17)$$

The number of outputs of the whole ANN equals the number of measurements, i.e., m.

This method is encapsulated as a class which is derived from *DDRMethod* class. The class is called *NetDDRMethod* and its architecture is shown in Fig. 11.

The *CustomInitializeDDR* method is called at the start of DDR method which itself calls *Initialize* method for initializing attributes of the object. In each step when measurements and plant inputs are received, the *CustomReconciliate* method is called which itself calls *NetStep* method.

The *net* attribute is also a class object and contains all attributes and methods required for training and simulation of an ANN (not shown here). Its most important method is the *Simulate* method which takes inputs of the network and returns outputs after calculation. The *NetStep* method constructs inputs to the ANN using received plant inputs, measurements and delays, and then calls *Simulate* method of *net* attribute. At last it returns the results.

3. DCON, an Object-Oriented Software for Performing Data Reconciliation

An academic software tool, namely DCON, is developed by coding all of the above classes and concepts using Visual C++ programming environment for performing data reconciliation. As can be seen in Fig. 12, the software has a graphical user interface (GUI) which

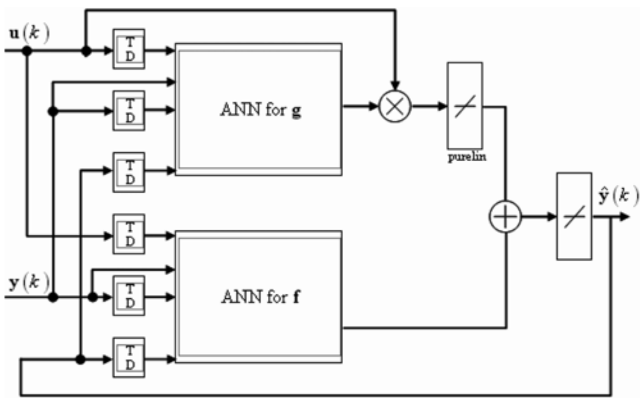
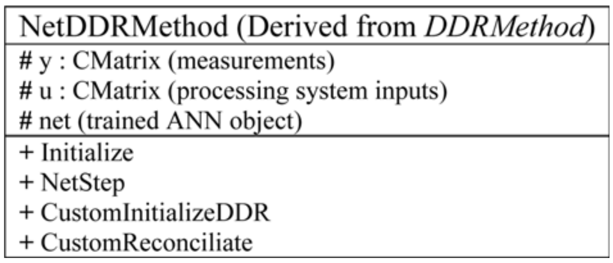


Fig. 10. ANN model for estimating nonlinear functions of f and g.



NetDDRMethod (Derived from *DDRMethod*)

y : CMatrix (measurements)
u : CMatrix (processing system inputs)
net (trained ANN object)

+ Initialize
+ NetStep
+ CustomInitializeDDR
+ CustomReconciliate

Fig. 11. Structure of *NetDDRMethod* class.

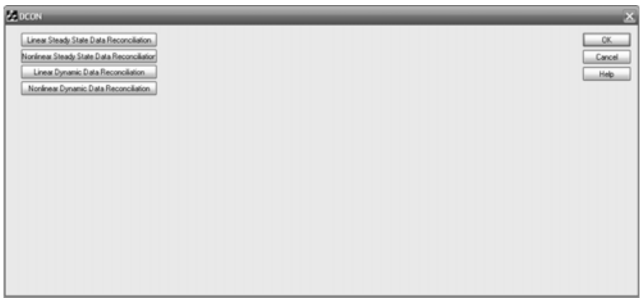
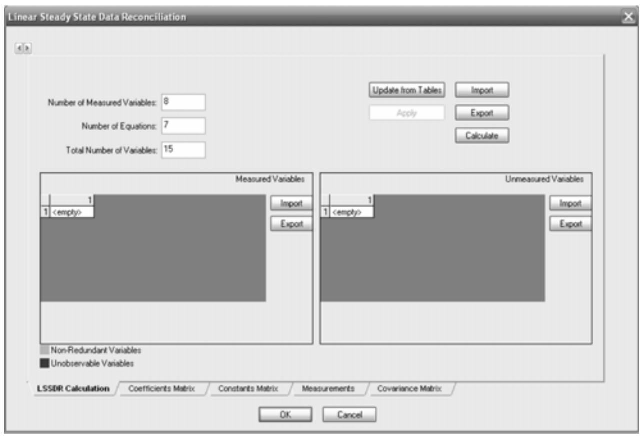
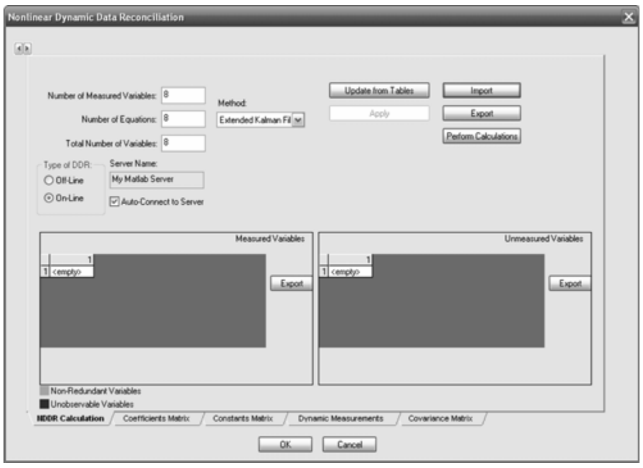

DCON
Linear Steady State Data Reconciliation
Nonlinear Steady State Data Reconciliation
Linear Dynamic Data Reconciliation
Nonlinear Dynamic Data Reconciliation
OK
Cancel
Help

Fig. 12. GUI of DCON software developed for DR.


Linear Steady State Data Reconciliation

Number of Measured Variables: 8
Number of Equations: 7
Total Number of Variables: 15

Fig. 13. GUI of performing LSSDR.


Nonlinear Dynamic Data Reconciliation

Number of Measured Variables: 8
Number of Equations: 8
Total Number of Variables: 8
Method: Extended Kalman Fil
Type of DDR: ○ Off-Line [✓ ⊙ On-Line]
Server Name: My Matlab Server
☑ Auto-Connect to Server

Fig. 14. GUI for performing NDDR.

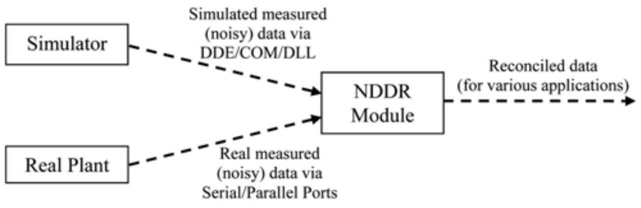

Simulator
Simulated measured (noisy) data via DDE/COM/DLL
NDDR Module
Reconciled data (for various applications)
Real Plant
Real measured (noisy) data via Serial/Parallel Ports

Fig. 15. Data communication of NDDR module with simulator/real plant.

leads the user to the type of DR that he/she wants to do. At the left hand side of GUI, there are four push buttons for performing LSSDR, NSSDR, LDDR and NDDR, respectively. By clicking any of these buttons a new window is opened and the user can enter data and perform DR.

For example, by clicking the first button a window, which is shown in Fig. 13, is displayed for LSSDR. As can be seen, it has five tabs. In the first tab (**LSSDR Calculation** tab), the number of measured variables, number of linear model equations and total number of variables can be entered. Estimation of unmeasured observable variables can also be done by the software. After LSSDR is performed, the results will be shown in two distinct tables for measured and unmeasured variables. The other tabs (not shown here) are for entering coefficients of model equations (**A**), constants of model equations (**B**), measurements to be reconciled and covariance matrix of measurements, respectively.

After entering required data and clicking **Calculate** button in first tab, if there exist any measurements in the **Measurements** tab and all data are correct, LSSDR will be performed for each column of the measurements entered in the measurements tab and results will be shown in two tables of **LSSDR Calculation** tab. The user can import or export all entered data or results individually or together, using **Export** and **Import** buttons located in different locations of the window.

For performing DDR, two buttons exist in the GUI of the software, one for linear and the other for nonlinear case. In this part, different aspects of the NDDR module will be discussed. By clicking **Nonlinear Dynamic Data Reconciliation** button in the GUI of the software, a window similar to Fig. 14 will appear. It is similar to the LSSDR window with some extra data needed for DDR. Because DDR is for nonlinear cases, no coefficients or constants matrices can be entered. Instead, users can write their own dynamic simulation programs using any programming software which supports DDE or COM communication mechanism. If the **On-Line** radio button is selected, the software will request measurements from the simulation program; but if the **Off-Line** is selected, it will use measurements entered in **Measurements** tab of Fig. 14.

After the required data are entered and **Perform Calculations** button is clicked, the software will trigger the simulation program to start running and will wait measurements to be sent from the simulation program via DDE. Other types of data communication such as communication by component object model (COM), and via serial and parallel ports for connecting to real plants are also considered in this software (Fig. 15). The user must add random noise to the simulation results that DR must be done on them and then send them to DR software by defined functions for DDE connection mechanism and obtain reconciled data. As the new measurements arrive, the software performs DDR using the method selected by the user and after reconciliation it sends reconciled data to the simulation program for storing or plotting purposes. For terminating DDR, simulation program must send a **Stop** signal to the software. All aspects of the software are completely discussed in its technical report.

CASE STUDY

Measurements can be obtained dynamically from transient simulation or from a real plant. Currently, dynamic simulation of processing systems is used for DR. In order that sending measurements from dynamic simulation is similar to a real plant, the DR software has completely been separated from simulation programs. Currently, two types of DDE and COM data communication mechanisms between DR software and simulation programs have been considered. These mechanisms are embedded into *DDRMethod* class and DR software in such a way that users can write their simulation programs with any programming software that supports one of DDE or COM mechanisms and send simulated measurements to the DR software.

To show the performance of object-oriented methodology applied on DR and also two developed classes for NetDDR and EKF methods, DDR applied on a distillation column and the results are presented here. In each time step, simulation results are corrupted by white noise with zero mean and a specified value of standard deviation according to the type of signal, and then sent to the requesting program. The objective is to reconcile temperature measurements. The distillation column under study has 6 trays with one partial reboiler and one partial condenser as shown in Fig. 16; thus there are 8 stages.

Feed with a composition of 70 mol% water and 30 mol% methanol, enters on the 4th stage from the bottom of the column with temperature of 78 °C and flow rate of 15 kgmol/min. According to Fig. 17, trays of the column are numbered from bottom. All symbols are defined in Appendix A.

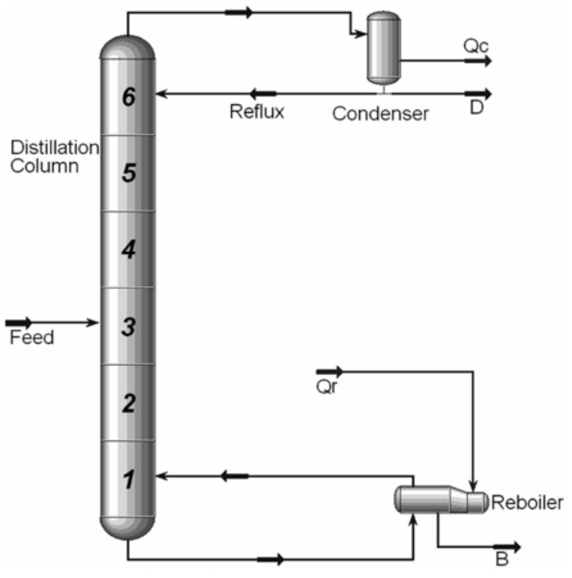


Fig. 16. Implemented distillation column.

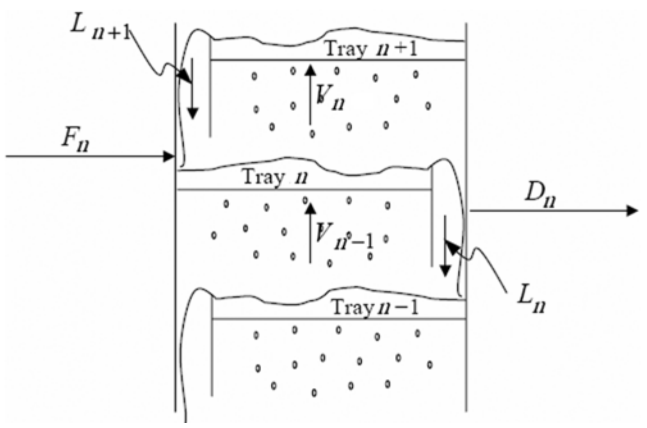


Fig. 17. The diagram used for modeling of distillation system.

MODELING AND SIMULATION OF DISTILLATION COLUMN

For modeling and simulation of distillation column, mass and energy balance equations must be written. Total mass balance equation based on Fig. 17 for stage n can be written as:

$$v_{Ln}\frac{d}{dt}(\rho_{Ln})+v_{Vn}\frac{d}{dt}(\rho_{Vn})=L_{n+1}-L_n-D_n+V_{n-1}-V_n+F_n \qquad (18)$$

where volumetric liquid and vapor hold-ups are assumed constant. Vapor holdup of a stage, v_{Vn}, is ignored compared to liquid holdup, v_{Ln}. Thus Eq. (18) can be written as:

$$v_{Ln}\frac{d}{dt}(\rho_{Ln})=L_{n+1}-L_n-D_n+V_{n-1}-V_n+F_n \qquad (19)$$

Stage outputs (D_n) can be represented as:

$$D_n=\begin{cases}\frac{L_n}{R} & n=N+2\\ 0 & n\neq N+2\end{cases} \qquad (20)$$

Mass balance equation for component i can also be written in the following form, neglecting vapor holdup:

$$v_{Ln}\frac{d}{dt}(\rho_{Ln}x_{i,n})=L_{n+1}x_{i,n+1}-L_nx_{i,n}-D_nx_{i,n}+V_{n-1}y_{i,n-1}$$
$$-V_ny_{i,n}+F_nx_{F,n} \qquad (21)$$

Assuming that the vapor phase is ideal and variation of liquid volume with pressure is negligible, the equilibrium relation for *i*-th component can be written as:

$$y_iP=x_i\gamma_iP_i^* \qquad (22)$$

or

$$y_i=K_ix_i \qquad (23)$$

where

$$K_i=\frac{\gamma_iP_i^*}{P} \qquad (24)$$

In practice, liquid and vapor phases on stages are not in equilibrium. In order to determine the actual rate of mass transfer, a parameter namely *plate efficiency* is used. This parameter is defined as:

$$E_n=\frac{x_{i,n+1}-x_{i,n}}{x_{i,n+1}(e)-x_{i,n}} \qquad (25)$$

where the parameter $x_{i,n+1}(e)$ is the composition of the component i in liquid phase at equilibrium with vapor leaving stage n+1 and can be replaced by Eq. (23).

Differentiating the left hand side of Eq. (21) results in:

$$v_{Ln}\frac{d}{dt}(\rho_{Ln}x_{i,n})=v_{Ln}\left(\rho_{Ln}\frac{dx_{i,n}}{dt}+x_{i,n}\frac{d\rho_{Ln}}{dt}\right) \qquad (26)$$

Multiplying Eq. (21) by E_n and using Eqs. (23) and (26), finally the following results can be obtained:

$$\frac{dx_{i,n}}{dt}=J_{1,n}x_{i,n+1}-J_{2,n}x_{i,n}+J_{3,n}x_{i,n-1}+J_{4,n} \tag{27}$$

$$J_{1,n}=\frac{L_{n+1}}{E_n v_{Ln}\rho_{Ln}} \tag{28}$$

$$J_{2,n}=\frac{E_n(K_{i,n}V_n+D_n)+L_n-(1-E_n)(L_n-L_{n+1})}{E_n v_{Ln}\rho_{Ln}}+\frac{1}{\rho_{Ln}}\frac{d\rho_{Ln}}{dt} \tag{29}$$

$$J_{3,n}=\frac{V_{n-1}K_{i,n-1}}{v_{Ln}\rho_{Ln}} \tag{30}$$

$$J_{4,n}=\frac{F_n x_{F,n}}{v_{Ln}\rho_{Ln}} \tag{31}$$

The last term in Eq. (29) can also be replaced by the result of Eq. (19).

An energy balance equation is needed to calculate vapor flow rates. Neglecting the effect of mixing and generation of heat, it can be written as:

$$\begin{aligned}F_n h_{F,n}+V_{n-1}\sum_{i=1}^{m}(\lambda_i y_{i,n-1}+S_{V,i}y_{i,n-1}T_{n-1})+L_{n+1}\sum_{i=1}^{m}S_{L,i}x_{i,n+1}T_{n+1}\\ -V_n\sum_{i=1}^{m}(\lambda_i y_{i,n}+S_{V,i}y_{i,n}T_n)-D_n\sum_{i=1}^{m}S_{L,i}x_{i,n}T_n-L_n\sum_{i=1}^{m}S_{L,i}x_{i,n}T_n\\ =\left[v_{Ln}\sum_{i=1}^{m}S_{L,i}x_{i,n}+v_{Vn}\sum_{i=1}^{m}S_{V,i}y_{i,n}+M_{c,n}\right]\frac{dT_n}{dt}+q_{c,n}-Q_{c,n}\end{aligned} \tag{32}$$

For computing temperature variations of each stage, the column model can be completed by the implementation of equilibrium relations. Summing Eq. (22) on number of components results in:

$$P=\sum_{i=1}^{m}\gamma_i x_i P_i^* \tag{33}$$

Differentiating the above equation with respect to time will result in:

$$\frac{dP}{dt}=\sum_{i=1}^{m}\left(\gamma_i x_i\frac{dP_i^*}{dt}+\gamma_i P_i^*\frac{dx_i}{dt}+x_i P_i^*\frac{d\gamma_i}{dt}\right) \tag{34}$$

Differentiation of vapor pressure, P_i^*, with respect to temperature and activity coefficient, γ_i, with respect to temperature and molar compositions results in:

$$dP_i^*=\frac{dP_i^*}{dT}dT \tag{35}$$

$$d\gamma_i=\left(\frac{d\gamma_i}{dT}\right)_{T,x_i}dT+\sum_{j=1}^{m}\left(\frac{d\gamma_i}{dx_j}\right)_{T,x_{l\neq j}}dx_j \tag{36}$$

By substituting the above equations in Eq. (34) and assuming isobaric conditions, the following equation can be obtained for the two component system of methanol-water:

$$\frac{dT}{dt}=-\frac{\frac{dx_1}{dt}\left(\gamma_1P_1^*+x_1P_1^*\frac{d\gamma_1}{dt}-\gamma_2P_2^*+x_2P_2^*\frac{d\gamma_2}{dt}\right)}{\sum_{i=1}^{2}\left(\gamma_i x_i\frac{dP_i^*}{dt}+x_iP_i^*\frac{d\gamma_i}{dt}\right)} \tag{37}$$

In this research the liquid phase activity coefficients, γ_i, are calculated by Wilson equation and vapor pressures of pure components by using the Antoine equation:

$$\ln(\gamma_i)=1-\ln\left(\sum_{j=1}^{m}x_jG_{i,j}\right)-\sum_{k=1}^{m}\left(\frac{x_kG_{k,i}}{\sum_{j=1}^{m}x_jG_{k,j}}\right) \tag{38}$$

$$\log P_i^*=A_i-\frac{B_i}{C_i+T};\ \ P_i^*[\mathrm{Pa}];\ \ T[\mathrm{K}] \tag{39}$$

Values of $G_{i,j}$ constant parameters of Antoine equation are presented in [20].

After model of the processing system is developed, the system can be simulated. The method for the solution of the above equations is as below:

a) Set all dT/dt=0 and dρ/dt=0 for all stages,

b) Generate an initial guess for all variables such as gas and liquid temperatures and compositions in each stage,

c) Solve the set of Eqs. (19) and (32), simultaneously for all stages and use Eq. (20) to obtain liquid and vapor flow rates of each stage,

d) Calculate dρ_{LN}/dt for each stage from Eq. (19),

e) Calculate activity coefficients, vapor pressures, total pressure, *K*-values and vapor compositions for all stages from Eqs. (38), (30), (33), (24) and (23), respectively,

f) Calculate plate-efficiencies by Eq. (25),

g) Calculate liquid compositions and temperatures for each stage by solving Eqs. (27) to (31),

h) Repeat from step c by new obtained values while steady-state condition is satisfied (all derivatives with respect to time must become very small) or the last specified time for simulation has reached.

SIMULATION RESULTS

According to the above algorithm, a simulation program was written and the plant was simulated. For a reflux ratio of 3, the process reached steady-state in less than 120 minutes. After 180 minutes, a step change from 3 to 2 applied on reflux ratio. In this case, steady-state condition was established in less than 60 minutes. Fig. 18 shows temperature profiles for all stages.

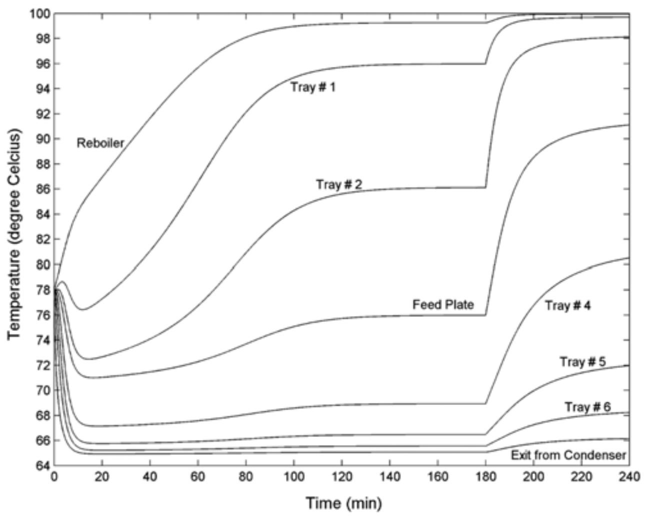


Fig. 18. Temperature profiles of the column with respect to time for reflux ratios of 3 and 2.

It is evident that temperature measurement is easier, faster and more economic than the measurement of liquid and gas compositions. As mentioned in the literature [21], for a distillation system, liquid and gas compositions can be calculated based on temperature measurements, known pressure and equilibrium relations. Thus, as the total pressure in a distillation column is assumed to be constant, only temperature measurements are needed to be reconciled. According to Eq. (32), for eight stages, there are eight measurements.

$$\mathbf{y}=[T_1, \ldots, T_{N+2}]^T \tag{40}$$

The reflux ratio is used as the input for the system of distillation column. In the following tests, simulation was started at steady-state condition for the reflux ratio of 3. Then a step change in reflux ratio from 3 to 6 was applied at time 10. The standard deviation of errors (SDE) for all noisy data is equal to 1.

1. DDR Using NetDDR Method

The developed class of *NetDDRMethod* is used for developing a GUI for NDDR using ANNs. Fig. 19 shows the developed GUI for training NetDDR ANN. As can be seen, it has three parts. In the first part, parameters of ANNs for **f** and **g** such as the number of layers and neurons can be specified. The second part is used for generating training data by simulation of the plant. The third part is for specifying training parameters. Before using ANN for DDR some input-output data of the plant must be generated for training ANN. This was done for a distillation column and 4000 input-output pairs of data from the plant were generated. These data include random values of reflux ratio, and true and noisy values of temperature measurements obtained by simulation of the plant. After generating training data and specifying required parameters and then clicking **Train Network** button, the network will be trained and prepared for DDR.

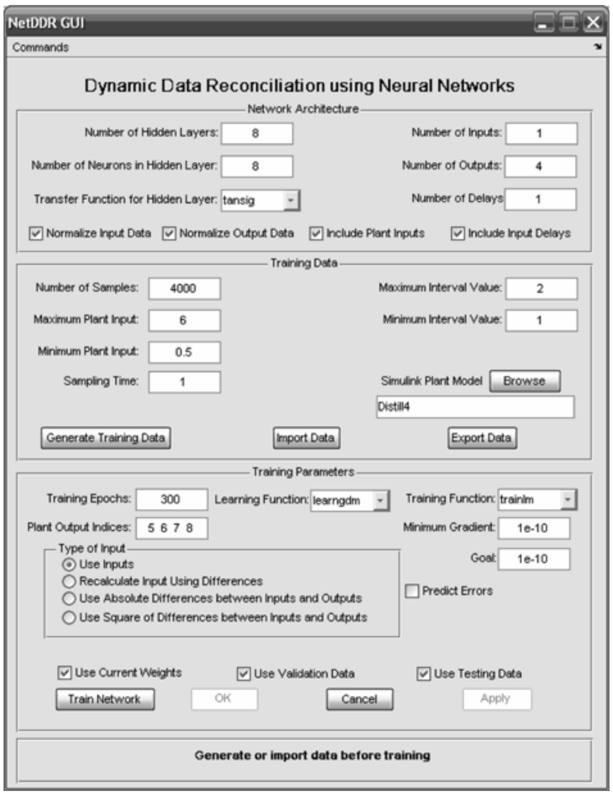

NetDDR GUI
Commands
Dynamic Data Reconciliation using Neural Networks
Network Architecture
Number of Hidden Layers: 8
Number of Inputs: 1
Number of Neurons in Hidden Layer: 8
Number of Outputs: 4
Transfer Function for Hidden Layer: tansig
Number of Delays: 1
[✓] Normalize Input Data [✓] Normalize Output Data [✓] Include Plant Inputs [✓] Include Input Delays
Training Data
Number of Samples: 4000
Maximum Interval Value: 2
Maximum Plant Input: 6
Minimum Interval Value: 1
Minimum Plant Input: 0.5
Sampling Time: 1
Simulink Plant Model: Distill4
Training Parameters
Training Epochs: 300
Learning Function: learngdm
Training Function: trainlm
Plant Output Indices: 5 6 7 8
Minimum Gradient: 1e-10
Goal: 1e-10
Type of Input
(● Use Inputs)
○ Recalculate Input Using Differences
○ Use Absolute Differences between Inputs and Outputs
○ Use Square of Differences between Inputs and Outputs
☐ Predict Errors
[✓] Use Current Weights [✓] Use Validation Data [✓] Use Testing Data
Generate or import data before training

Fig. 19. GUI used for training NetDDR ANN.

To use the trained network for NDDR, the NetDDR method must be selected in the GUI of DCON software for NDDR (Fig. 14). By selecting it and clicking on **Perform Calculations** button (Fig. 14), DCON software will trigger the simulation program and will wait for the simulated noisy measurements to be sent from it. During simulation, in each time step, white noises are added to the calculated values to simulate real measurements and then sent to DR software. When noisy measurements are received by DCON software, it reconciles measurements using *NetDDRMethod* class. It has been assumed that measurements contain no gross errors. Then the *NetDDRMethod* class uses the trained ANN for calculating reconciled values of measurements. Eight layers and eight neurons in hidden layers of the ANNs for **f** and **g** were considered. The Levenberg-Marquardt algorithm [22] was used for training.

In order to show the performance of NetDDR ANN for DDR of multiple measurements, measurements of stages 5 to 8 were selected for DDR. According to Eq. (17) with one delay in inputs, the number of inputs to the ANN is 29. After training the network using available training data, the network was used for DDR. A step change from 3 to 6 in reflux ratio was applied and measurements were reconciled by using the trained network. Fig. 20 shows the results of DDR with trained network using *NetDDRMethod* class.

2. DDR Using EKF Method

To compare the results of NetDDR method with EKF method, the above test was also repeated by EKF method. As mentioned before, the EKF method is also encapsulated into a class, namely *DDRByKalman*, which inherited the members of *DDRMethod*. Fig. 21 shows the results of NDDR using EKF method by applying *DDRByKalman* class.

By comparing the results of Fig. 20 and Fig. 21 it is evident that the NetDDR method gives better results than the EKF method. Also, the NetDDR method does not need any information about the model and state variables of the process. It is also faster than the EKF method and can be used for on-line applications as the EKF method. Table 1 compares the results of using NetDDR and EKF

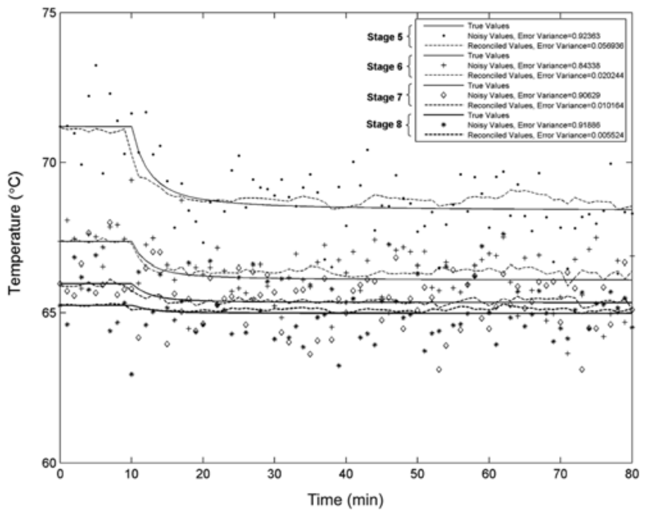


Fig. 20. Results of DDR on temperature of stages 5 to 8 using trained ANN with one delay.

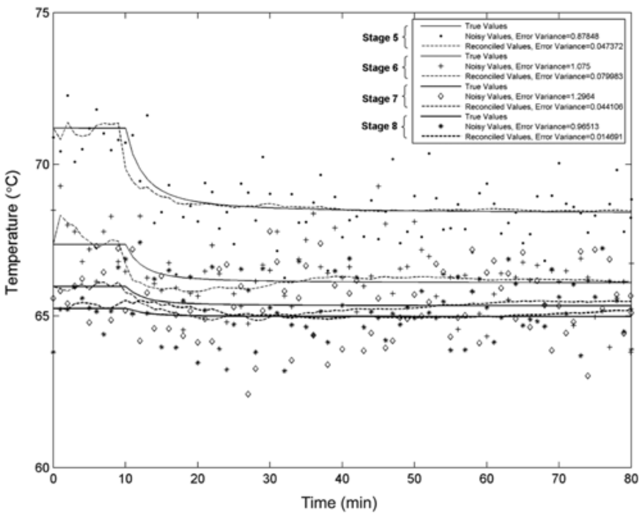


Fig. 21. Results of DDR on temperature of stages 5 to 8 using EKF method.

Table 1. Comparison of the results of NetDDR (with one input delay) and EKF methods for NDDR of temperature measurements

Method	Standard deviation of errors of reconciled data for different stages				Running time for each step of DR (seconds)
	5	6	7	8	
NetDDR	0.24	0.14	0.10	0.07	0.11
EKF	0.22	0.28	0.21	0.12	0.25

methods for NDDR of temperature measurements.

CONCLUSIONS

The object-oriented method has the benefits of *complexity management* and *code reusability* and can be used to manage complexity of DR problems especially on nonlinear dynamic systems. Complexity management property of object-oriented method makes it possible that large scale and complex operating systems can be divided into simple subsystems, and DR can be applied on simple subsystems instead of the whole system. In this work, object-oriented modeling methodology is used for encapsulating different types of DR problems in classes.

The main classes that are defined are *Constraints* for manipulating constraints, *Optimization* for defining and applying different methods of optimization, *DR* the main class for all types of DR, *SSDR*, *LSSDR* and *NSSDR* for steady-state DR in linear and nonlinear cases, and *DDR*, *LDDR* and *NDDR* for linear and nonlinear dynamic DR.

The *DDRMethod* class is developed to let users to create and develop their own classes and apply their methods for DDR by inheriting from it without any need to pay attention to other aspects of developing a DDR software tool. Users can derive their classes from it and customize its methods and apply their own methods for DDR. *Inheritance* and *aggregation* mechanisms provide relationships between the above mentioned classes.

An academic software tool, DCON, is developed for performing different types of DR by coding classes declared in this paper. Two parts of the software for performing LSSDR and NDDR are discussed briefly. For performing NDDR, the DDE and COM data communication mechanisms can be used between the software and simulation programs to communicate between each other.

As an example, the class *DDRByKalman* was developed by applying KF and EKF methods for DDR. Another method, NetDDR, which uses ANNs for DDR, was also implemented. A class with the name of *NetDDRMethod* was inherited from *DDRMethod* class and used NetDDR method for DDR.

To show the performance of the described object-oriented method and classes defined for DDR, the illustration example of DDR on temperature measurements of a distillation column was presented using EKF and NetDDR methods. Both methods showed high performance in terms of removing noise from measurements and speed of DDR. NetDDR method is faster and variance of errors in reconciled data using NetDDR method is smaller than that of the EKF method.

NOMENCLATURE

D_n	: product flow rate from stage n [gmol/min]
E_n	: plate efficiency for stage n
F_n	: feed flow rate to stage n [gmol/min]
$G_{k,i}$	: interaction parameter between components k and i, Eq. (16)
$h_{F,n}$	: enthalpy of feed stream entering stage n [J/gmol]
K_i	: K-value of component i at system temperature and pressure
$K_{i,n}$	: K-value of component i on stage n
L_n	: liquid flow rate on stage n [gmol/min]
$M_{c,n}$	: heat capacity of stage n [J/K]
m	: total number of components=2
P	: total pressure [Pa]
P_i^*	: vapor pressure of component i at system temperature [Pa]
$Q_{c,n}$	: heat entered to stage n [J/min]
$q_{c,n}$	: heat loss from stage n [J/min]
$S_{L,i}$	: liquid heat capacity of component i [J/gmol·K]
$S_{V,i}$	: vapor heat capacity of component i [J/gmol·K]
T	: system temperature [K]
T_n	: temperature on stage n [K]
t	: time [min]
V_n	: gas flow rate on stage n [gmol/min]
v_{Ln}	: liquid hold-up on stage n [m^3]
v_{Vn}	: gas hold-up on stage n [m^3]
x_i	: molar composition of component i in liquid phase
$x_{i,n}$	: molar composition of component i in liquid stream on stage n
$x_{i,n}(e)$	: molar composition of component i in liquid phase in equilibrium with $y_{i,n}$ at system temperature and pressure on stage n
$x_{F,n}$	: molar composition of methanol in feed stream entering to stage n
y_i	: molar composition of component i in vapor phase
$y_{i,n}$	: molar composition of component i in vapor stream on stage n

Greek Letters

γ_i	: activity coefficient of component i at system temperature
λ_i	: heat of vaporization of component i [J/gmol]

ρ_{Ln}, ρ_{Vn}: liquid and vapor molar density on stage n, respectively [gmol/m^3]

Subscripts

i, j, k	: component indices
n	: stage index

REFERENCES

1. J. A. Romagnoli and M. B. Sanchez, *Data processing and reconciliation for chemical process operations*, Academic Press, San Diego, California (2000).
2. C. M. Crowe, Y. A. G. Campos and A. Hrymak, *AIChE J.*, **29**, 881 (1983).
3. C. M. Crowe, *AIChE J.*, **32**, 616 (1986).
4. M. J. Liebman, T. F. Edgar and L. S. Lasdon, *Computers Chem. Engng.*, **16**(10/11), 963 (1992).
5. I. W. Kim, S. Park and T. F. Edgar, *Korean J. Chem. Eng.*, **13**, 211 (1996).
6. K. Meert, *Artificial Intelligence in Engineering*, **12**, 213 (1998).
7. J. Chen and J. A. Romagnoli, *Computers Chem. Engng.*, **22**(4/5), 559 (1998).
8. Z. H. Abu-el-zeet, V. M. Becerra and P. D. Roberts, *Computers Chem. Engng.*, **26**, 921 (2002).
9. J. D. Kelly, *Comp. Chem. Eng.*, **28**, 2837 (2004).
10. S. Bai, J. Thibault and D. D. McLean, *Journal of Process Control*, **16**(5), 485 (2006).
11. M. O'Docherty, *Object-oriented analysis and design, understanding system development with UML 2.0*, John Wiley and Sons Ltd., Chichester, England (2005).
12. S. Narasimhan and C. Jordache, *Data reconciliation and gross error detection: An intelligent use of process data*, Gulf Professional Publishing, Houston, Texas, November (1999).
13. M. S. Grewal and A. P. Andrews, *Kalman filtering: Theory and practice using MATLAB, second edition*, John Wiley and Sons Inc. (2001).
14. A. Yoo, T. C. Lee and D. R. Yang, *Korean J. Chem. Eng.*, **21**, 753 (2004).
15. R. G. Brown and P. Y. C. Hwang, *Introduction to random signals and applied Kalman filtering*, 3rd ed., John Wiley & Sons Inc., New York (1997).
16. D. M. Himmelblau, *Korean J. Chem. Eng.*, **17**, 373 (2000).
17. M. T. Hagan, O. De Jesus and R. Schultz, Training Recurrent Networks for Filtering and Control, Chapter 12 in *Recurrent neural networks: Design and applications*, L. Medsker and L. C. Jain, Eds., CRC Press, 311-340 (1999).
18. K. S. Narendra and K. Parthasarathy, *IEEE Transactions on Neural Networks*, **2**(2), 252 (1991).
19. K. S. Narendra and S. Mukhopadhyay, *IEEE Transactions on Neural Networks*, **8**, 475 (1997).
20. A. Z. Mehrabani, *Non-linear parameter estimation of distillation column*, M.Sc. Thesis, University of Wales, Department of Chemical Engineering, Nov. (1986).
21. J. Shin, M. Lee and S. Park, *Korean J. Chem. Eng.*, **15**, 667 (1998).
22. J. J. Moré, The Levenberg-Marquardt Algorithm: Implementation and Theory, *Numerical analysis*, ed. G. A. Watson, *Lecture Notes in Mathematics 630*, Springer Verlag, 105-116 (1977).